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

Textbook under Construction
Version of July 13, 2024

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. Based 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. 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

$$\mathsf{B} ::= \mathsf{true} \mid \mathsf{false}$$

introduces three typed constants called **constructors**:

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

The constructors represent the type B and its two values true and false. 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:

$$\begin{aligned} ! &: \mathsf{B} \to \mathsf{B} \\ !\,\mathsf{true} &:= \mathsf{false} \\ !\,\mathsf{false} &:= \mathsf{true} \end{aligned}$$

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

$$!\,!\,!\,\mathsf{true} = !\,!\,\mathsf{false} = !\,\mathsf{true} = \mathsf{false}$$

by rewriting with the first, the second, and again with the first defining equation of !. Note that $!\,!\,!\,\mathsf{true}$ is to be read as $!(!(!\,\mathsf{true}))$, and that the first rewrite step replaces the subterm $!\,\mathsf{true}$ with false. Computation in Coq is logical and is used in proofs. For instance, the equation

$$!\,!\,!\,\mathsf{true} = !\,\mathsf{true}$$

follows by computation:

$$\begin{aligned} & !\,!\,!\,\mathsf{true} \\ =\; & !\,!\,\mathsf{false} \\ =\; & !\,\mathsf{true} \\ =\; & \mathsf{false} \end{aligned} \qquad\qquad \begin{aligned} & !\,\mathsf{true} \\ =\; & \mathsf{false} \end{aligned}$$

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 $!!\mathsf{true} = \mathsf{true}$ and $!!\mathsf{false} = \mathsf{false}$, which both hold by computational equality.

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

$$\begin{array}{rcl} \& &:& \mathsf{B} \to \mathsf{B} \to \mathsf{B} \\ \mathsf{true} \mathbin{\&} y &:=& y \\ \mathsf{false} \mathbin{\&} y &:=& \mathsf{false} \end{array} \qquad\qquad \begin{array}{rcl} | &:& \mathsf{B} \to \mathsf{B} \to \mathsf{B} \\ \mathsf{true} \mid y &:=& \mathsf{true} \\ \mathsf{false} \mid y &:=& y \end{array}$$

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 $\mathsf{false} = \mathsf{false}$. 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 $0, 1, 2, \ldots$

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

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.

Truncating subtraction gives us a test for comparisons $x \le y$ of numbers. We have $x \le y$ if and only if $x - y = 0$.

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

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\multicolumn{3}{l}{\cdot : \mathsf{N} \to \mathsf{N} \to \mathsf{N}} & \multicolumn{3}{l}{\hat{} : \mathsf{N} \to \mathsf{N} \to \mathsf{N}} \\
0 \cdot y & := & 0 & x^0 & := & 1 \\
\mathsf{S}x \cdot y & := & y + x \cdot y & x^{\mathsf{S}n} & := & x \cdot x^n
\end{array}$$

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:

$$\begin{array}{rcl}
\multicolumn{3}{l}{\& : \mathsf{B} \to \mathsf{B} \to \mathsf{B}} \\
\mathsf{true} \mathbin{\&} \mathsf{true} & := & \mathsf{true} \\
\mathsf{true} \mathbin{\&} \mathsf{false} & := & \mathsf{false} \\
\mathsf{false} \mathbin{\&} \mathsf{true} & := & \mathsf{false} \\
\mathsf{false} \mathbin{\&} \mathsf{false} & := & \mathsf{false}
\end{array}$$

a) Prove that the symmetric function satisfies the defining equations for the standard boolean conjunction function ($\mathsf{true} \mathbin{\&} y = y$ and $\mathsf{false} \mathbin{\&} y = \mathsf{false}$).

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:

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

		$x + 0 = x$	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 table for Equation 1.1

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

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

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 table** 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.

		$x + y - y = x$	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 table for Equation 1.4

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 table 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 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 tables is that proof tables 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 tables. The induction tactic displays the subgoals and automatically provides the inductive hypothesis. Except for the initial claim, all the equations appearing in the proof tables are displayed automatically by Coq, saving a lot of tedious writing. Replay all proof tables 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 table comes with three columns listing assumptions, claims, and proof actions.[1] Subgoals are marked by hierarchical numbers and horizontal lines. Our

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

proof tables 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 table for Equation 1.2. Follow the layout of Figure 1.2.

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 table.

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 $x \le y$ 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:

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

		$\forall y.\ Dxy = (x - y) + (y - x)$	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	$\mathsf{IH} : \forall y. \cdots$	$\forall y.\ D(\mathsf{S}x)y = (\mathsf{S}x - y) + (y - \mathsf{S}x)$	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 table for a proof with a quantified inductive hypothesis

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

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

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 table 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 5 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 Preview of Proof Rules

We have seen a number of proofs using structural case analysis and structural induction on numbers. Our presentation is informal and leaves important issues unexplained, in particular as it comes to structural induction on numbers. Latter in this text, we will derive formal proof rules for structural case analysis and induction from first principle in type theory. We now give a preview of the formal proof rules explaining their meaning informally.

The rule for structural case analysis on booleans takes the form

$$\mathsf{E_B} : \forall p^{\mathsf{B} \to \mathbb{P}}.\ p\,\mathsf{true} \to p\,\mathsf{false} \to \forall x.px$$

The rule says that we can prove a proposition px for all booleans x by proving it for true and false. Type-theoretically, we see $\mathsf{E_B}$ as a function that given proofs of the propositions $p\,\mathsf{true}$ and $p\,\mathsf{false}$ yields a proof of the proposition px for every boolean x.

The rule for structural case analysis on numbers takes the form

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

The rule says that we can prove a proposition pn for all numbers n by giving proofs for 0 and all successors $\mathsf{S}n$. Type-theoretically, we see $\mathsf{M_N}$ as a function that given proofs of the propositions $p0$ and $\forall n.\ p(\mathsf{S}n)$ yields a proof of the proposition $\forall n.pn$.

The rule for structural induction on numbers takes the form

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

It says that a proposition pn can be proved for all numbers n by giving proofs for the *zero case* $p0$ and the *successor case* $\forall n.\ pn \to p(\mathsf{S}n)$. A proof of the successor case $\forall n.\ pn \to p(\mathsf{S}n)$ is a function that given a number n and a proof of pn yields a proof of the proposition $p(\mathsf{S}n)$. In the context of the rule $\mathsf{E_N}$ the proposition pn acts as an assumption which is usually referred to as inductive hypothesis.

We may see the proof of the successor case of the induction rule as a method that for every n upgrades a proof of pn to a proof of $p(\mathsf{S}n)$. By iterating this method n-times on a proof of $p0$

$$p0,\ p1,\ p2, \dots, pn$$

we can obviously get a proof of pn for every n.

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

1.7 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:

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

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

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

(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

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

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 $F(n + b)$:

$$\begin{aligned} &F' : \mathsf{N} \to \mathsf{B} \to \mathsf{N} \\ &\quad F'0\,\mathsf{false} := 0 \\ &\quad F'0\,\mathsf{true} := 1 \\ &\quad F'(\mathsf{S}n)\,\mathsf{false} := F'n\,\mathsf{true} \\ &\quad F'(\mathsf{S}n)\,\mathsf{true} := F'n\,\mathsf{false} + F'n\,\mathsf{true} \end{aligned}$$

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

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

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 1.10.8 will obtain a Fibonacci function by iteration on pairs, and Exercise 1.12.5 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 30).

Exercise 1.7.1 Prove $E(n \cdot 2) = \mathsf{true}$.

Exercise 1.7.2 Verify that $Fn := F'n\,\mathsf{false}$ satisfies the specifying equations for the Fibonacci function.

Exercise 1.7.3 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.

1.8 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:

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

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:

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

The **polymorphic value constructor** pair comes with a **polymorphic function type** saying that pair takes four arguments, where the first argument X and the second argument Y 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:

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

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

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

Note that the 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.

1.9 Argument Inference and Implicit Arguments

If we look at the type of the polymorphic pair constructor

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

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. In Coq, we can write the **underline symbol** "_" for arguments the proof assistant should derive from the context. For instance, we may write $\mathsf{pair}\,\_\,\_\,5\,\mathsf{true}$ for the term $\mathsf{pair}\,\mathsf{N}\,\mathsf{B}\,5\,\mathsf{true}$. More generally, we may write

$$\mathsf{pair}\,\_\,\_\,s\,t$$

for the term $\mathsf{pair}\,u\,v\,s\,t$ if the types of the terms s and t can be determined as u and v. We can now define the usual pair notation for applications of the constructor pair:

$$(s,t)\ :=\ \mathsf{pair}\,\_\,\_\,s\,t$$

We remark that Coq will accept an input only if it can derive all terms specified with the underline symbol.

Coq offers the possibility to declare arguments of functional constants as **implicit arguments**. This has the effect, that the implicit arguments will always be inserted as underlines. For instance, if we declare the type arguments of

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

as implicit arguments, we can write

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

for the otherwise bloated equation

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

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

With implicit arguments, argument inference, and the notation for pairs we can write the definition of swap as follows:

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

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

1. Polymorphic function types for functions taking types as arguments.
2. Argument inference requested with the underline symbol.
3. Notation for pairs and pair types.
4. Implicit arguments for polymorphic functions.

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

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

Note that X and Y are accommodated as implicit arguments of π_1 and π_2. We can now state the **η-law** for pairs

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

To prove the η-law, one discriminates on the variable a, which replaces it with a general pair (x, y). This leaves us with the claim

$$(\pi_1(x, y), \pi_2(x, y)) = (x, y)$$

which follows by computational equality. We will refer to discriminations where there is only a single constructor as **destructurings**.

Exercise 1.9.1 Write the definitions of the projections and the η-law and in full detail not using the notations (x, y) and $X \times Y$, underlines, or implicit arguments. Give the types of all variables.

Exercise 1.9.2 Let a be a variable of type $X \times Y$. Write proof tables for the equations $\mathsf{swap}\,(\mathsf{swap}\,a) = a$ and $(\pi_1 a, \pi_2 a) = a$.

1.10 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:

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

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

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

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

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

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

Exercise 1.10.1 Check that $\mathsf{iter}\;\mathsf{S}\;2 = \lambda x.\,\mathsf{S}(\mathsf{S}x)$ holds by computational equality.

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

Figure 1.4: Correctness of multiplication with iter

Exercise 1.10.2 Prove $n + x = \mathsf{iter}\ \mathsf{S}\ n\ x$ and $x^n = \mathsf{iter}\ (\cdot\, x)\ n\ 1$ by induction.

Exercise 1.10.3 Check that the plain function

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

satisfies the defining equations for inductive addition

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

by computational equality.

Exercise 1.10.4 (Shift) Prove $\mathsf{iter}\ f\ (\mathsf{S}n)\ x = \mathsf{iter}\ f\ n\ (fx)$.

Exercise 1.10.5 (Tail recursive iteration) Define a tail recursive version of iter and verify that it agrees with iter.

Exercise 1.10.6 (Even) The term $!^n\,\mathsf{true}$ 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 1.10.7 (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 1.10.8 (Fibonacci with iteration) Fibonacci numbers (§1.7) 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.

1.11 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]

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

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

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

Note that the three specifying equations 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 1.11.1 (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.

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

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

Figure 1.5: Unfolding functions for the Fibonacci and Ackermann functions

Exercise 1.11.2 (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.

1.12 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 §1.7 and §1.11.

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.

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:

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

The same is true for the Ackermann function.

Exercise 1.12.1 Verify with the proof assistant that the realizations of the Fibonacci function defined in §1.7 and Exercise 1.10.8 satisfy the unfolding equation for the specifying unfolding function.

Exercise 1.12.2 Verify with the proof assistant that the realizations of the Ackermann function defined in §1.11 satisfies the unfolding equation for the specifying unfolding function.

Exercise 1.12.3 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 1.12.4 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 1.12.5 (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 1.10.8.

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 be passed 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.

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, argument inference, 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 presents the basic fragment of computational type theory (CTT) in a nutshell. The basic fragment covers the type and function definitions we have seen in Chapter 1. The next two chapters will extend the basic fragment such that propositions and proofs are covered.

We start with the grammar and the typing rules for terms obtained with applications and dependent function types. The typing rules yield a type checking algorithm assigning unique types to well-typed terms.

We continue with inductive type definitions and inductive function definitions. The defining equations of inductive functions provide for rewriting of terms, a form of computation we call reduction. The essential properties of reduction are type preservation, termination, unique normal forms, and canonicity. The properties will be maintained by all further extensions of CTT. The properties crucially depend on the assumption that only well-typed terms are admitted, a discipline proof assistants will enforce.

The basic fragment also comes with plain definitions, lambda abstractions, let expressions, and matches. Matches and plain definitions can be expressed with inductive function definitions, and let expressions can be expressed with lambda abstractions.

The proof assistant Coq we are targeting deviates from the architecture of CTT in that it obtains inductive functions with recursive abstractions and native matches rather than with defining equations. Thus we also consider recursive abstractions.

2.1 Simply Typed Terms

We start with a small fragment of CTT featuring simple function types and applications. The **terms** of the simply typed fragment are given by the grammar

$$s, t, u, v \ ::= \ c \mid \mathbb{T} \mid s \to t \mid st$$

The letter c ranges over **constants**. The term $\mathbb{T}$ is called **universe** and serves as the type of all types. A **function type** $s \to t$ applies to functions taking arguments of type s and returning results of type t. An **application** st describes the application of the function described by the term s to the argument described by the term t.

The **typing rules** for simply typed terms are as follows:

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

The rule for $\mathbb{T}$ says that $\mathbb{T}$ has type $\mathbb{T}$ (an arrangement to be refined in § 5.3). The rule for function types says that $s \to t$ has type $\mathbb{T}$ if the constituents s and t have type $\mathbb{T}$. The rule for applications st says that s can be applied to t if s has a function type $u \to v$ and t has type u. In this case the application has type v.

A term s is **well-typed** and **has type t** if a typing $\vdash s : t$ can be derived with the rules. Given unique types for the involved constants, well-typed terms have **unique types**. For instance, we can derive the typings

$$\begin{aligned} &\vdash \mathbb{T} \to \mathbb{T} \;:\; \mathbb{T} \\ &\vdash \mathbb{T} \to (\mathbb{T} \to \mathbb{T}) \;:\; \mathbb{T} \end{aligned}$$

An example for an ill-typed term (i.e., not well-typed term) is the application $\mathbb{T}\,\mathbb{T}$.

Given the typings of the constructors for numbers

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

we can derive the typings

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

using the typing rules.

Type theory comes with the principle that only well-typed terms are admitted. **Type checking** is an algorithm that determines whether a term is well-typed. In case a term is well-typed, the unique type of the term is determined. The proof assistant always checks well-typedness of a given term and computes its unique type. Terns that are not well-typed are rejected. Examples for ill-typed terms are the applications $\mathsf{S}\,\mathbb{T}$ and $S\,\mathsf{S}$. 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.

CTT treats types and functions like all other values. We say that types and functions are *first-class citizens* in CTT.

Parentheses in **right-nested function types** may be omitted:

$$\begin{array}{ccc} \mathbb{T} \to \mathbb{T} \to \mathbb{T} & \rightsquigarrow & \mathbb{T} \to (\mathbb{T} \to \mathbb{T}) \\ \mathsf{N} \to \mathsf{N} \to \mathsf{N} & \rightsquigarrow & \mathsf{N} \to (\mathsf{N} \to \mathsf{N}) \\ s \to t \to u & \rightsquigarrow & s \to (t \to u) \end{array}$$

Correspondingly, parentheses may be omitted in **left-nested applications**:

$$s\,t\,u \quad \rightsquigarrow \quad (s\,t)\,u$$

2.2 Dependently Typed Terms

A distinguishing feature of CTT are **dependent function types** $\forall x{:}s.\,t$, which may also be written as $\forall x^s.\,t$. A **dependent function type** $\forall x{:}u.\,v$ applies to functions taking arguments of type u and returning a result of type v^x_s for an argument s. The notation v^x_s stands for the term that is obtained from the term v by replacing the variable x with the term s. We speak of **substitution**.

CTT sees simple function types $s \to t$ as dependent function types $\forall x{:}s.\,t$ where the target 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 $\forall x{:}\mathsf{N}.\,\mathsf{N}$. You can check this fact with the proof assistant.

Dependently typed terms are given by the grammar

$$s, t, u, v \ ::= \ c \mid x \mid \mathbb{T} \mid \forall x{:}s.\,t \mid st$$

where c ranges over constants and x ranges over **variables**. The typing rules for dependently typed terms are as follows:

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

Using the typing rules for dependent types, we can derive the typing

$$\vdash \ \mathbb{T} \to \mathbb{T} \to \mathbb{T} \ : \ \mathbb{T}$$

which in turn validates the typing

$$\vdash \ \mathsf{Pair} \ : \ \mathbb{T} \to \mathbb{T} \to \mathbb{T}$$

of the type constructor for pairs. We can now derive the typing

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

which validates the typing of the value constructor for pairs:

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

Given the typings of the constructors for numbers and pairs, we can derive the following typings:

$$\begin{aligned}
&\vdash \ \mathsf{Pair}\,\mathsf{N} \ : \ \mathbb{T} \to \mathbb{T}\\
&\vdash \ \mathsf{pair}\,\mathsf{N} \ : \ \forall Y{:}\mathbb{T}.\ \mathsf{N} \to Y \to \mathsf{Pair}\,\mathsf{N}\,Y\\
&\vdash \ \mathsf{pair}\,\mathsf{N}\,\mathsf{N} \ : \ \mathsf{N} \to \mathsf{N} \to \mathsf{Pair}\,\mathsf{N}\,\mathsf{N}\\
&\vdash \ \mathsf{pair}\,\mathsf{N}\,\mathsf{N}\,0\ (\mathsf{S}0) \ : \ \mathsf{Pair}\,\mathsf{N}\,\mathsf{N}
\end{aligned}$$

The names of the variables of dependent function types $\forall x{:}s.\,t$ do not matter, as is standard for **bound variables** in programming languages and mathematics. For instance, the terms $\forall X.X \to X$ and $\forall Y.Y \to Y$ are considered equal.

Note that the typing rules admit any type $\vdash u : \mathbb{T}$ as argument type of a dependent function type $\forall x{:}u.v$. So far we have only seen examples where u is $\mathbb{T}$ but dependent function types $\forall x{:}u.v$ where u is not $\mathbb{T}$ will turn out to be important (for instance, u may be the type of numbers N).

Exercise 2.2.1 Convince yourself that the terms

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

are well-typed given the typings for the constructors for numbers and pairs. In each case determine the type of the term.

2.3 Inductive Type Definitions

Inductive type definitions introduce typed constants called constructors. We have already seen inductive definitions for a type of numbers and a family of pair types:

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

Every inductive type definition 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.

We say that the value constructor pair for pairs has two **parametric** and two **proper arguments**. The parametric arguments are the leading arguments shared with the type constructor Pair.

We take the opportunity to define two particular inductive types called **unit** and **void** by listing their constructors:

$$\begin{aligned} \mathbb{1} &: \mathbb{T} \\ \mathsf{U} &: \mathbb{1} \\ \mathbb{0} &: \mathbb{T} \end{aligned}$$

Note that $\mathbb{1}$ (unit) has exactly one value U, and that $\mathbb{0}$ (void) has no value at all.

2.4 Inductive Function Definitions

Inductive function definitions define functions by case analysis on one or more inductive arguments called **discriminating arguments**. We will speak of **inductive functions**. Figure 2.4 defines four inductive functions that will serve as examples. An inductive function definition first declares the name (a constant) and the type of the defined function. Then **defining equations** are given realizing a **disjoint** and **exhaustive** case analysis. Note that add, swap, and Fib have one discriminating argument, while sub has two discriminating arguments. There is only one defining equation for swap since there is only one value constructor for pairs.

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. For this reason the parametric arguments of the constructor pair in the pattern of the defining equation of swap are omitted (the parametric arguments may also be indicated with the underline symbol).

The defining equations for a function must be disjoint and exhaustive. In the simplest case with only one discriminating argument and no nested discrimination (e.g., add and swap in Figure 2.4), there will be exactly one defining equation for every value constructor of the type of the discriminating argument. Fib is an example for a **nested discrimination**.

$$\begin{aligned}
&\mathsf{add} : \ \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\
&\qquad \mathsf{add}\, 0\, y \ := \ y \\
&\qquad \mathsf{add}\,(\mathsf{S}x)\, y \ := \ \mathsf{S}(\mathsf{add}\, x\, y) \\[1em]
&\mathsf{sub} : \ \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\
&\qquad \mathsf{sub}\, 0\, y \ := \ 0 \\
&\qquad \mathsf{sub}\,(\mathsf{S}x)\, 0 \ := \ \mathsf{S}x \\
&\qquad \mathsf{sub}\,(\mathsf{S}x)\,(\mathsf{S}y) \ := \ \mathsf{sub}\, x\, y \\[1em]
&\mathsf{swap} : \ \forall X^{\mathbb{T}}.\forall Y^{\mathbb{T}}.\ \mathsf{Pair}XY \to \mathsf{Pair}YX \\
&\qquad \mathsf{swap}\, XY(\mathsf{pair}\, xy) \ := \ \mathsf{pair}YXyx \\[1em]
&\mathsf{Fib} : \ (\mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N} \\
&\qquad \mathsf{Fib}\, f\, 0 \ := \ 0 \\
&\qquad \mathsf{Fib}\, f\,(\mathsf{S}0) \ := \ 1 \\
&\qquad \mathsf{Fib}\, f\,(\mathsf{S}(\mathsf{S}n)) \ := \ fn + f(\mathsf{S}n)
\end{aligned}$$

Figure 2.1: Inductive function definitions

The defining equations of an inductive function must all take the same number $n \geq 1$ of arguments. We speak of the **arity** of an inductive function. In Figure 2.4, add, sub, and Fib have arity 2 and swap has arity 4.

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 a defining equation works as usual.

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 2.4, only the variable x in the defining equations for add and sub qualifies for recursion. We refer to this severely restricted form of recursion as **structural recursion**.

2.5 Reduction

The defining equations of an inductive function serve as **reduction rules** that rewrite applications of the defined function. For instance, the application $\mathsf{sub}\,(\mathsf{S}s)\,(\mathsf{S}t)$ can be **reduced** to $\mathsf{sub}\, s\, t$ using the third defining equation of sub.

Things are arranged such that at most one defining equation applies to an application (disjointness), and such that every application of an inductive function can be reduced (exhaustiveness) provided all arguments required by the defining equations are given such that the discriminating arguments start with value constructors. We refer to the process of applying reductions rules as **reduction**. We see reduction as computation and refer to reduction rules also as **computation rules**.

We consider reduction only for well-typed terms. Reduction has the important property that it preserves well-typedness and the type of a term.

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 normal and **closed** (i.e., no unbound variables) are called **canonical terms**. Reduction preserves closedness of terms.

We now formulate the **key properties** of reduction:

- **Termination** Reduction always terminates.
- **Unique normal forms** Terms reduce to unique normal forms.
- **Type preservation** Reduction preserves types: If a term of type t is reduced, the obtained term is again of type t.
- **Canonicity** Canonical 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 can be obtained 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 defining equations are needed for type preservation.

Given an inductive type, we refer to the canonical terms of this type as the **values** of t. This speak is justified by the canonicity property of reduction. The values of the type N are exactly the terms 0, $\mathsf{S}0$, $\mathsf{SS}0$, ...

Exercise 2.5.1 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

$$\mathsf{sub}\,(\mathsf{S}0)\,(\mathsf{S}y) \succ \mathsf{sub}\,0\,y \succ 0$$

for a reduction chain to normal form using $s \succ t$ as notation for a reduction step.

2.6 Plain Definitions

A **plain definition** introduces a constant with a type, an arity $n \geq 0$, and a single defining equation not discriminating on an argument:

$$\begin{aligned} &c : t \\ &cx_1 \dots x_n \;:=\; s \end{aligned}$$

Chapter 1 contains several examples for plain definitions. Plain definitions can be expressed as inductive function definitions discriminating on an extra argument of type $\mathbb{1}$:

$$\begin{aligned} &c : \mathbb{1} \to t \\ &c\,\mathsf{U}\,x_1 \dots x_n \;:=\; s \end{aligned}$$

Expressing plain definitions as inductive function definitions has the benefit that type checking of plain definitions is explained as type checking of inductive functions. Moreover, reduction of plain function applications is explained with reduction of inductive function applications. Most importantly, expressing plain definitions as inductive function definitions has the advantage that the properties of the type theory remain unchanged.

Exercise 2.6.1 Argue why plain definitions cannot be recursive.

Exercise 2.6.2 Recall the definition of iter (§1.10). Explain the difference between the following plain definitions:

$$\begin{aligned} A \;&:=\; \mathsf{iter}\,\mathsf{S} \\ Bxy \;&:=\; \mathsf{iter}\,\mathsf{S}\,x\,y \end{aligned}$$

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

2.7 Lambda Abstractions

A key ingredient of computational type theory are terms of the form

$$\lambda x : t.\, s$$

called **lambda abstractions**. Lambda abstractions describe functions with a single argument. They come with an **argument variable** x, an **argument type** t, and a **body** s. A lambda abstraction does not give a name to the function it describes.

The **typing rule** for lambda abstractions is

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

A nice example is the typing

$$\vdash\; \lambda X^{\mathbb{T}}.\lambda x^X.x \;:\; \forall X^{\mathbb{T}}.\, X \to X$$

of 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 featuring two consecutive β-reductions of an application of the polymorphic identity function:

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

As with dependent function types, the particular name of an argument variable of a lambda abstractions 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. One speaks of **alpha renaming** of bound variables in the textual representation of terms.

A complex operation the β-reduction rules builds on is **substitution** s^x_t. Substitution must be performed such that local binders do not **capture free variables** (a free variable is an unbound variable). 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 free variable y. We speak of **capture-free substitution**. We remark that capture-free substitution s^x_t is also used with the dependent typing rule for applications and with reduction steps rewriting with defining equations of inductive functions.

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:

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

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 all key properties: unique types, unique normal forms, termination, type preservation, and canonicity.

Exercise 2.7.1 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 variable x makes the term well-typed.

2.8 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 is

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

The typing rule for let expressions is

$$\frac{\vdash t : \mathbb{T} \qquad \vdash s : t \qquad x : t \vdash u : v \qquad x \text{ not in } v}{\vdash\ \text{LET } x : t = s \text{ IN } u\ :\ v}$$

It turns out that in the system we are considering let expressions can be expressed with β-redexes:

$$\text{LET } x : t = s \text{ IN } u \quad\leadsto\quad (\lambda x : t.u)\, s$$

This syntactic transformation explains the typing rule and the reduction rule for let expressions.

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

2.9 Matches

Matches are a defined notation for applicative expressions providing structural case analysis for inductive types. A match has a **clause** for every constructor of the underlying inductive type.

Matches for numbers take the form

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

and reduce with the derived reduction rules

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

Matches for booleans take the form

$$\textsc{match}\ s\ [\,\mathsf{true} \Rightarrow u \mid \mathsf{false} \Rightarrow v\,]$$

and reduce with the derived reduction rules

$$\begin{aligned} \textsc{match}\ \mathsf{true}\ [\,\mathsf{true} \Rightarrow u \mid \mathsf{false} \Rightarrow v\,] &\succ u \\ \textsc{match}\ \mathsf{false}\ [\,\mathsf{true} \Rightarrow u \mid \mathsf{false} \Rightarrow v\,] &\succ v \end{aligned}$$

Matches for pairs take the form

$$\textsc{match}\ s\ [\,(x,y) \Rightarrow u\,]$$

and reduce with the derived reduction rule

$$\textsc{match}\ (s,t)\ [\,(x,y) \Rightarrow u\,] \succ (\lambda xy.u)\,s\,t$$

Matches are accommodated as applications of **match functions**:

$$\begin{aligned} \textsc{match}\ s\ [\,0 \Rightarrow u \mid \mathsf{S}x \Rightarrow v\,] &\rightsquigarrow \mathsf{M}_{\mathsf{N}}\,\_\,s\,u\,(\lambda x.v) \\ \textsc{match}\ s\ [\,\mathsf{true} \Rightarrow u \mid \mathsf{false} \Rightarrow v\,] &\rightsquigarrow \mathsf{M}_{\mathsf{B}}\,\_\,s\,u\,v) \\ \textsc{match}\ s\ [\,(x,y) \Rightarrow u\,] &\rightsquigarrow \mathsf{M}_{\times}\,\_\,\_\,\_\,s\,(\lambda xy.u) \end{aligned}$$

Match functions are inductive functions defined as one would expect form the derived reduction rules:

$$\begin{aligned} \mathsf{M_N} :\ & \forall Z^{\mathbb{T}}.\ \mathsf{N} \to Z \to (\mathsf{N} \to Z) \to Z \\ \mathsf{M_N}\, Z\, 0\, e_1 e_2 \ &:=\ e_1 \\ \mathsf{M_N}\, Z\, (\mathsf{S}x)\, e_1 e_2 \ &:=\ e_2 x \end{aligned} \qquad \begin{aligned} \mathsf{M_B} :\ & \forall Z^{\mathbb{T}}.\ \mathsf{B} \to Z \to Z \to Z \\ \mathsf{M_B}\, Z\, \mathsf{true}\, e_1 e_2 \ &:=\ e_1 \\ \mathsf{M_B}\, Z\, \mathsf{false}\, e_1 e_2 \ &:=\ e_2 \end{aligned}$$

$$\begin{aligned} \mathsf{M}_\times :\ & \forall XYZ^{\mathbb{T}}.\ X \times Y \to (X \to Y \to Z) \to Z \\ \mathsf{M}_\times\, XYZ\, (x,y)\, e \ &:=\ exy \end{aligned}$$

Note that the definition of the match notation as an application of a match function provides both for type checking and reduction. Informally, we may summarize the typing of matches as follows: 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**:

$$\text{IF } s \text{ THEN } t_1 \text{ ELSE } t_2 \quad\rightsquigarrow\quad \text{MATCH } s\ [\,\mathsf{true} \Rightarrow t_1 \mid \mathsf{false} \Rightarrow t_2\,]$$

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. For numbers we have

$$\text{IF } s \text{ THEN } t_1 \text{ ELSE } t_2 \quad\rightsquigarrow\quad \text{MATCH } s\ [\,0 \Rightarrow t_1 \mid \mathsf{S}\,\_ \Rightarrow t_2\,]$$

Recall that $x - y = 0$ iff $x \le y$. Thus IF $s_1 - s_2$ THEN t_1 ELSE t_2 is a conditional testing $s_1 \le s_2$. We may use the notation

$$\text{IF } s_1 \le s_2 \text{ THEN } t_1 \text{ ELSE } t_2 \quad\rightsquigarrow\quad \text{MATCH } (s_1 - s_2)\ [\,0 \Rightarrow t_1 \mid \mathsf{S}\,\_ \Rightarrow t_2\,]$$

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

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

2.10 Terms and Values

We see terms as **syntactic descriptions** of **informal semantic objects** called **values**. Examples for values are numbers, functions, and types. Reduction of a term preserves the value of the term, and also the type of the term. Informally, 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**, and values of $\mathbb{T}$ are referred to as **types**.

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 well-typed term.

Recall that a term is **closed** if it has no free variables, and **canonical** if it is closed and irreducible. CTT 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 a function type (obtained with $\forall$), or a universe (so far we have $\mathbb{T}$). A **constant** is either a constructor or the name of an inductive function.

CTT 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.

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 canonical terms of the type as the values of the type. For function types the situation is more involved since semantically we may want to consider two functions as equal if they agree on all arguments.

2.11 Function Definitions in Coq

The proof assistant Coq realizes function definitions based on **plain definitions** of the form

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

using lambda abstractions, native matches, and recursive abstractions. In fact, all plain definitions in Coq have arity 0. Not having function definitions with arity $n \geq 1$ results in more fine-grained reduction and introduces intermediate normal forms one doesn't want to see from the application perspective. To mitigate the problem, Coq refines the basic reduction rules with *simplification rules* simulating the reductions one would have with inductive function definitions. Sometimes the simulation is not perfect and the user is confronted with unpleasant intermediate terms.

Coq comes with syntactic sugar facilitating the translation of inductive and plain function definitions into Coq's kernel language.

In this text we work with inductive function definitions and do 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.

To express recursive inductive functions, Coq has recursive abstractions. **Recursive abstractions** take the form

$$\textsc{fix}\ f^{s\to t}\ x^s.\ u$$

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 $s \to t$.

We restrict our discussion to simply typed recursive abstractions. To gain full expressivity, recursive abstractions in Coq are dependently typed.

Using an inductive function definition, a function D doubling its argument can be defined as follows:

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

To express this definition in Coq, we use a plain definition with a recursive abstraction and a match:

$$D^{\mathsf{N}\to\mathsf{N}} \ := \ \textsc{fix}\ f^{\mathsf{N}\to\mathsf{N}}\, x^{\mathsf{N}}.\ \textsc{match}\ x\ [\,0 \Rightarrow 0 \mid \mathsf{S}x \Rightarrow \mathsf{S}(\mathsf{S}(fx))\,]$$

The reduction rule for recursive abstractions looks as follows:

$$(\textsc{fix}\, f x. s)\, t \ \succ \ (\lambda f.\lambda x.\, s)\,(\textsc{fix}\, f x. s)\, t$$

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

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

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

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

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

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

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

Exercise 2.11.1 (Boolean negation) Consider the inductive type definition

$$\mathsf{B} : \mathbb{T} \ ::= \ \mathsf{true} \mid \mathsf{false}$$

for booleans and the plain definition

$$! \ := \ \lambda x^{\mathsf{B}}.\ \textsc{match}\ x\ [\,\mathsf{true} \Rightarrow \mathsf{false} \mid \mathsf{false} \Rightarrow \mathsf{true}\,]$$

of a boolean negation function. Give a complete reduction chain for $!(!\,\mathsf{true})$.

Exercise 2.11.2 (Swap function for pairs)

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

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

Exercise 2.11.3 Verify every single reduction step in Figure 2.2 and convince yourself that there is no other reduction chain.

2.12 Summary

We have outlined a typed and terminating functional language where functions and types are first-class values 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 (canonicity).

A key feature of modern type theories are dependent function types generalizing simple function types. 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: Type checking does not involve reduction, and reduction does not involve type checking. Soon we will boost the expressivity of the system by extending it such that type checking operates modulo computational equality of types. Type checking modulo computational equality is needed so that the proof rules for equational rewriting and induction on numbers can be obtained within computational type theory.

The computational type theory we are considering in this text is based on dependent function types and inductive function definitions. In addition there are lambda abstractions and let expressions. The language of the proof assistant Coq we are using differs in that it replaces native function definitions with native matches and recursive abstractions

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.

2.13 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 will prove helpful.

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 [30] for a complementary presentation of computational type theory. The reader may also be interested in learning more about *lambda calculus* [4, 17], 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 (note the notational coincidence) and thus receive proofs as follows:

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

The types for conjunctions $s \wedge t$ and disjunctions $s \vee t$ will be obtained with inductive type constructors such that a proof of $s \wedge t$ consists of a proof of s and a proof of t, and a proof of $s \vee t$ consists of 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 proof terms is an incremental process that can be carried out in interaction with a proof assistant. On paper we will facilitate the construction of proof terms with proof tables.

3.1 Implication and Universal Quantification

We extend our type theory with a second universe of **propositional types**:

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

The second rule establishes $\mathbb{P}$ as a subuniverse $\mathbb{P} \subseteq \mathbb{T}$ of $\mathbb{T}$. The third rule populates $\mathbb{P}$ with all function types $\forall x^u.v$ whose target type v is a propositional type. Given the typing rules for $\mathbb{P}$, we can derive the typings

$$\vdash \; \forall X^{\mathbb{P}}.\, X \to X \;:\; \mathbb{P}$$
$$\vdash \; \lambda X^{\mathbb{P}}.\lambda x^X.\, x \;:\; \forall X^{\mathbb{P}}.\, X \to X$$

We can now write propositions with implications and universal quantifications as propositional types. For instance, the propositional type $\forall X^{\mathbb{P}}.\, X \to X$ represents a proposition saying that every proposition implies itself, and the term $\lambda X^{\mathbb{P}} x^X.x$ appears as proof of this proposition. From now on we reserve the word **proposition** for propositional types.

We will soon see inductive propositional types. Two obvious examples are the propositional variants of void and unit from § 2.3:

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

For propositions and proofs to be meaningful, there must be propositions that do not have a proof. In our setup this means that there must be empty propositional types. Examples for empty propositional types are $\bot$ and $\forall X^{\mathbb{P}}.X$.

Here are more examples for 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:

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

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 negations $\neg s$ can be expressed as implications $s \to \bot$. The official name for $\bot$ is **falsity**. We obtain falsity as the type constructor of an inductive type definition not declaring a value constructor:

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

Since $\bot$ has no value constructor, the canonicity property of computational type theory ensures that $\bot$ has no element.[1] We define an inductive function

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

discriminating on its second argument of type $\bot$. Since $\bot$ has no value constructor, no defining equation for $\mathsf{E}_\bot$ is required. 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 $s \to \bot$:

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

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

[1] 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

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 without assumptions in 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 value constructor:

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

We have now seen two inductive type definitions whose type constructors $\bot$ and $\top$ are declared as propositional types. We will call the value constructors for inductive propositions (e.g., $\top$) **proof constructors**.

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:

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

With the type constructors '$\wedge$' and '$\vee$' we can form conjunctions $s \wedge t$ and disjunctions $s \vee t$ from given propositions s and t. With the proof 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 proof constructors as implicit arguments, something we have seen before with the value constructor for pairs. Since the explicit arguments of the proof constructors for disjunctions determine only one of the two implicit arguments, the other implicit argument 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. 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.

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

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

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

$$\mathsf{M}_\vee : \forall XYZ^{\mathbb{P}}.\; X \vee Y \to (X \to Z) \to (Y \to Z) \to Z$$
$$\mathsf{M}_\vee\, XYZ\,(\mathsf{L}x)\,e_1e_2 \;:=\; e_1x$$
$$\mathsf{M}_\vee\, XYZ\,(\mathsf{R}y)\,e_1e_2 \;:=\; e_2y$$

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

Figure 3.2: Match functions for conjunctions and disjunctions

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

The variables X, Y, Z range over propositions.

Figure 3.3: Proofs for propositions involving conjunctions and disjunctions

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

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

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

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

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

nesting a second plain match.

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

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

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

Figure 3.4: Proofs with nested patterns

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

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

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.

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

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

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

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 Tables

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 tables we have introduced in Chapter 1 provide a convenient information structure for constructing proof terms.

Here is a proof table 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 table is written top-down beginning with the initial claim. It records the construction of the proof term

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

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

Recall that proof tables are have-want tables that record on the left what we have and on the right what we want. When we start, the proof table 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 table**.

The rightmost column of a proof table records the actions developing the table 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 tables 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 tables. 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 table:

	$\neg\neg X \to (X \to \neg X) \to \bot$	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 table 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 table 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 table 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 table 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: $\lambda Xpqfgx.\, \textsc{match}\ fx\ [\, \mathsf{C}\_h \Rightarrow h(gx)\,]$

Figure 3.8: Proof table 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:

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

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

Figure 3.10 gives a proof table 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 tables 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 table 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 $\forall X^{\mathbb{T}} \forall p^{X \to \mathbb{P}} \forall Z^{\mathbb{P}}.\ (\forall x. px) \to Z \to \forall x.\ px \land Z$.

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 $\forall P^{\mathbb{P}}.\ P \lor \neg P$. 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 $\forall P^{\mathbb{P}}.\ P \lor \neg P$ 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) $\mathsf{XM} \to \forall PQ^{\mathbb{P}}.\ \neg(P \land Q) \to \neg P \lor \neg Q$ 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 17.5.5 will tell you more.

3.9 Discussion

In this chapter we have seen that computational type theory can elegantly express propositions and their proofs. Implications are expressed as function types $s \to t$, and functions of type $s \to t$ are taken as proofs of the implication $s \to t$. More generally, universal quantifications $\forall x^u. t$ are expressed as dependent function types $\forall x^u. t$. One speaks of the propositions as types approach. In contrast to conventional mathematical reasoning, the propositions as types approach accommodates propositions and proofs as first-class values that can be passed around with functions.

We have expressed falsity $\bot$, conjunction $s \land t$, and disjunction $s \lor t$ as inductive propositions. We have seen that these propositions can be characterized without inductive types just using function types. Negation $\neg s$ is obtained as implication $s \to \bot$.

The logical reasoning coming with the propositions as types approach is known as intuitionistic reasoning. Intuitionistic reasoning is more basic than conventional mathematical reasoning in that it does not provide the law of excluded middle. Conventional mathematical reasoning is then recovered by having the law of excluded middle as an explicit assumption when needed. The law of excluded middle may be expressed with the proposition $\forall X^{\mathbb{P}}.\ X \vee \neg X$ quantifying over propositions. Having a dedicated universe $\mathbb{P} : \mathbb{T}$ for propositional types is essential so that assumptions such as excluded middle can be limited to propositional types and do not concern computational types such as B and N.

In later chapters, we will see that the propositions as types approach extends to equations and existential quantifications.

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

In the propositions as types approach, formal proofs are obtained as terms built with constants, variables, function application, and lambda abstraction. Constants appear as proof constructors for $s \wedge t$, and $s \vee t$ and with the inductive match functions for $\bot$, $s \wedge t$, and $s \vee t$. The construction of (proof) terms is best done in a type-driven top-down fashion recorded with a proof table. The main operation is function application, where the function applied yields the required propositional type and subgoals are issued for the arguments of the function. The resulting proof language is amazingly elegant and easily yields familiar proof patterns: making assumptions (lambda abstraction), destructuring of assumptions (application of a match function), applying implicational assumptions (function application).

For the beginner, this chapter is the place where you will get fluent with lambda abstractions, matches (i.e. applications of match functions), and dependent function types. We offer dozens of examples for exploration on paper and in interaction with a proof assistant. For proving on paper, we use proof tables recording incremental top-down 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 table.

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.

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.

We have already pointed out that we have a separate universe $\mathbb{P}$ of propositional types so that assumptions like the law of excluded middle do not affect computational types. If we were not interested in computational types, we could work with a single universe $\mathbb{T}$ and consider all types as propositions.

4 Conversion and Leibniz Equality

This chapter introduces a typing rule called conversion. The conversion rule strengthens basic computational type theory so that propositional equality, existential quantification, and induction lemmas can be defined. The conversion rule relaxes the type discipline so that the typing rules apply modulo computational equality of types.

We will define propositional equality $s = t$ following a scheme known as Leibniz equality. Three typed constants will suffice: One constant accommodating equations $s = t$ as propositions, one constant providing proofs of trivial equations $s = s$, and one constant providing for equational rewriting. As it turns out, the constants provide for equational reasoning by means of their types, the actual definitions of the constants are not needed.

The conversion rule of the type theory gives the constants for trivial equations and for rewriting the proof power required. Due to the conversion rule, propositional equality will subsume computational equality, and a single rewriting constant will capture all equational rewriting situations.

We extend the type theory with abstract constant definitions hiding the definition of constants except for the types of the constants. Abstract constants suffice for propositional equality and many other purposes, including propositional connectives and theorems.

While abstract constants don't strengthen the type theory as it comes to what can be defined and to what can be proven, they provide a means for enforcing abstraction layers. A system of abstract constants constitutes an interface between the uses and the definitions of the constants.

In this chapter we start proving theorems in type theory. Theorems will be represented as abstract constants in type theory. On paper, we will transition from formal proofs (proof terms) to informal proofs similar to informal proofs in mathematical presentations. Our informal proofs will be formulated such that they can be elaborated into formal proofs, and the scripts constructing the formal proofs in the proof assistant will appear in the accompanying Coq files.

There is much elegance and surprise in this chapter. There are plenty of new subjects, including typing modulo computational equality, propositional equality, abstract constants, and the representation of theorems. Take your time to fully understand the beautiful constructions.

4.1 Conversion Rule

Recall the typing rules from Chapter 2. 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. We use the notation $u \approx u'$ to say that two terms u and u' are computationally equal. For now two terms are computationally equal if they reduce to the same term (up to renaming of bound variables). Later, we will strengthen computational equality with a law for lambda abstractions called eta equivalence.

Note that the conversion rule has a premise $\vdash u : \mathbb{T}$, which ensures that the term u describes a type. Importantly, the conversion rule also applies to propositional types (since there is a typing rule typing types in $\mathbb{P}$ also as types in $\mathbb{T}$).

Adding the conversion rule to the basic typing rules preserves the key properties of computational type theory (§ 2.5). As before, there is a type checking algorithm deriving types for well-typed terms, where the type is now unique up to computational equality and minimality with respect to universes.

For a first example of the use of the conversion rule consider the proposition

$$\forall p^{\mathsf{N}\to\mathbb{P}}.\ p(4) \to p(2+2)$$

Without the conversion rule, the proposition has no proof. To see this, consider the term

$$\lambda p^{\mathsf{N}\to\mathbb{P}}.\ \lambda a^{p(4)}.\ a$$

which has the propositional type $\forall p^{\mathsf{N}\to\mathbb{P}}.\ p(4) \to p(4)$, which is equal to the given proposition up to computational equality. Using the conversion rule the term is a proof of the proposition. There is also the possibility to use the conversion rule early as shown in the proof table

	$\forall p^{\mathsf{N}\to\mathbb{P}}.\ p(4) \to p(2+2)$	intro	
$p : \mathsf{N} \to \mathbb{P}$			
$a : p(4)$	p(2+2)	conversion	$p(2+2) \approx p(4)$
	$p(4)$	a	

The proof table gives us exactly the proof term shown above. So we learn that a term leaves open where type checking uses the conversion rule. On the other hand, we can use proof tables to say where the conversion rule is used with type checking.

Negation and propositional equivalence as plain functions

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

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

The plain definitions provide us with the constants $\neg$ and $\longleftrightarrow$ for functions constructing negations and propositional equivalences. 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).

A proof term for the proposition $\neg X \to X \to \bot$ is $\lambda f^{\neg X}.\, f$ The conversion facilitating the type checking is $X \to \bot \approx \neg X$, or alternatively, $\neg X \to X \to \bot \approx \neg X \to \neg X$.

Type ascription and polymorphic identity function

Type ascription is the ability to annotate a term with the type we want it to have. Type ascriptions can be expressed with the polymorphic identity function:

$$\begin{aligned} \mathsf{id} &: \forall X^{\mathbb{T}}.\, X \to X \\ \mathsf{id}\, X\, x &:= x \end{aligned}$$

If we write $\mathsf{id}\, t\, s$, we force a conversion of the type of s to the type t. If the two types are not computationally equal, $\mathsf{id}\, t\, s$ will not type check. Note that the terms $\mathsf{id}\, t\, s$ and s are computationally equal.

Exercise 4.1.1 Prove $\forall p^{\mathsf{N} \to \mathbb{P}}.\, p(2) \to p(7-5) \vee p(3)$ with a proof table and give the constructed proof term.

Exercise 4.1.2 (Leibniz symmetry)
Prove $\forall X^{\mathbb{T}}\, \forall xy^X.\ (\forall p^{X \to \mathbb{P}}.\, px \to py) \to (\forall p^{X \to \mathbb{P}}.\, py \to px)$ with a proof table and give the constructed proof term. Hint: Instantiate the predicate p in the premise with $\lambda y.py \to px$ where p is the predicate from the conclusion.

4.2 Abstract Propositional Equality

With 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:

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

We will 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 types of the constants.

The constant $\mathsf{eq} : \forall X^{\mathbb{T}}.\ X \to X \to \mathbb{P}$ allows us to write equations as propositional types:

$$\begin{array}{lcl} s = t & \rightsquigarrow & \mathsf{eq}\,\_\,st \\ s \neq t & \rightsquigarrow & \neg\mathsf{eq}\,\_\,st \end{array}$$

The constants Q and R provide the basic proof rules for equations. For applications of Q and R the conversion rule of the type theory is essential.

Q provides computational equality

The constant $\mathsf{Q} : \forall X^{\mathbb{T}}\,\forall x^{X}.\ \mathsf{eq}\,X\,x\,x$ provides for proofs by computational equality. Obviously, Q can prove trivial equations $s = s$. Given the conversion rule, Q can also proof every equation $s = t$ where s and t are computationally equal. This is the case since computational equality is compatible with the term structure. For instance, $s \approx s'$ and $t \approx t'$ implies $(s = t) \approx (s' = t')$.

For instance, Q proves $2 + 3 = 9 - 4$ using the conversion rule. This is the case since $2 + 3 \approx 5$ and $9 - 4 \approx 5$ imply $(2 + 3 = 9 - 4) \approx (5 = 5)$. This justifies the proof term $\mathsf{Q}\,\mathsf{N}\,5$. Other possible proof terms for $2 + 3 = 9 - 4$ are $\mathsf{Q}\,\mathsf{N}\,(9 - 4)$ and $\mathsf{Q}\,\mathsf{N}\,(4 + 1)$.

R provides equational rewriting

The constant $\mathsf{R} : \forall X^{\mathbb{T}}\,\forall xy^{X}\,\forall p^{X \to \mathbb{P}}.\ \mathsf{eq}\,Xxy \to px \to py$ provides for equational rewriting. Given a proof of an equation $s = t$, we can replace 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.

The conversion rule is essential for working with the rewriting law. Suppose we want to prove $x = y \to y = z \to x = z$. Then we assume $e : x = y$ and prove $y = z \to x = z$. With the conversion rule we rewrite the claim to

$$(\lambda y.y = z \to x = z)y$$

Now rewriting with R and $e : x = y$ reduces the claim to $(\lambda y.y = z \to x = z)x$, which simplifies to the trivial implication $x = z \to x = z$ using the conversion rule. This reasoning is best represented with a proof table:

	$x = y \to y = z \to x = z$	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
	$x = z \to x = z$	$\lambda e.e$

The table constructs the proof term

$$\lambda e^{x=y}.\,\mathsf{R}_{(\lambda y.y=z\to x=z)}\,e\,(\lambda e^{x=z}.e)$$

While the uses of the conversion rule appear prominently in the proof table, they don't show up explicitly in the proof term. While the first conversion is forced by the rewriting predicate in the application of R, the second conversion will only appear if the context imposes the type $x = y \to y = z \to x = z$ for the proof term.

Target type functions

The type of the rewrite law is our first use of a *target type function*:

$$\mathsf{R} \;:\; \forall X^{\mathbb{T}}\,\forall xy^{X}\,\forall p^{X\to\mathbb{P}}.\;\mathsf{eq}\,Xxy \to px \to py$$

Here the predicate p taken as argument determines the target type py of the rewriting constant R. Target type functions work because of the conversion rule and lambda abstractions. We will see nonpropositional target type functions in Chapter 5.

Exercise 4.2.1 Give a proof term for the equation $!\mathsf{true} = \mathsf{false}$. Explain why the term is also a proof term for the equation $\mathsf{false} = !!\mathsf{false}$.

Exercise 4.2.2 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 4.2.3 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 4.2.4 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.

$\top \neq \bot$	propositional disjointness
$\mathsf{true} \neq \mathsf{false}$	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
$\forall X^{\mathbb{T}} x^X y^X.\ x = y \to y = x$	symmetry
$\forall X^{\mathbb{T}} x^X y^X z^X.\ x = y \to y = z \to x = z$	transitivity

Figure 4.1: Basic equational facts

4.3 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 4.1 shows a collection of basic equational facts, and Figure 4.2 gives proof tables 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 tables in Figure 4.2 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 tables in Chapter 1 are realized in the propositions as types approach.

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.

Constructor laws

If we look at the facts in Figure 4.2, we see that three of them

$\mathsf{true} \neq \mathsf{false}$	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

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

	$\top \neq \bot$	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

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

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

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

$x : \mathsf{N},\ y : \mathsf{N}$	$\mathsf{S}x = \mathsf{S}y \to x = y$	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
	$x = x$	$\mathsf{Q}x$

Fact: $\forall X^{\mathbb{T}} \forall xyz^{X}.\, x = y \to y = z \to x = z$
Proof term: $\lambda Xxyze.\, \mathsf{R}_{(\lambda y.\ y = z \to x = z)}\, e\, (\lambda e.e)$

$X : \mathbb{T},\ x : X,\ y : X,\ z : X,$	$x = y \to y = z \to x = z$	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
	$x = z \to x = z$	$\lambda e.e$

Figure 4.2: Proof terms for some equational facts

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. A proof assistant will automatically provide constructor laws for every inductive type definition it accepts.

Exercise 4.3.1 Study the two proof terms given for transitivity in detail using Coq. Give the proof table 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 4.3.2 Give proof tables and proof terms for the following propositions:

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

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

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

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

e) $(\forall X^{\mathbb{P}}. X \to X) \neq (\forall X^{\mathbb{P}}. X)$

Exercise 4.3.3 (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 4.3.4 (Leibniz characterization of equality)
Verify the following characterization of equality:

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

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

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

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

Note that each of the two equivalences suggests a possible definition of the constant eq. We will choose the first equivalence.

Exercise 4.3.5 (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 $\forall X^{\mathbb{T}}\, \forall xy^X\, \forall p^{X \to \mathbb{P}}.\ px \to \neg py \to x \neq y$ expressing this insight.

4.4 Definition of Leibniz Equality

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

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

$$\begin{aligned}
&\mathsf{Q} : \forall X^{\mathbb{T}} \forall x.\ \mathsf{eq}\,X\,x\,x\\
&\mathsf{Q}\,Xx \ := \ \lambda pa.a
\end{aligned}$$

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

The definitions are amazingly 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 4.3.4.

The above definition of propositional equality is known as **Leibniz equality** and appears 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 see the definition of inductive equality in §15.1 and study it carefully in Chapter 29. Except for Chapter 29 the concrete definition of propositional equality does not matter since everything we need can be obtained from the abstract equality constants eq, Q, R introduced in §4.2.

Arithmetic Comparisons as defined functions

Now that we have equality, we define arithmetic comparisons as plain functions:

$$\begin{aligned}
&\leq :\ \mathsf{N} \to \mathsf{N} \to \mathbb{P}\\
&x \leq y \ := \ (x - y = 0)
\end{aligned}$$

In addition, we will use three notational variants:

$$x \geq y \ := \ y \leq x \qquad\qquad x < y \ := \ \mathsf{S}x \leq y \qquad\qquad x > y \ := \ y < x$$

4.5 Abstract Constants and Theorems

We now extend the definitional facilities of our type theory with abstract constant definitions. An **abstract constant definition** is like a plain constant definition but comes with the proviso that the defining equation is not available for reduction.

Thus when we use an abstract constant we cannot make use of its definition. We may say that the definition of an abstract constant is hidden.

We may use abstract constants to represent the constants for equality, conjunction, disjunction, and falsity. For using these constructs the definitions of the constants are not needed, all we need are the types of the constants.

In the examples considered so far always a group of constants is defined and the formation constants can only be made abstract after the accompanying introduction and elimination constants have been defined.

A mathematical development is a carefully arranged collection of definitions and theorems that build on each other. Theorems are observations that have been verified. In type theory, we model theorems as abstract constants with propositional types. The use of an abstract constant for a theorem models the fact that theorems can be used without knowing their proofs. It suffices to know that there is a proof. Sometimes theorems have involved proofs appearing in the literature the user has never worked through.

Theorems come with several different names including facts, lemmas, and corollaries. The different names are a matter of presentation and do not have technical significance. In type theory all we have are abstract constants with propositional types. We often use the term **lemma** to refer to abstract constants with propositional types.

4.6 Theorems in this Text

All theorems stated as such in this text are theorems in type theory. They are given informal proofs in the text and formal proofs in the accompanying Coq files, where they appear as abstract constants with propositional types. The informal proofs we give in the text are designed such that they can be elaborated into the formal proofs appearing in the accompanying Coq files.

In this text, we mostly use the heading “Fact” to state theorems obtained in type theory. Here is a first example.

Fact 4.6.1 (Applicative closure)
$\forall XY^{\mathbb{T}}\ \forall f^{X \to Y}\ \forall xx'^{X}.\ x = x' \to fx = fx'.$

Proof Rewriting. ∎

Following the usual mathematical conventions we use a numbering scheme to name facts in this text. In addition we may give a fact a more telling name, such as “applicative closure” in the above example.

We will mostly give informal proofs for the theorems we state in this text. The informal proofs will carry enough information so that they can be elaborated into formal proofs (i.e., proof terms) using a proof assistant. Typically, we will delegate the elaboration to the accompanying Coq files. For the purpose of illustration, we give a complete proof table for the proof of Fact 4.6.1:

	$\forall XY^{\mathbb{T}}\,\forall f^{X\to Y}\,\forall xx'^{X}.\; x = x' \to fx = fx'$	intro
$X, Y : \mathbb{T}$		
$f : X \to Y$		
$x, x' : X$		
$e : x = x'$	$fx = fx'$	conversion
	$(\lambda x'.fx = fx')\,x'$	apply R_e
	$(\lambda x'.fx = fx')\,x$	conversion
	$fx = fx$	$\mathsf{Q}\,(fx)$

To demonstrate the use of Fact 4.6.1, we will prove the constructor laws for numbers using inductive functions for predecessors and not-zero tests:

$$\begin{aligned} &\mathsf{pred} : \mathsf{N} \to \mathsf{N} \\ &\mathsf{pred}\,0 := 0 \\ &\mathsf{pred}\,(\mathsf{S}x) := x \end{aligned} \qquad\qquad \begin{aligned} &\mathsf{notzero} : \mathsf{N} \to \mathbb{P} \\ &\mathsf{notzero}\,0 := \bot \\ &\mathsf{notzero}\,(\mathsf{S}\_) := \top \end{aligned}$$

Fact 4.6.2 (Injectivity of successor function) $\mathsf{S}x = \mathsf{S}y \to x = y$.

Proof We assume $\mathsf{S}x = \mathsf{S}y$ and prove $x = y$. By conversion it suffices to prove $\mathsf{pred}\,(\mathsf{S}x) = \mathsf{pred}\,(\mathsf{S}y)$, which follows by rewriting with the assumption. ■

Fact 4.6.3 (Disjointness of successor and zero) $\mathsf{S}x \neq 0$.

Proof We assume $\mathsf{S}x = 0$ and prove $\bot$.
By conversion it suffices to prove $\mathsf{notzero}\,0$. Follows by rewriting with the assumption. ■

Exercise 4.6.4 Give proof terms for Facts 4.6.1, 4.6.2, and 4.6.3.

Exercise 4.6.5 Prove $\forall XY^{\mathbb{T}}\,\forall fg^{X\to Y}\,\forall x^{X}.\; f = g \to fx = gx$. We shall often tacitly use this applicative closure law.

4.7 Abstract Presentation of Propositional Connectives

Similar to propositional equality, the propositional connectives falsity, conjunction, and disjunction can be accommodated with systems of abstract constants as shown in Figure 4.3. This phenomenon demonstrates a general abstractness property of logical reasoning. Among the constants in Figure 4.3, we distinguish between **constructors** and **eliminators**. The inductive definitions of falsity, conjunction, and

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

Figure 4.3: Abstract constants for falsity, conjunctions, and disjunctions

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 as match functions (Figure 3.2). If we look at the abstract constants for equality,

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

we can identify eq and Q as constructors and $\mathsf{E}_=$ as eliminator. We now use $\mathsf{E}_=$ for the equational constant R to avoid a conflict with the constant R for disjunction.

There is great beauty to the abstract presentation of the propositional connectives and equality 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, Q) provide the basic proof rules for the connectives.
- The **elimination constants** ($\mathsf{E}_\bot$, $\mathsf{E}_\wedge$, $\mathsf{E}_\vee$, $\mathsf{E}_=$) provide proof rules that for the proof of an arbitrary proposition 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.

There is another important fact about the abstract presentation of the logical connectives: In each case the impredicative characterization can be established using the abstract constants, and in each case the impredicative characterization can be read of the type of the elimination constants. Thus any definition of a formation constant satisfying the impredicative characterization will be equivalent to the abstract formation constant. Moreover, in each case, the impredicative characterization of the formation constant suggests a definition of the abstract constants.

One more observation. If we have the type of the formation constant, the type of the elimination constant follows from the types of the introduction constants by thinking of the given constants an as inductive definition. Conversely, given the types of the formation and elimination constants, the types of the introduction constants follow from the type of the elimination constant by reconstructing the connecting inductive definition.

We will see in § 6.1 that existential quantification can also be captured with a system of abstract constants following the formation-introduction-elimination scheme, and that the constants can be defined either inductively or impredicatively.

Exercise 4.7.1 (Impredicative characterizations) Assume the abstract constants for falsity, conjunction, and disjunction and in each case establish the impredicative characterization of the formation constant.

Exercise 4.7.2 (Impredicative definitions) Define the abstract constants for falsity, conjunction, and disjunction using the impredicative characterization of the formation constant. Do not use the inductive definitions.

Exercise 4.7.3 Prove commutativity of conjunction and disjunction using the abstract constants for conjunction and disjunction.

4.8 Computational Equality and Eta Equivalence

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, lambda abstractions $\lambda x^t.s$, and let expressions. 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 $\forall Y^{\mathbb{T}}.Y \to Y$.

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.

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 4.2 contains several proof tables with expansion steps.

Exercise 4.8.1 (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.

Exercise 4.8.2 Verify that the following equations hold by computational equality using the definitions of $+$, $-$, and iter from Chapter 1.

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

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

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

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

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

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

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

4.9 Notes

Computational equality is also known as *definitional equality* in type theory.

There are plenty of new ideas in this chapter playing a major role from now on:

- Type checking modulo computational equality of types
- Target type functions
- Abstract constants
- Representation of equality with abstract constants
- Representation of theorems with abstract constants
- Representation of propositional connectives with abstract constants
- Informal proofs rather than formal proofs in the text
- Elaboration of informal proofs in the text into formal proofs in the Coq files

Exercise 4.9.1 Figure 4.4 summarizes the equational laws we have discussed in this chapter. Make sure you can give for each law the complete quantifier prefix and a proof term using the abstract constants eq, Q, and R.

Equivalence

$$\begin{aligned}&x = x\\&x = y \to y = x\\&x = y \to y = z \to x = z\end{aligned}$$

Rewriting

$$\begin{aligned}&x = y \to px \to py\\&x = y \to py \to px\end{aligned}$$

Applicative closure

$$\begin{aligned}&x = y \to fx = fy\\&f = g \to fx = gx\end{aligned}$$

Impredicative characterization

$$x = y \longleftrightarrow \forall p.\, px \to py$$

Disequality

$$px \to \neg py \to x \neq y$$

Constructor laws

$$\begin{aligned}&\mathsf{true} \neq \mathsf{false}\\&0 \neq \mathsf{S}x\\&\mathsf{S}x = \mathsf{S}y \to x = y\\&(x, y) = (x', y') \to x = x' \wedge y = y'\end{aligned}$$

Figure 4.4: Summary of equational laws

5 Inductive Eliminators

For the inductive types we have seen we have defined match functions. Match functions are general inductive functions applying continuations given as arguments. The type of a match function may be formulated with a target type function, something we have seen before with the rewriting law. Target type functions raise the proof power of match functions substantially, as we will see with booleans, numbers, and pairs.

Following comon language, we call general inductive functions like match functions (inductive) eliminators. For every inductive type there is a canonical eliminator, which, in some sense, is most general. If the inductive type is recursive (like the type of numbers), the canonical eliminator will also take care of the recursion. Speaking from the perspective of proving, the types of eliminators describe proof rules for structural case analysis and structural induction. With the canonical eliminators for booleans, numbers, and pairs we can finally formalize all informal proofs we have seen in Chapter 1.

We will study the eliminators for booleans, numbers, and pairs using examples. This will include a formal proof of the disequation $\mathsf{N} \neq \mathsf{B}$. We will also look at the eliminators for void and unit.

It is time to spell out two important restrictions of the typing discipline. The restrictions are needed so that we obtain a consistent type theory that cannot prove falsity. The restrictions concern propositional discrimination and the self-containment $\mathbb{T} : \mathbb{T}$.

With this chapter we arrive at a computational type theory that can formalize a large variety of theorems, including all theorems in Chapter 1. The only extension of computational type theory still to come concerns generalized inductive types (Chapter 15).

5.1 Generalized Subtyping

The subtyping rule from § 3.1

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

makes it possible to see propositional types as types of the full universe $\mathbb{T}$. We now generalize the subtyping rule such that predicates can be seen as type functions:

$$\frac{\vdash s \;:\; \forall x_1\!:\!t_1.\;\cdots\;\forall x_n\!:\!t_n.\;\mathbb{P}}{\vdash s \;:\; \forall x_1\!:\!t_1.\;\cdots\;\forall x_n\!:\!t_n.\;\mathbb{T}}\quad n \geq 0$$

To have an example for the use of the subtyping rule, suppose we have a function

$$\mathsf{E}:\;\forall p^{\mathsf{B}\to\mathbb{T}}.\;p\,\mathsf{true} \to p\,\mathsf{false} \to \forall b.\,pb$$

Then both $\lambda p^{\mathsf{B}\to\mathbb{P}}.\,\mathsf{E}\,p$ and $\lambda p^{\mathsf{B}\to\mathbb{T}}.\,\mathsf{E}\,p$ are well-typed terms. A function E with the given type will appear as the boolean eliminator in §5.4, and the generalized subtyping rule frees us from defining E separately for predicates and type functions.

5.2 Propositional Discrimination Restriction

There is a prominent restriction on inductive functions discriminating on arguments with inductive propositional types. The restriction says that in such a case the target type of the function must be propositional. We speak of the **propositional discrimination restriction** (**PDR**). PDR is needed so that excluded middle $\forall X^{\mathbb{P}}.\,X \vee \neg X$ can be assumed consistently. Without the propositional discrimination restriction, we would have a proof of $(\forall X^{\mathbb{P}}.\,X \vee \neg X) \to \bot$ (see Chapter 31).

There are two exceptions to the propositional discrimination restriction. The first exception says that discrimination on inductive propositions with no proof constructor is admissible. Thus the definition of an inductive function

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

generalizing the function $\mathsf{E}_\bot : \forall Z^{\mathbb{P}}.\;\bot \to Z$ from §3.2 is admissible. We speak of **computational falsity elimination** when we apply $\mathsf{E}_\bot$ to a nonpropositional target type. It will turn out that computational falsity elimination is essential for the definition of many functions. Examples will appear in §11.1, §11.2, and §19.1.

The second exception to the propositional discrimination restriction concerns inductive propositions with a single proof constructor where all nonparametric arguments of the proof constructor have propositional types. This exception applies to the inductive proposition

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

and provides for the definition of the inductive function

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

We will refer to $\mathsf{E}_\bot$ and $\mathsf{E}_\top$ as the **eliminators** for the inductive propositions $\bot$ and $\top$. The eliminator $\mathsf{E}_\bot$ makes it possible to obtain a value for every type in a propositionally contradictory situation. The eliminator $\mathsf{E}_\top$ makes it possible to prove that I is the only value of $\top$.

The inductive types $\mathbb{0}$ (void) and $\mathbb{1}$ (unit) defined in § 2.3 correspond to the inductive propositions $\bot$ (falsity) and $\top$ (truth). The propositional variants $\bot$ and $\top$ are more flexible than $\mathbb{0}$ and $\mathbb{1}$ since they can also be used as propositions. As a matter of style, we will prefer $\mathbb{1}$ over $\top$ in computational situations.

We call inductive predicates that are exempted from the discrimination restriction **computational predicates**. It will take some time until we see computational predicates that matter. You may check § 14.1 (linear search) and Chapter 30 (well-founded recursion). The inductive predicate $\wedge^{\mathbb{P}\to\mathbb{P}\to\mathbb{P}}$ providing conjunction also comes out as computational, but there are no worthwhile applications of this feature.

We call a discrimination on a value of a propositional inductive type a **propositional transfer discrimination** if the traget type of the discrimination is not propositional.[1] A propositional transfer discrimination is only admissible if its propositional inductive type is computational. The defining equations of the eliminators $\mathsf{E}_\bot$ and $\mathsf{E}_\mathbb{1}$ are examples for transfer discriminations.

A helpful intuition for the propositional discrimination restriction is that it ensures that no information is leaked from the propositional level to the computational level. For instance, given a proof of a disjunction, it cannot be leaked to the computational level which side of the disjunction is served by the proof.

Exercise 5.2.1 We can obtain an empty inductive proposition using a recursive proof constructor:

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

The propositional discrimination restriction does not apply to F since the argument of the single proof constructor C has a propositional type.

a) Define an inductive function $\mathsf{E} : \forall Z^{\mathbb{T}}.\ \mathsf{F} \to Z$.

b) Prove $\mathsf{F} \longleftrightarrow \bot$.

Exercise 5.2.2 Prove $\forall x^\top.\ x = \mathsf{I}$ in two ways:

1. By defining an inductive function of this type.
2. Using the eliminator $\mathsf{E}_\top$.

[1] In Coq slang transfer discriminations are called large eliminations.

5.3 Universe Levels

The typing rules involving the universe $\mathbb{T}$ need to be refined with universe levels in order that a consistent type theory is obtained. The problem is with the self-containment $\mathbb{T} : \mathbb{T}$, which permits *vicious cycles* if not restricted.[2] To fix the problem, every occurrence of the universe $\mathbb{T}$ is assigned a positive integer called a *universe level* during type checking. The typing rules containing more than one occurrence of $\mathbb{T}$ are augmented so that they exclude cycles $\mathbb{T}_i : \mathbb{T}_i$.

$$\frac{}{\vdash \mathbb{T}_i : \mathbb{T}_{i+1}} \qquad\qquad \frac{\vdash u : \mathbb{T}_i \qquad x : u \vdash v : \mathbb{T}_i}{\vdash \forall x{:}u.v \;:\; \mathbb{T}_i}$$

So we have $\mathbb{T}_i : \mathbb{T}_{i+1}$ but not $\mathbb{T}_i : \mathbb{T}_i$.

Higher universe levels can be forced with function types taking types as arguments. For instance, consider the function type

$$u \;:=\; \forall X{:}\mathbb{T}_i.\, t$$

where $t : \mathbb{T}_j$ but t is not propositional. Then u can only be typed with some $\mathbb{T}_k$ where $k > i$ and $k \geq j$. The situation completely changes if t is propositional. Then u can be typed with $\mathbb{P}$ no matter how large the level i is.

One may think of *graded universes* $\mathbb{T}_i$ as an infinite cumulative hierarchy of universes:

$$\begin{array}{c} \mathbb{P} \subseteq \mathbb{T}_1 \subseteq \mathbb{T}_2 \subseteq \mathbb{T}_3 \subseteq \cdots \\ \mathbb{P} \;:\; \mathbb{T}_1 \;:\; \mathbb{T}_2 \;:\; \mathbb{T}_3 \;:\; \cdots \end{array}$$

The lowest universe $\mathbb{P}$ is distinguished from the higher universes in that it is closed under function types taking types as arguments. Speaking propositionally, propositions are closed under all quantifications, including *big quantifications* over universes. This feature of $\mathbb{P}$ is known as *impredicativity*. The impredicative characterizations we have seen for falsity, conjunction, disjunction, and equality exploit the impredicativity of $\mathbb{P}$.

In practice, there is no need to worry about universe levels since the proof assistant will verify that they can be assigned consistently. It requires agressive constructions to force a universe level inconsistency. In Chapter 31 we will present a construction where universe levels become crucial and ignoring them would lead to a proof of falsity.

Finally, there is a generalized subtyping rule for graded universes:

$$\frac{\vdash s \;:\; \forall x_1{:}t_1.\ \cdots\ \forall x_n{:}t_n.\ \mathbb{T}_i}{\vdash s \;:\; \forall x_1{:}t_1.\ \cdots\ \forall x_n{:}t_n.\ \mathbb{T}_{i+1}}\; n \geq 0$$

[2]A related problem appears in set theory, where the set of all sets must be excluded.

Exercise 5.3.1 Consider the following terms describing functions:

$$\begin{aligned} \lambda a^{\mathbb{T}_1}.\,0 &: \mathbb{T}_1 \to \mathsf{N} : \mathbb{T}_2 \\ \lambda a^{\mathbb{T}_1 \to \mathsf{N}}.\,0 &: (\mathbb{T}_1 \to \mathsf{N}) \to \mathsf{N} : \mathbb{T}_3 \\ \lambda a^{(\mathbb{T}_1 \to \mathsf{N}) \to \mathsf{N}}.\,0 &: ((\mathbb{T}_1 \to \mathsf{N}) \to \mathsf{N}) \to \mathsf{N} : \mathbb{T}_4 \end{aligned}$$

a) Check the consistency of the given universe level assignments.
b) Explain why the application $(\lambda a^{\mathbb{T}_1}.\,0)(\mathbb{T})$ has no consistent universe level assignment and hence cannot be typed.
c) Explain why the application $(\lambda a^{\mathbb{T}}.\,0)(\mathbb{P})$ has a consistent universe level assignment and hence can be typed.
d) Give the least types of the terms $\lambda a^{\mathbb{T}_1}.\top$ and $\lambda a^{\mathbb{T}_1 \to \mathsf{N}}.\top$.

5.4 Boolean Eliminator

Recall the definition of the inductive type of booleans from §1.1 :

$$\mathsf{B} : \mathbb{T} \ ::= \ \mathsf{true} \mid \mathsf{false}$$

An inductive function f discriminating on a boolean argument has two defining equations:

$$\begin{aligned} f\,\mathsf{true} &:= s_1 \\ f\,\mathsf{true} &:= s_2 \end{aligned}$$

We generalize the format by taking the terms s_1 and s_2 as arguments:

$$\begin{aligned} f\,e_1\,e_2\,\mathsf{true} &:= e_1 \\ f\,e_1\,e_2\,\mathsf{true} &:= e_2 \end{aligned}$$

A possible type for this format is

$$f : \forall Z^{\mathbb{T}}.\, Z \to Z \to \mathsf{B} \to Z$$

A more general type for this format uses a target type function:[3]

$$f : \forall p^{\mathsf{B} \to \mathbb{T}}.\, p\,\mathsf{true} \to p\,\mathsf{false} \to \forall x.px$$

It turns out that the general format with the target type function is necessary when we use f to do a boolean case analysis. We fix the definition

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

[3]Recall that target type functions appeared first with the rewriting law for propositional equality.

and call the function $\mathsf{E_B}$ the **boolean eliminator**. We refer to the arguments e_1 and e_2 of $\mathsf{E_B}$ as **continuations**.

The type of $\mathsf{E_B}$ says that we obtain a proof of $\forall x^{\mathsf{B}}.\, px$ if we provide proofs of $p\,\mathsf{true}$ and $p\,\mathsf{false}$. This provides us with a general means to do boolean case analysis. To do a boolean case analysis with $\mathsf{E_B}$, the defining equations of $\mathsf{E_B}$ are not needed.

Consider the defining equations of $\mathsf{E_B}$. They are well-typed since the patterns $\mathsf{E_B}\,pab\,\mathsf{true}$ and $\mathsf{E_B}\,pab\,\mathsf{false}$ on the left instantiate the target type to $p\,\mathsf{true}$ and $p\,\mathsf{false}$, which are the types of the variables e_1 and e_2.

Note that the eliminator $\mathsf{E_B}^{\forall p^{\mathsf{B}\to\mathbb{T}}.\; p\,\mathsf{true}\to p\,\mathsf{false}\to\forall x.px}$ generalizes the match function $\mathsf{M_B}^{\forall Z^{\mathbb{T}}.\;\mathsf{B}\to Z\to Z\to Z}$ from § 2.9. The generalization comes with the target type function p of $\mathsf{E_B}$, which provides more flexibility than the target type Z of $\mathsf{M_B}$.

Recall that in type theory a boolean conditional is an abbreviation for an application of an inductive function discriminating on B. From now on we shall use the boolean eliminator for this purpose:

$$\text{IF } s_1 \text{ THEN } s_2 \text{ ELSE } s_3 \quad \rightsquigarrow \quad \mathsf{E_B}\; t\; s_2\; s_3\; s_1$$

Note that the term t describing the target type function must be derived from the context of the boolean conditional.

Exercise 5.4.1 Define boolean negation and boolean conjunction with the boolean eliminator.

5.5 Example: Boolean Case Analysis

Informally, the proposition

$$\forall x^{\mathsf{B}}.\; x = \mathsf{true} \vee x = \mathsf{false}$$

can be proved with a boolean case analysis reducing it to the subgoals $\mathsf{false} = \mathsf{true} \vee \mathsf{false} = \mathsf{false}$ and $\mathsf{false} = \mathsf{true} \vee \mathsf{false} = \mathsf{false}$. Formally, we obtain a proof term for the proposition using the boolean eliminator:

$$\mathsf{E_B}\; (\lambda x.\; x = \mathsf{true} \vee x = \mathsf{false})\; \ulcorner \mathsf{true} = \mathsf{true} \vee \mathsf{true} = \mathsf{false} \urcorner\; \ulcorner \mathsf{false} = \mathsf{true} \vee \mathsf{false} = \mathsf{false} \urcorner$$

The types enclosed in the upper corners are placeholders for terms having the given types. We refer to the placeholders as subgoals. Note that the types of the subgoals are obtained with the conversion rule. We may now use the proof terms $\mathsf{L}(\mathsf{Q}\,\mathsf{true})$ and $\mathsf{R}(\mathsf{Q}\,\mathsf{false})$ for the subgoals and obtain the complete proof term

$$\mathsf{E_B}\; (\lambda x.\; x = \mathsf{true} \vee x = \mathsf{false})\; (\mathsf{L}(\mathsf{Q}\,\mathsf{true}))\; (\mathsf{R}(\mathsf{Q}\,\mathsf{false}))$$

	$\forall x.\ x = \mathsf{true} \lor x = \mathsf{false}$	conversion
	$\forall x.\ (\lambda x.\, x = \mathsf{true} \lor x = \mathsf{false})\, x$	apply $\mathsf{E_B}$
1	$(\lambda x.\, x = \mathsf{true} \lor x = \mathsf{false})\, \mathsf{true}$	conversion
	$\mathsf{true} = \mathsf{true} \lor \mathsf{true} = \mathsf{false}$	trivial
2	$(\lambda x.\, x = \mathsf{true} \lor x = \mathsf{false})\, \mathsf{false}$	conversion
	$\mathsf{false} = \mathsf{true} \lor \mathsf{false} = \mathsf{false}$	trivial

Proof term constructed: $\mathsf{E_B}\ (\lambda x.\ x = \mathsf{true} \lor x = \mathsf{false})\ (\mathsf{L}(\mathsf{Q\,true}))\ (\mathsf{R}(\mathsf{Q\,false}))$

Figure 5.1: Proof table for a boolean elimination

Figure 5.1 shows a proof table constructing the proof term. The table makes explicit the conversions handling the applications of the target type functions.

That all boolean case analysis can be obtained with a single eliminator crucially depends on the use of a target type function in the type of the eliminator. A simply typed boolean eliminator $\forall Z^{\mathbb{T}}.\ Z \to Z \to \mathsf{B} \to Z$ can for instance not express the boolean case analysis needed for $\forall x^{\mathsf{B}}.\ x = \mathsf{true} \lor x = \mathsf{false}$. And recall that target type functions only work with lambda abstractions and a conversion rule taking care of β-reductions.

Exercise 5.5.1 For each of the following propositions give a proof term applying the boolean eliminator.

a) $\forall p^{\mathsf{B}\to\mathbb{P}} \forall x.\ (x = \mathsf{true} \to p\mathsf{true}) \to (x = \mathsf{false} \to p\mathsf{false}) \to px.$

b) $x \,\&\, y = \mathsf{true} \longleftrightarrow x = \mathsf{true} \land y = \mathsf{true}.$

c) $x \mid y = \mathsf{false} \longleftrightarrow x = \mathsf{false} \land y = \mathsf{false}.$

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

5.6 Kaminski's Equation

Here is a somewhat challenging fact known as **Kaminski's equation**[4] that can be shown with boolean elimination:

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

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

$$\forall xyz.\ f\,\mathsf{true} = y \ \to\ f\,\mathsf{false} = z \ \to\ f(f(fx)) = fx$$

[4]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.

by boolean case analysis on x, y, and z. This yields 8 subgoals, all of which have straightforward equational proofs. Here is the subgoal for $x = \mathsf{false}$, $y = \mathsf{false}$, and $z = \mathsf{true}$:

$$f\,\mathsf{true} = \mathsf{false} \;\to\; f\,\mathsf{false} = \mathsf{true} = \;\;\to\; f(f(f\,\mathsf{false})) = f\,\mathsf{false}$$

Speaking informally, the proof of Kaminski's equation proceeds by cascaded discrimination on x, $f\mathsf{true}$, and $f\mathsf{false}$, where the equations recording the discriminations on the terms $f\mathsf{true}$, and $f\mathsf{false}$ 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.[5]

Exercise 5.6.1 (Boolean pigeonhole principle)

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

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

Exercise 5.6.2 (Boolean enumeration) Prove $\forall x^{\mathsf{B}}.\; x = \mathsf{true} \lor x = \mathsf{false}$ 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.

5.7 Eliminator for Numbers

Recall the definition of the inductive type of numbers from §1.2:

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

An inductive function f discriminating on a numeric argument has two defining equations:

$$\begin{aligned} f\,0 &:= s_1 \\ f\,(\mathsf{S}n) &:= s_2 \end{aligned}$$

We generalize the format by taking the terms s_1 and s_2 as arguments:

$$\begin{aligned} f\,e_1\,e_2\,0 &:= e_1 \\ f\,e_1\,e_2\,(\mathsf{S}n) &:= e_2\,n\,(fn) \end{aligned}$$

Note that the argument e_2 is a function taking the predecessor n and the result of the recursive application fn as arguments.[6]

[5]Coq supports the pattern with the eqn modifier of the destruct tactic.

[6]Since computation in type theory is always terminating, eager recursion cannot cause problems.

We now come to the typing of the eliminator function. To obtain enough flexibility, we shall employ a target type function $p^{\mathsf{N}\to\mathbb{T}}$. Given the equations for f, this design decision forces the type

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

We now fix the definition

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

of the **arithmetic eliminator $\mathsf{E_N}$**. We refer to the arguments e_1 and e_2 of $\mathsf{E_N}$ as **continuations**. We say that the arithmetic eliminator takes continuations for the *zero case* and the *successor case*.

Next we look at the type of $\mathsf{E_N}$ as a proof rule:

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

It says that we can obtain a proof of $\forall n.pn$ by supplying proofs for $p0$ and $\forall n.\ pn \to p(\mathsf{S}n)$. For the second proof obligation we have to supply a function that for every n yields a proof of $p(\mathsf{S}n)$ given a proof of pn. Thus $\mathsf{E_N}$ gives us a proof rule for structural **induction on numbers**. Note that the so-called **inductive hypothesis** appears as pn in the type $\forall n.\ pn \to p(\mathsf{S}n)$ of the continuation function for the successor case.

We have just seen one of the most elegant and novel aspects of computational type theory: Induction on numbers is obtained through (the type of) the arithmetic eliminator, which is obtained as a recursive inductive function on numbers.

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

We shall use the notations

$$\begin{aligned} \text{MATCH } s\ [\,0 \Rightarrow s_1 \mid \mathsf{S}x \Rightarrow s_2\,] \ &\rightsquigarrow\ \mathsf{E_N}\, t\, s_1\, (\lambda x\_.s_2)\, s \\ \text{IF } s \text{ THEN } s_1 \text{ ELSE } s_2 \ &\rightsquigarrow\ \mathsf{E_N}\, t\, s_1\, (\lambda\_\_.s_2)\, s \end{aligned}$$

for arithmetic case analysis. The terms t describing the target type function must be inferred from the context.

Exercise 5.7.1 (Match function for numbers)
A match function for numbers has the type

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

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

Proof term constructed:

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

Figure 5.2: Proof table for $x + 0 = x$

a) Explain how a match function for numbers provides for case analysis.

b) Define a match function for numbers as an inductive function.

c) Define a match function for numbers using the arithmetic eliminator $\mathsf{E_N}$.

5.8 A Formal Inductive Proof

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

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

We do the proof by induction on x, which amounts to an application of the eliminator $\mathsf{E_N}$:

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

The application yields 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 5.2 shows a proof table for a proof term completing the partial proof term.

We will see many inductive proofs in this text. We shall write inductive proofs in an informal style making sure that the informal proof can be easily elaborated into a formal proof with the proof assistant. A (detailed) informal proof for our example may look as follows.

Fact 5.8.1 $x + 0 = x$.

Proof By induction on x. If $x = 0$, the claim follows by computational equality. In the successor case we have the claim $\mathsf{S}x + 0 = \mathsf{S}x$ and the inductive hypothesis $x+0 = x$. By conversion is suffices to show $\mathsf{S}(x+0) = \mathsf{S}x$, which follows by rewriting with the inductive hypothesis. ∎

Exercise 5.8.2 Consider the following facts about numbers.

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

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

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

In each case an inductive proof is required. Use the proof assistant to obtain informal and formal proofs of the facts. Remark: The Coq file accompanying Chapter 1 contains many examples of inductive proofs.

Exercise 5.8.3 Prove the following equations stating the correctness of addition functions obtained with the arithmetic eliminator. Both equations require inductive proofs.

a) $x + y = \mathsf{E_N}\,(\lambda\_.\mathsf{N})\,y\,(\lambda\_.\mathsf{S})\,x$

b) $x + y = \mathsf{E_N}\,(\lambda\_.\mathsf{N} \to \mathsf{N})\,(\lambda y.y)\,(\lambda\_ay.\mathsf{S}(ay))\,x\,y$

5.9 Equality of Numbers is Logically Decidable

We now show that equality of numbers is logically decidable.

Fact 5.9.1 $\forall x^{\mathsf{N}}\, y^{\mathsf{N}}.\ x = y \vee x \neq y$.

Proof By induction on x with y quantified followed by case analysis on y.

1. $x = 0$ and $y = 0$. Then $x = y$.
2. $x = 0$ and $y = \mathsf{S}\_$. Then $x \neq y$ by constructor law.
3. $x = \mathsf{S}\_$ and $y = 0$. Then $x \neq y$ by constructor law.
4. $x = \mathsf{S}x'$ and $y = \mathsf{S}y'$. The induction hypothesis gives us two cases:
   a) $x' = y'$. Then $x = y$.
   b) $x' \neq y'$. We assume $\mathsf{S}x' = \mathsf{S}y'$ and prove $\bot$. Injectivity of S gives us $x' = y'$, which contradicts the assumption $x' = y'$. ∎

The informal proof is burdened with many cases and much detail. Constructing a formal proof with a proof assistant will organize the cases and the details in a pleasant and more manageable way.

		$\forall x^{\mathsf{N}} y^{\mathsf{N}}.\ x = y \vee x \neq y$	apply $\mathsf{E_N}$, intro y
1		$0 = y \vee 0 \neq y$	destruct y
1.1		$0 = 0 \vee 0 \neq 0$	trivial
1.2		$0 = \mathsf{S}y \vee 0 \neq \mathsf{S}y$	constructor law
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		$\mathsf{S}x = 0 \vee \mathsf{S}x \neq 0$	constructor law
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 S

Figure 5.3: Proof table with a quantified inductive hypothesis

If you are not experienced with inductive proofs, the remark that y be quantified in the inductive hypothesis may be confusing. The confusion will go away with the formal proof.

A proof table constructing a proof term following the informal proof appears in Figure 5.3.

Following the table, we begin the construction of the formal proof with the partial proof term

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

This first step makes explicit the quantified inductive hypothesis. Both subgoals are shown by case analyis on the quantfied target. This can be done with the eliminator $\mathsf{E_N}$. To keep things manageable we will work with a match function

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

omitting the inductive hypothesis.

In the zero case $\ulcorner \forall y.\ 0 = y \vee 0 \neq y \urcorner$ we proceed with

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

The first subgoal is trivial, and the second subgoal follows with constructor disjointness.

In the successor case $\ulcorner\forall x.\,(\forall y.\, x = y \lor x \neq y) \to \forall y.\, \mathsf{S}x = y \lor \mathsf{S}x \neq y\urcorner$ we proceed with

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

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

This completes our explanation of a (formal) proof of Fact 5.9.1. 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 the assistant. To document the proof informally for a human reader, it's probably best to 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 5.9.2 (Boolean equality decider for numbers)
Write a function $\mathsf{eqb} : \mathsf{N} \to \mathsf{N} \to \mathsf{B}$ such that $\forall xy.\ x = y \longleftrightarrow \mathsf{eqb}\,xy = \mathsf{true}$. Prove the correctness of your function.

Exercise 5.9.3 (Antisymmetry)
Prove $x \leq y \to y \leq x \to x = y$.
Hint: Induction on x with y quantified using the constructor laws.

5.10 Eliminator for Pairs

Recall the inductive type definition for pairs from § 1.8 :

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

As before we use use the notations

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

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

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

We shall use the notation

$$\text{LET } (x_1, x_2) = s \text{ IN } s_1 \quad \rightsquigarrow \quad \mathsf{E}_\times\, t_1\, t_2\, t_3\, (\lambda xy.s_1)\, s$$

for cartesian destructuring. The terms t_1, t_2, and t_1 describing the component types and the target type function must be inferred from the context.

Exercise 5.10.1 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

See § 1.8 for the definitions of the projections π_1 and π_2 and the function swap.

Exercise 5.10.2 Use $\mathsf{E}_\times$ to write functions that agree with π_1, π_2, and swap.

Exercise 5.10.3 By now you know enough to formalize all proofs of Chapter 1 in computational type theory. Do some of the proofs in Coq without using the tactics for destructuring and induction. Apply the eliminators you have seen in this chapter instead.

5.11 Disequality of Types

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 how can we show this fact in the logical system we have arrived at?

Since B and N both have type $\mathbb{T}$, 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 4.3.5). We choose the predicate

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

saying that a type has at most two elements. It 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

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

by disjunctive case analysis and the constructor laws for 0 and S.

Fact 5.11.1 $\mathsf{N} \neq \mathsf{B}$.

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

Exercise 5.11.3 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 $\mathsf{B} = \mathsf{B} \times \top$.

5.12 Notes

We may accommodate the eliminators in this chapter as abstract constants (§4.5) hiding their defining equations. The abstract eliminators will be fine when we use them as proof rules. However, there are two uses of the eliminators where the reductions provided by the defining equations matter:

- Reducible matches contributing to type checking through the conversion rule. See the proofs of the constructor laws for B and N (§4.3).
- Local definitions of reducible inductive functions contributing to type checking. No examples yet.

We remark that the eliminators for conjunction and disjunction

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

used in Chapter 3 (Figure 3.2) don't use target type functions. In fact, there is no need for target type functions since in ordinary mathematical reasoning propositions don't talk about their proofs.

Functions typed with target type functions are polymorphic in the number of their arguments. For instance:

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

We remark that the proof assistant Coq automatically derives eliminators for every inductive type definition it processes. For the inductive types discussed in this chapter Coq derives the eliminators we have presented (except for $\top$).

6 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).

6.1 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

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

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

Figure 6.1 shows a proof table and the constructed proof term for a de Morgan law for existential quantification. The proof table 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 6.1. Instead, it is preferable to think modulo conversion. Figure 6.2 shows a proof table with implicit conversions constructing the same proof term. This is certainly a better presentation of the proof. The second table 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 6.1 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 6.1: 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 6.2: Proof of existential de Morgan law with implicit conversions

Exercise 6.1.1 Prove the following propositions with proof tables 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 6.1.2 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 6.1.3 Verify the following existential characterization of disequality:

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

Exercise 6.1.4 Verify the impredicative characterization of existential quantification:

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

Exercise 6.1.5 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 6.1.6 (Abstract presentation) We have seen that conjunction, disjunction, and propositional equality can be modeled with abstract constants (§4.7). For existential quantification, we may use the constants

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

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 6.1.7 (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.

6.2 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 6.2.1 (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 6.2.2 Give a proof table and a proof term for the barber theorem. Construct a detailed proof with Coq.

Exercise 6.2.3 Consider the following predicate on types:

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

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

6.3 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 6.3.1 Boolean negation has no fixed point.

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

Fact 6.3.2 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 6.3.3 (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 6.3.4 There is no surjective function $X \to (X \to \mathsf{B})$.

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

Corollary 6.3.5 There is no surjective function $X \to (X \to \mathbb{P})$.

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

We remark that Corollaries 6.3.4 and 6.3.5 may be seen as variants of Cantor's theorem.

Exercise 6.3.6 Construct with Coq detailed proofs of the results in this section.

Exercise 6.3.7

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 6.3.8 With Lawvere's theorem we can give another proof of Fact 6.3.2 (propositional negation has no fixed point). In contrast to the proof given with Fact 6.3.2, 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.

7 Arithmetic Pairing

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

7.1 Definitions

We will construct and verify two functions

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

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

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

Assuming a function

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

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

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

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

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

y	⋮						
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	⋯
	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 will prove $E(Dn)) = n$ and $D(Ea) = a$. The first equation is easier than the second equation. Both directions will profit from an equation saying that the encoding of the successor of a pair is the successor of the encoding of the pair.

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

Proof Case analysis on $a = (0, y)$, $(\mathsf{S}x, y)$ 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. ∎

We now come to the proof of the second equation $D(Ea) = a$. The difficulty here is that there is no n we can do induction on. We solve the problem by doing induction on the number Ea using an auxiliary variable $n = Ea$.[1] We prepare the necessary case analysis for the induction with three conditional equations for the encoding function. Two of the equations tell us how we can obtain for $Ea = \mathsf{S}n$ a pair a' such that $\eta(Ea') = a$. Note that a' is clear from the geometric presentation of the counting scheme.

Fact 7.2.3 (Backwards equations)

1. $Ea = 0 \to a = (0,0)$
2. $E(\mathsf{S}x, 0) = \mathsf{S}n \to E(0,x) = n$
3. $E(x, \mathsf{S}y) = \mathsf{S}n \to E(\mathsf{S}x, y) = n$

Proof (1) Follows with $a = (x,y)$ and case analysis on x and y using $\mathsf{S}\,\_ \neq 0$. (2) follows with $(\mathsf{S}x, 0) = \eta\,(0,x)$ and Fact 7.2.1. (3) follows with $(x, \mathsf{S}y) = \eta\,(\mathsf{S}x, y)$ and Fact 7.2.1. ∎

Fact 7.2.4 $D(Ea) = a$ for all pairs a.

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 Ea. Formally, we prove the proposition

$$\forall n\, \forall a.\ Ea = n \to Dn = a$$

by induction on n.

For $n = 0$, Fact 7.2.3 gives us $a = (0,0)$ making the conclusion trivial.

For the successor case we prove

$$Ea = \mathsf{S}n \to D(\mathsf{S}n) = a$$

We consider three cases: $a = (0,0)$, $(\mathsf{S}x, 0)$, $(x, \mathsf{S}y)$. The case $a = (0,0)$ is trivial since the premise is contradictory. The other two cases follow with (2) and (3) of Fact 7.2.3 and the inductive hypothesis. ∎

Exercise 7.2.5 (Walking diagonals the other way) The encoding function for pairs walks diagonals from right to left. One may also walk the diagonals from left to right. Rework the Coq development accordingly.

The exercise will help you with your understanding of arithmetic pairing.

The exercise will boost your understanding of the role a proof assistant can play in a mathematical development. Modifying a development with a proof assistant is more efficient and more reliable than doing it on paper. Type checking and proof checking turn out to be very helpful for this task.

[1] We have seen a similar unfolding step with Kaminski's equation in §5.6.

Exercise 7.2.6 (Bijections) A *bijection* between two types X and Y consists of two functions $f^{X \to Y}$ and $g^{Y \to X}$ such that $\forall x.\ g(fx) = x$ and $\forall y.\ f(gy) = y$.

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

b) Prove that there is no bijection between B and $\mathbb{1}$.

c) Prove that there is no bijection between N and $\mathsf{N} \to \mathsf{B}$.

Exercise 7.2.7 (Internal Arithmetic pairing)
Arithmetic pairing establishes N as a universal type that can represent pairs internally. To make this insight explicit, define functions

$$\pi : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \qquad \pi_1 : \mathsf{N} \to \mathsf{N} \qquad \pi_2 : \mathsf{N} \to \mathsf{N}$$

such that you can verify the equations

$$\pi_1(\pi n k) = n \qquad \pi_2(\pi n k) = k \qquad \pi(\pi_1 n)(\pi_2 n) = n$$

for all n and k.

7.3 Discussion

Arithmetic pairing is a great case study at several levels. The magic starts with the geometrical presentation of the counting scheme, which informally establishes a computable bijection. All the rest is about formal verification of what we see geometrically. The first step is the recursive definition of the encoding and decoding function, which is a nice programming exercise in computational type theory. As it comes to the verification of the two roundtrip equations, it is clear that both require induction on numbers. For one of the equations, the number we can do induction on needs to be explicated. Both inductive proofs hinge on the fact that a successor function η for pairs is made explicit that commutes with the successor function on numbers.

The elaboration of the proof in the accompanying Coq file is enlightening. We use it to demo several advanced features of the tactic language. As with most proofs in the text, the informal proofs in this chapter were written only after formal proofs had been constructed and polished with the proof assistant.

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. One typically starts with the encoding function using 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. The construction presented here is due to Andrej Dudenhefner (March 2020) who contributed it to Coq's standard library.

What I like about the development of the pairing function is the interplay between the geometric presentation of the counting scheme and the formalization with function definitions and proofs. Their is much elegance at all levels. Cantor's pairing function could make a great example for an educated Programming 1 course addressing functional programming and program verification.

8 Abstract Syntax

Inductive types provide for a tree-structured representation of syntactic objects. One speaks of abstract syntax if syntactic objects are represented as tree-structured objects, and of concrete syntax if syntactic objects are represented as character strings.

As example we consider arithmetic expressions and 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, code execution, 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 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 equation.

This chapter is also our first encounter with lists. Lists are obtained with a recursive inductive type definition refining the definition of numbers with elements. To be self-contained, we give a quick introduction to lists.

An important point of the chapter is in explaining that the principles behind the inductive definitions of numbers and pairs also cover lists and expressions. The constructor laws and the eliminators for lists and expressions can be obtained with the same scheme we have practiced with numbers and pairs.

8.1 Lists

Finite sequences $[x_1, \dots, x_n]$ may be represented as tree-structured values using 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 representing the sequence $[x_1, \dots, x_n]$ a list representing the sequence $[x, x_1, \dots, x_n]$. Given a list cons x A, we call x the **head** and A the **tail** of the list. We say that lists provide a nested pair representation of sequences.

Following the above design in type theory, we obtain lists with an inductive type definition

$$\mathcal{L}(X : \mathbb{T}) : \mathbb{T} \ ::= \ \mathsf{nil} \mid \mathsf{cons}\,(X, \mathcal{L}(X))$$

The type constructor $\mathcal{L} : \mathbb{T} \to \mathbb{T}$ gives us a **list type** $\mathcal{L}(X)$ for every **base type** X. The value constructor $\mathsf{nil} : \forall X^{\mathbb{T}}.\ \mathcal{L}(X)$ gives us an **empty list** for every base type. Finally, the value constructor $\mathsf{cons} : \forall X^{\mathbb{T}}.\ X \to \mathcal{L}(X) \to \mathcal{L}(X)$ provides for the construction of nonempty lists by adding an element in front of a given list. The type of cons ensures that all elements of a list of type $\mathcal{L}(X)$ are of type X.

For nil and cons, we don't write the first argument X. We use the notations

$$\begin{aligned} [] &:= \mathsf{nil} \\ x :: A &:= \mathsf{cons}\, x\, A \end{aligned}$$

and omit parentheses as follows:

$$x :: y :: A \quad \rightsquigarrow \quad x :: (y :: A)$$

When convenient, we shall use the sequence notation $[x_1, \dots, x_n]$ for lists.

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 may see the constructors nil and cons as refinements of the constructors 0 and S.

Concatenation of sequences

$$[x_1, \dots, x_m] +\!\!+ [y_1, \dots, y_n] \ = \ [x_1, \dots, x_m, y_1, \dots, y_n]$$

appends two sequences. You may be familiar with concatenation of strings, which are sequences of characters.

We provide **concatenation of lists** with an inductive function

$$\begin{aligned} +\!\!+ :\ & \forall X^{\mathbb{T}}.\ \mathcal{L}(X) \to \mathcal{L}(X) \to \mathcal{L}(X) \\ [] +\!\!+ B &:= B \\ (x :: A) +\!\!+ B &:= x :: (A +\!\!+ B) \end{aligned}$$

Concatenation of lists is similar to addition of numbers. Two basic laws for addition are $x + 0$ and $(x + y) + z = x + (y + z)$. We prove the list versions $A +\!\!+ [] = A$ and $A +\!\!+ (B +\!\!+ C) = (A +\!\!+ B) +\!\!+ C$ of the laws. The proofs are very similar to the arithmetic proofs with induction on lists replacing induction on numbers.

Fact 8.1.1 (Concatenation with nil) $A +\!\!+ [\,] = A$.

Proof By induction on A, which yields the proof obligations

$$[\,] +\!\!+ [\,] = [\,]$$
$$(x :: A) +\!\!+ [\,] = x :: A$$

The first obligations holds by computational equality. The second obligation simplifies to $x :: (A +\!\!+ [\,]) = x :: A$ and follows with the induction hypothesis. ■

Fact 8.1.2 (Associativity) $A +\!\!+ (B +\!\!+ C) = (A +\!\!+ B) +\!\!+ C$.

Proof By induction on A. Check the details with the proof assistant. ■

As with numbers, induction on lists is formalized with an eliminator function

$$\mathsf{E}_{\mathcal{L}} : \ \forall X^{\mathbb{T}}\, \forall p^{\mathcal{L}(X)\to\mathbb{T}}.\ p\,[\,] \to (\forall x A.\ pA \to p(x :: A)) \to \forall A.\, pA$$

which is obtained with a scheme generalizing the scheme we have seen for numbers. Proof assistants will generate a suitable eliminator function when they accept an inductive type definition.

Exercise 8.1.3 (Eliminator for lists) Define the eliminator function for lists with the type stated above. Verify that the inductive proofs of Facts 8.1.2 and 8.1.1 can be formalized with the inductive eliminator function.

Exercise 8.1.4 (Length) Define a length function $\mathsf{len} : \forall X.\ \mathcal{L}(X) \to \mathsf{N}$ for lists and prove $\mathsf{len}\,(A +\!\!+ B) = \mathsf{len}\,A + \mathsf{len}\,B$. Note that the length function prunes a list into the number that remains if one cuts away the elements coming with cons.

Exercise 8.1.5 (Reversal) Define a function $\mathsf{rev} : \forall X.\ \mathcal{L}(X) \to \mathcal{L}(X)$ reversing lists and prove $\mathsf{rev}\,(A +\!\!+ B) = \mathsf{rev}\,B +\!\!+ \mathsf{rev}\,A$ and $\mathsf{rev}\,(\mathsf{rev}\,A) = A$. For instance, we have $\mathsf{rev}\,[1,2,3,4] = [4,3,2,1]$.

8.2 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 a BNF grammar:

$$e : \mathsf{exp} \ ::= \ x \mid e_1 + e_2 \mid e_1 - e_2 \qquad (x : \mathsf{N})$$

The grammar models expressions as tree-structured objects. Following the grammar, we represent **expressions** with an inductive type

$$\mathsf{exp} : \mathbb{T} \ ::= \ \mathsf{con}(\mathsf{N}) \mid \mathsf{add}(\mathsf{exp}, \mathsf{exp}) \mid \mathsf{sub}(\mathsf{exp}, \mathsf{exp})$$

Note that the inductive type exp has one value constructor for every form of expression identified by the grammar.

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 value of an expression:

$$\begin{aligned} &E : \mathsf{exp} \to \mathsf{N} \\ E\,x \ &:= \ x \\ E\,(e_1 + e_2) \ &:= \ E\,e_1 + E\,e_2 \\ E\,(e_1 - e_2) \ &:= \ E\,e_1 - E\,e_2 \end{aligned}$$

Note that E is defined with binary structural recursion. Moreover, E is executable. For instance, $E(3 + 5 - 2)$ reduces to 6, and the equation $E(3 + 5 - 2) = E(2 + 3 + 1)$ holds by computational equality.

Exercise 8.2.1 Do the reduction $E(3 + 5 - 2) \succ^* 6$ step by step (at the equational level).

Exercise 8.2.2 (Constructor Laws for expressions) Prove some of the constructor laws for expressions. For instance, show that con is injective and that add and sub are disjoint.

Exercise 8.2.3 (Eliminator for expressions) 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. Following this design, the type of the eliminator is

$$\begin{aligned} &\forall p^{\mathsf{exp} \to \mathbb{T}}. \\ &(\forall x.\ p(\mathsf{con}\, x) \to \\ &(\forall e_1 e_2.\ pe_1 \to pe_2 \to p(\mathsf{add}\, e_1 e_2) \to \\ &(\forall e_1 e_2.\ pe_1 \to pe_2 \to p(\mathsf{sub}\, e_1 e_2) \to \\ &\forall e.pe \end{aligned}$$

8.3 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:

$$R : \mathcal{L}(\mathsf{N}) \to \mathcal{L}(\mathsf{N}) \to \mathcal{L}(\mathsf{N})$$

$$\begin{aligned}
R\ []\ A &:= A \\
R\ (0 :: C)\ (x_1 :: x_2 :: A) &:= R\ C\ (x_1 + x_2 :: A) \\
R\ (1 :: C)\ (x_1 :: x_2 :: A) &:= R\ C\ (x_1 - x_2 :: A) \\
R\ (\mathsf{SS}x :: C)\ A &:= R\ C\ (x :: A) \\
R\ \_\ \_ &:= []
\end{aligned}$$

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 8.3.1 Do the reduction $R[5,7,1][] \succ^* [2]$ step by step (doing operations on numbers in one step).

8.4 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 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 8.4.1 (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 of concatenation (Fact 8.1.2). The details can be explored with the proof assistant. ■

Corollary 8.4.2 $R\ (\gamma e)\ [] = [Ee]$.

Proof Theorem 8.4.1 with $C = A = []$ and Fact 8.1.1. ■

Exercise 8.4.3 Do the reduction $\gamma(5 - 2) \succ^* [4, 7, 1]$ step by step (at the equational level).

Exercise 8.4.4 Explore the proof of the correctness theorem starting with the proof script in the accompanying Coq development.

8.5 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 maintained:

$$\delta : \mathcal{L}(\mathsf{N}) \to \mathcal{L}(\mathsf{exp}) \to \mathcal{L}(\mathsf{exp})$$

$$\begin{aligned}
\delta\ []\ A &:= A \\
\delta\ (0 :: C)\ (e_1 :: e_2 :: A) &:= \delta\ C\ (e_1 + e_2 :: A) \\
\delta\ (1 :: C)\ (e_1 :: e_2 :: A) &:= \delta\ C\ (e_1 - e_2 :: A) \\
\delta\ (\mathsf{SS}x :: C)\ A &:= \delta\ C\ (x :: A) \\
\delta\ \_\ \_ &:= []
\end{aligned}$$

The correctness theorem for decompilation closely follows the correctness theorem for compilation.

Theorem 8.5.1 (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 8.5.2 $\delta\ (\gamma e)\ [] = [e]$.

8.6 Notes

The semantics of the expressions and codes considered here is particularly simple since evaluation of expressions and execution of codes can be accounted for by structural recursion.

Expressions are represented 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 realization of programming systems, it has no semantic relevance.

Early papers (late 1960's) on verifying compilation of expressions are McCarthy and Painter [25] and Burstall [6]. Burstall's paper is 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.

9 Certifying Functions and Sum Types

In computational type theory one can write function types that specify the input-output relation of their functions. We speak of certifying function types and of certifying functions. When proving properties of a certifying function, one ignores its definition and relies on its type. For this reason, we will accommodate certifying functions with abstract constants.

Certifying functions are at the heart of type theory and will play a main role from now on. As it turns out, language used for mathematical proofs generalizes to language for constructing certifying functions.

Certifying functions are the computational analogue of propositional lemmas. Like lemmas, they come as abstract constants whose definition does not matter for their use. Like lemmas, certifying functions are constructed with informal arguments providing for their formal construction in tactic mode. As with lemmas, the details of the formal definition of certifying functions do not matter.

The types of certifying functions are often obtained with sum types and sigma types, which are computational variants of propositional disjunctions and existential quantifications. To keep the amount of new ideas digestible, we postpone the discussion of sigma types to the next chapter.

One important application of sum type are certifying equality deciders. We refer to types having a certifying equality decider as discrete types. We will see that the class of discrete types is closed under taking product types and under taking sum types. Moreover, discrete types are closed under taking injective preimages.

9.1 Sum Types

Sum types are a basic type construction and it is about time we introduce them. Like product types $X \times Y$ sum types $X + Y$ combine two types X and Y. However, sum types are dual to product types in that their elements carry a value of one of the types rather than values of both types. Sum types may be seen as disjoint type unions. The propositional versions of sum types and product types are conjunctions and disjunctions.

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

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

The definition gives us 3 constructors:

$$\begin{aligned} + &: \ \mathbb{T} \to \mathbb{T} \to \mathbb{T} \\ \mathsf{L} &: \ \forall XY^{\mathbb{T}}.\, X \to X + Y \\ \mathsf{R} &: \ \forall XY^{\mathbb{T}}.\, Y \to X + Y \end{aligned}$$

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 discrimination restriction.

A simply typed eliminator for sum types has the type

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

We will see destructurings where a simply typed eliminator doesn't suffice. We define a dependently typed **eliminator for sum types** as follows:

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

$$\begin{aligned} \mathsf{E}_+\, XYpe_1e_2\,(\mathsf{L}x) &:= \ e_1x \\ \mathsf{E}_+\, XYpe_1e_2\,(\mathsf{R}y) &:= \ e_2y \end{aligned}$$

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

$\bot$	no element
$\bot + \mathbb{1}$	1 element
$(\bot + \mathbb{1}) + \mathbb{1}$	2 elements
$((\bot + \mathbb{1}) + \mathbb{1}) + \mathbb{1}$	3 elements

Exercise 9.1.1 Give all elements of the type $((\bot + \mathbb{1}) + \mathbb{1}) + \mathbb{1}$ and prove that your enumeration is complete: $\forall a^{((\bot+\mathbb{1})+\mathbb{1})+\mathbb{1}}.\ a = \mathsf{R\,I} \vee \cdots$.

Exercise 9.1.2 (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) $\mathsf{R}y = \mathsf{R}y' \to y = y'$.

Hint: The techniques used for numbers (Figure 4.2) also work for sums.

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

Exercise 9.1.3 Define a *truncation function* $T : \forall XY.\ X + Y \to \mathsf{B}$ such that $\forall a.\ \text{IF } Ta \text{ THEN } (\exists x.\, a = \mathsf{L}\,x) \text{ ELSE } (\exists y.\, a = \mathsf{R}\,y)$.

9.2 Proving at Type Level

It will be handy to have **equivalence types**:

$$\mathsf{X} \Leftrightarrow \mathsf{Y} \ := \ (X \to Y) \times (Y \to X)$$

A value of an equivalence type $X \Leftrightarrow Y$ is a pair of two functions translating between the types X and Y. For instance, the equivalence types

$$\begin{aligned}
(X \times Y \to Z) &\ \Leftrightarrow\ (X \to Y \to Z) \\
(X + Y \to Z) &\ \Leftrightarrow\ (X \to Z) \times (Y \to Z) \\
(X \to Y \times Z) &\ \Leftrightarrow\ (X \to Y) \times (X \to Z)
\end{aligned}$$

are inhabited for all types X, Y, Z.

Equivalence types $X \Leftrightarrow Y$ are similar to equivalence propositions $P \longleftrightarrow Q$. Equivalence types $X \Leftrightarrow Y$ are more general than equivalence propositions $P \longleftrightarrow Q$ in that they can take all types as arguments and not just propositions.

It is often advantageous to approach the construction of a value of a type X as a proof of X. For instance, the construction of a value of the type equivalence

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

is at a certain abstraction level identical with a proof of the propositional equivalence

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

The advantage of taking the proof view for types is that the construction of a value can be done at the proof level rather than at the term level. As we know from propositions, writing an informal proof that can be elaborated into a formal proof is much easier than writing a formal proof. Moreover, the elaboration of informal proofs into formal proofs is greatly assisted by the tactic interpreter of a proof assistant. It is now time to acknowledge that the tactic interpreter of Coq works for types in general, not just propositions. The only significant difference between the propositional and the general level is the propositional discrimination restriction, which imposes extra conditions on propositional destructuring.

Exercise 9.2.1 Prove the following type equivalences:

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

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

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

Exercise 9.2.2 Prove the following types:

a) $\forall b^{\mathsf{B}}.\ (b = \mathsf{true}) + (b = \mathsf{false})$.

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

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

The above types can all be shown by boolean case analysis. To see the dramatic reduction of formal detail obtained by using proof-oriented language construct the functions asked for at the term level.

Exercise 9.2.3 Prove that double negated disjunction agrees with double negated sum: $\neg\neg(P \vee Q) \longleftrightarrow \neg(P + Q \to \bot)$.

Exercise 9.2.4 (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.

Exercise 9.2.5 Construct 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 obtained because of the propositional discrimination restriction.

9.3 Decision Types

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

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

A value of type $\mathcal{D}(X)$ is a **decision** carrying either an element of X or a proof $X \to \bot$ verifying 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 $\neg X$.

If we have a decision $\mathcal{D}(X)$, we call X a **decided type**. It turns out that $\bot$ and $\mathbb{1}$ are decided types, and that decided types are closed under taking function types, product types and sum types. Moreover, decided types are closed under type equivalence. Finally, decided propositions are closed under the propositional connectives and propositional equivalence.

Fact 9.3.1 (Closure laws for decided types)

1. $\mathcal{D}(\mathbb{1})$ and $\mathcal{D}(\bot)$.
2. $\forall XY^{\mathbb{T}}.\ \mathcal{D}(X) \to \mathcal{D}(Y) \to \mathcal{D}(X \to Y)$.
3. $\forall X^{\mathbb{T}}.\ \mathcal{D}(X) \to \mathcal{D}(X \to \bot)$.
4. $\forall XY^{\mathbb{T}}.\ \mathcal{D}(X) \to \mathcal{D}(Y) \to \mathcal{D}(X \times Y)$.
5. $\forall XY^{\mathbb{T}}.\ \mathcal{D}(X) \to \mathcal{D}(Y) \to \mathcal{D}(X + Y)$.
6. $\forall XY^{\mathbb{T}}.\ (X \Leftrightarrow Y) \to \mathcal{D}(X) \to \mathcal{D}(Y)$.
7. $\forall XY^{\mathbb{P}}.\ \mathcal{D}(X) \to \mathcal{D}(Y) \to \mathcal{D}(X \land Y)$.
8. $\forall XY^{\mathbb{P}}.\ \mathcal{D}(X) \to \mathcal{D}(Y) \to \mathcal{D}(X \lor Y)$.

Proof The proofs are identical with the propositional proofs where $+$ and $\Leftrightarrow$ are replaced with $\lor$ and $\Leftrightarrow$. ■

Exercise 9.3.2 Prove Fact 9.3.1 with the proof assistant.

Exercise 9.3.3 Prove $\forall XY^{\mathbb{T}}.\ (X \Leftrightarrow Y) \to (\mathcal{D}(X) \Leftrightarrow \mathcal{D}(Y))$.

Exercise 9.3.4 Prove $\forall X^{\mathbb{T}} f^{X \to \mathsf{B}} x^X.\ \mathcal{D}(fx = \mathsf{true})$.

Exercise 9.3.5 Prove $\forall X^{\mathbb{T}}. (\mathcal{D}(X) \to \bot) \to \bot$.

9.4 Certifying Functions

Our lead example for certifying functions are certifying equality deciders for numbers:

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

Functions of this type take two numbers and decide whether they are equal. The decision is returned with a proof asserting the correctness of the decision. When convenient, we will refer to such proofs as **certificates**.

Constructing a certifying equality decider for numbers in tactic mode is routine. In fact, we have carried out the construction before (Fact 5.9.1) for the proposition $\forall xy^{\mathsf{N}}. (x = y) \lor (x \neq y)$.

Fact 9.4.1 $\forall x^{\mathsf{N}} y^{\mathsf{N}}.\ (x = y) + (x \neq y)$.

Proof By induction on x and case analysis on y with y quantified in the inductive hypothesis. There are four cases which follow with the constructor laws for numbers. The most interesting case is the successor-successor case, where we need to show $(\mathsf{S}x = \mathsf{S}y) + (\mathsf{S}x \neq \mathsf{S}y)$ given the inductive hypothesis $(x = y) + (x \neq y)$. If we have $x = y$, we also have $\mathsf{S}x = \mathsf{S}y$. If we have $x \neq y$, $Sx \neq \mathsf{S}y$ follows with the injectivity of S. ■

Suppose we have a function $F : \forall xy^{\mathsf{N}}.\ (x = y) + (x \neq y)$. Then we know exactly what F gives us without knowing how F is defined. So we can use F in proofs without knowing its definition. The situation is the same as with a propositional lemma $L : \forall xy^{\mathsf{N}}.\ (x = y) \lor (x \neq y)$ that we use without knowing its proof. Recall that we call defined constants with a propositional type *lemmas* if their definition is blocked for reduction. We now generalize the notion of a lemma such that it can have any type. We will use the term **certifying functions** for lemmas with a non-propositional functional type.

Recall that the type of a lemma serves as in interface between the uses of a lemma and possible proofs of the lemma. As with a software interface, the type of a lemma decouples uses of the lemma from proofs of the lemma.

In practice, most functions we construct in computational type theory are accommodated as certifying functions or propositional lemmas. Accommodating a function as a lemma is perfect whenever we have a relational specification of the function. Nevertheless, there are important cases where a function is best specified with defining equations. Example are the basic operations on numbers (e.g., addition) and booleans (e.g., boolean negation).

An interesting kind of functions are eliminators. They are always defined with as inductive functions with defining equations. However, when we destructure assumptions and do case analysis and induction in proofs, we apply the eliminators without using their defining equations. So we could restrict the use of eliminators in proofs to irreducible versions hiding their definitions. There remains one important use of reducible eliminators in that they facilitate local definitions of inductive functions.

To summarize, we will see many certifying functions from now on, and we will always construct them in tactic mode.

Exercise 9.4.2

a) Construct a certifying equality decider $F : \forall xy^{\mathsf{B}}.\ (x = y) + (x \neq y)$.

b) Prove $\forall xy.\ \text{IF } Fxy \text{ THEN } x = y \text{ ELSE } x \neq y$.

c) Define a boolean equality decider $f^{\mathsf{B}\to\mathsf{B}\to\mathsf{B}}$ and prove $\forall xy.\ \text{IF } fxy \text{ THEN } x = y \text{ ELSE } x \neq y$.

d) Prove $\forall xy.\ fxy = \text{IF } Fxy \text{ THEN } \mathsf{true} \text{ ELSE } \mathsf{false}$.

9.5 Certifying Equality Deciders

A certifying equality decider for a type X is a function

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

Given two values of type X, a certifying equality deciders decides whether the values are equal and returns its decision together with a certificate (a proof that the decision is correct).

We say that a type is **discrete** if it has a certifying equality decider. It will be convenient to have a notation for the type of certifying equality deciders for a type X:

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

Formally, $\mathcal{E}$ is a plain function $\mathbb{T} \to \mathbb{T}$.

It turns out that $\perp$, $\mathbb{1}$, B, and N are discrete types, and that discrete types are closed under taking product types and sum types.

Fact 9.5.1 (Transport of equality deciders)

1. $\mathcal{E}(\perp)$, $\mathcal{E}(\mathbb{1})$, $\mathcal{E}(\mathsf{B})$, $\mathcal{E}(\mathsf{N})$.
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 9.4.1. The remaining claims are left as exercises. ■

Discrete types are also closed under injective preimages.

Fact 9.5.2 (Transport)
Injective functions transport equality deciders backwards:
$\forall XY^{\mathbb{T}} \forall f^{X \to Y}.\ \mathsf{injective}\, \mathsf{f} \to \mathcal{E}(Y) \to \mathcal{E}(X)$.

Exercise 9.5.3 Proof Facts 9.5.1 and 9.5.2.

Exercise 9.5.4 Prove $\mathcal{E}(\mathbb{1} \to \perp)$.

Exercise 9.5.5 (Boolean equality)
Define an inductive function $f^{\mathsf{N} \to \mathsf{N} \to \mathsf{B}}$ testing equality of numbers and prove $\forall xy.\ \text{IF } fxy \text{ THEN } x = y \text{ ELSE } x \neq y$.

9.6 Computational Decidability

Computational type theory is designed such that every definable function is algorithmically computable. Thus we can prove that predicates are computationally decidable within computational type theory by constructing (certifying) deciders for them. Decidability proofs in computational type theory are formal computability proofs that avoid the unmanageable formal details coming with explicit models of computation (e.g., Turing machines).

We call a predicate **decidable** if it has a certifying decision function:

$$\mathsf{dec}\,(p^{X\to\mathbb{T}}) \;:=\; \forall x.\,\mathcal{D}(px)$$

Decidable predicates are algorithmically decidable. Moreover, every decidable predicate p is logically decidable in that it satisfies the law of excluded middle $\forall x.\, px \lor \neg px$.

The above definition of $\mathsf{dec}(p)$ is for unary predicates. It can be extended to predicates with two arguments:

$$\mathsf{dec}_2\,(p^{X\to Y\to\mathbb{T}}) \;:=\; \forall xy.\,\mathcal{D}(pxy)$$

We can also define $\mathsf{dec}_0(X) := \mathcal{D}(X)$. We remark that type theory cannot express a uniform and readable definition of a type family dec_n for $n \geq 0$.

So far we have focussed on deciders for the equality predicate. We have seen that the data types $\bot$, $\mathbb{1}$, B, and N have equality deciders, and that this extends to their closure under product and sum types and injective preimages.

We remark that type theory can easily express undecidable predicates. We will formalize Post correspondence problem with an inductive predicate in §15.6.

10 Certifying Functions and Sigma Types

Sigma types are dependent pair types. Sigma types are obtained with a type function $p^{X \to \mathbb{T}}$ and take as values pairs (x, y) with $x : X$ and $y : px$. Sigma types are the computational version of the propositional types for existential quantification.

Sigma types are essential for certifying function types. Given a relation $p^{X \to Y \to \mathbb{P}}$, we can write the function type $\forall x. \Sigma y. pxy$ whose functions take an argument x^X and yield a result y^Y such that pxy. More precisely, functions of type $\forall x. \Sigma y. pxy$ yield for every x a pair (y, a) where $a : pxy$. A concrete example we will consider is a certifying function type

$$\forall x^{\mathsf{N}}. \Sigma n. (x = 2 \cdot n) + (x = 2 \cdot n + 1)$$

for division by 2. Functions of this type take a number x and yield the Euclidean quotient of $x/2$. Moreover, functions of this type decide whether there is a remainder and yield a proof for the correctness of the result (a certificate).

We can translate a certifying function into a simply typed function and a correctness proof. The translation from simply typed to certified is also possible. For certifying deciders, we can state the presence of the translations with the computational equivalence

$$(\forall x. \mathcal{D}(px)) \Leftrightarrow (\Sigma f^{X \to \mathsf{B}}. \forall x. px \longleftrightarrow fx = \mathsf{true})$$

We will define a truncation operator $\square X$ that for a type X yields a proposition $\square X$ that is provable if and only if X is inhabited. Using truncation, the correspondence between disjunctions and sum types and existential quantifications and sigma types can be expressed with the propositional equivalences

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

10.1 Dependent Pair Types

We have seen pair types (better known a product types)

$$(x, y) : X \times Y$$

fixing the types X and Y of their components a priori. *Dependent pair types*

$$(x, y) : \mathsf{sig}\, p$$

employ a type function $p^{X \to \mathbb{T}}$ and admit all pairs (x, y) such that $x : X$ and $y : px$. Thus a dependent pair type doesn't fix the type of the second component a priory but instead determines it for each pair depending on the first component.

Dependent pair types $\mathsf{sig}\, p$ are usually written as

$$\Sigma x^X.px$$

using a quantifier-style notation emphasizing the relationship with existential quantifications $\exists x^X.\, px$ and dependent function types $\forall x^X.\, px$. All three type families accommodate a type dependency with a type function p and generalize their simply typed versions $X \times Y$, $X \wedge Y$, and $X \to Y$. Note that function types are native to the type theory while pair types, conjunctions, and existential quantifications are obtained as inductive types.

The inductive definition of **dependent pair types** is now routine:

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

The definition yields a type constructor and a value constructor as follows:

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

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 often write $\Sigma x^X.px$ for $\mathsf{sig}_X\, p$. Following common speak, we will often refer to dependent pair types as **sigma types**.

We define the **universal eliminator for sigma types**:

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

There is considerable typing bureaucracy but the basic idea is familiar from product types $X \times Y$ and existential quantifications $\exists x^X.\, px$: If we have a pair $s : \mathsf{sig}_X\, p$, we can assume x and y and replace s with (x, y). This idea becomes apparent with the simple type

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

Using the eliminator with the simple type suffices whenever $s : \mathsf{sig}_X\, p$ occurs only once in the context we are working in.

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

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

10.2 Certifying Division

We discuss certifying division as our lead example for certifying functions whose target type is a sigma type:

$$\forall x^{\mathsf{N}}.\ \Sigma n.\ (x = 2 \cdot n) + (x = 2 \cdot n + 1)$$

A function of this type takes a number x and returns the Euclidean quotient n of x and 2 together with a decision of whether x is even or odd. In each case a certificate (i.e., correctness proof) is provided.

Proving that there is a certifying division function of the given type is straightforward. As it comes to the informal proof, there is no difference to proving the propositional version obtained by replacing Σ with $\exists$ and $+$ with $\vee$.

Fact 10.2.1 $\forall x^{\mathsf{N}}.\ \Sigma n.\ (x = 2 \cdot n) + (x = 2 \cdot n + 1)$.

Proof By induction on x. For the zero case $x = 0$, we choose $n = 0$ and prove $x = 2 \cdot 0$. For the successor case $x = \mathsf{S}x'$, the inductive hypothesis gives us n such that $(x' = 2 \cdot n) + (x' = 2 \cdot n + 1)$. If $x' = 2 \cdot n$, we have $x = 2 \cdot n + 1$. If $x' = 2 \cdot n + 1$, we have $x = 2 \cdot \mathsf{S}n$. ■

We remark that the proof uses simply typed applications of the eliminators for sigma and sum types for destructuring the inductive hypothesis. As usual we do not mention these type-theoretic details.

Fact 10.2.2 Let $F : \forall x^{\mathsf{N}}.\ \Sigma n.\ (x = 2 \cdot n) + (x = 2 \cdot n + 1)$ and D and M be functions $\mathsf{N} \rightarrow \mathsf{N}$ defined as follows:

$$\begin{aligned} Dx &:= \text{LET } (n, \_) = Fx \text{ IN } n \\ Mx &:= \text{LET } (\_, a) = Fx \text{ IN IF } a \text{ THEN } 0 \text{ ELSE } 1 \end{aligned}$$

Then $\forall x.\ x = 2 \cdot Dx + Mx$ and $\forall x.\ Mx \leq 1$.

Proof The let expressions used in the definitions of D and M are notation for simply typed applications of the eliminator for sigma types. The conditional in the definition of M is notation for a simply typed application of the eliminator for sum types.

We fix x and prove $x = 2 \cdot Dx + Mx$ and $Dx \leq 1$. Let $Fx = (n, a)$.

If $a = \mathsf{L}\_$, we have $x = 2 \cdot n$, $Dx = n$, and $Mx = 0$. Thus $x = 2 \cdot Dx + Mx$ and $Mx \leq 1$.

If $a = \mathsf{R}\_$, we have $x = 2 \cdot n + 1$, $Dx = n$, and $Mx = 1$. Thus $x = 2 \cdot Dx + Mx$ and $Mx \leq 1$. ■

The type-theoretic details of the proof can be elaborated and verified with a proof assistant. On paper, one follows mathematical intuition and ignores type-theoretic details as done above. In case of doubt, one elaborates the informal proof with a proof assistant.

We provide a few remarks on the type-theoretic details of the above proof. The equation $x = 2 \cdot Dx + Mx$ is shown after unfolding of D and M (conversion rule) by destructuring Fx into (n, a) (dependently typed application of eliminator for sigma types) and destructuring a (simply typed application of the eliminator for sum types). If interested, elaborate the proof with the proof assistant so that all destructuring is done by explicitly applying the eliminators for sigma and sum types.

Note that Fact 10.2.2 makes explicit that a definition of the assumed certifying division function F is not available for the proof.

We remark that in tactic mode eliminators are typically used as certifying functions whose definition is not needed (here the eliminators for numbers in Fact 10.2.1, and the eliminators for sum types and sigma types in Fact 10.2.2). The exception are the applications of the eliminators in the local definitions of D and M in Fact 10.2.2, where the defining equations of the eliminators are needed for the proof.

Exercise 10.2.3 (Certifying distance function)
Prove $\forall xy^{\mathsf{N}} \Sigma z^{\mathsf{N}}.\ (x + z = y) \vee (y + z = x)$.
Hint: Induction on x with y quantified.

Exercise 10.2.4 (Equational match function for numbers)
Construct a certifying function $\forall x^{\mathsf{N}}.\ (x = 0) + (\Sigma k.\ x = \mathsf{S}k)$.

Exercise 10.2.5 (Equational match function for sum types)
Construct a certifying function $\forall a^{X+Y}.\ (\Sigma x.\ a = \mathsf{L}\,x) + (\Sigma y.\ a = \mathsf{R}\,y)$.

Exercise 10.2.6 (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 $\mathsf{N} \to \mathsf{B}$ that tests whether a number is even. Prove the correctness of your function.

Exercise 10.2.7 (Certifying division by 2)
Define a recursive function $\mathsf{N} \to \mathsf{N} \times \mathsf{B}$ such that

$$\forall x.\ \text{LET } (n, b) = fx \text{ IN } x = 2 \cdot n + \text{IF } b \text{ THEN } 0 \text{ ELSE } 1$$

Verify the correctness statement.

Exercise 10.2.8 (Functional characterization)
Prove $\mathsf{sig}\,p \Leftrightarrow \forall Z^{\mathbb{T}}.\ (\forall x.\ px \to Z) \to Z$. Note that the equivalence is analogous to the impredicative characterization of existential quantification.

Exercise 10.2.9 Prove that double negated existential quantification agrees with double negated sigma quantification: $\neg\neg\mathsf{ex}\,p \longleftrightarrow \neg\neg\mathsf{sig}\,p$.

Exercise 10.2.10 Define a function $\forall X^{\mathbb{T}} \forall p^{X \to \mathbb{P}}.\ \mathsf{sig}\,p \to \mathsf{ex}\,p$. Note that a converse function $\mathsf{ex}\,p \to \mathsf{sig}\,p$ cannot be defined because of the propositional discrimination restriction.

10.3 Translation Theorems

We can translate between certifying and boolean deciders for types and equality. Using sigma types, we can state this fact formally.

Recall the notation for certifying deciders for type functions:

$$\mathsf{dec}\,(p^{X \to \mathbb{T}}) \;:=\; \forall x.\ \mathcal{D}(px)$$

A function of type $\mathsf{dec}\,(p^{X \to \mathbb{T}})$ decides for every x whether the type px is inhabited. In case px is inhabited, a witness is returned, and otherwise a proof $px \to \bot$.

Fact 10.3.1 (Decider translations)
1. $\forall X^{\mathbb{T}}.\ \mathcal{E}(X) \Leftrightarrow \Sigma f^{X \to X \to \mathsf{B}}.\, \forall xy.\ x = y \longleftrightarrow fxy = \mathsf{true}$
2. $\forall X\, \forall p^{X \to \mathbb{T}}.\ \ \mathsf{dec}\,p \Leftrightarrow \Sigma f^{X \to \mathsf{B}}.\, \forall x.\ px \Leftrightarrow fx = \mathsf{true}$

Proof We argue (1), claim (2) is similar.

Suppose we have certifying equality decider d for X. Then

$$fxy := \text{IF } dxy \text{ THEN } \mathsf{true} \text{ ELSE } \mathsf{false}$$

satisfies the equivalence.

Suppose we have a boolean function f deciding equality on x. Then we have px if $fxy = \mathsf{true}$, and $px \to \bot$ if $fxy = \mathsf{false}$. ■

Often it is convenient to specify a function $f^{X \to Y}$ with a type function $p^{X \to Y \to \mathbb{T}}$ such that $\forall x. px(fx)$. Given the specification, we may construct either a simply typed function f as specified, or a certifying function $F : \forall x. \Sigma y. pxy$ giving us both f and its correctness proof. In most cases it turns out that constructing a certifying function $F : \forall x. \Sigma y. pxy$ first is the way to go. For a concrete example consider the type function $\lambda xn.\ (x = 2 \cdot n) + (x = 2 \cdot n + 1)$ specifying a certifying division function as discussed in § 10.2

Fact 10.3.2 (Skolem translations)
$\forall XY^{\mathbb{T}}\,\forall p^{X\to Y\to\mathbb{T}}.\ (\forall x\,\Sigma y.\ pxy) \Leftrightarrow (\Sigma f.\,\forall x.\ px(fx))$.

Proof Suppose we have a certifying function $F : \forall x\,\Sigma y.\ pxy$. Then

$$fx := \text{LET } (y, \_) = Fx \text{ IN } y$$

satisfies $px(fx)$ for all x. The other direction is obvious since $px(fx)$. ∎

We speak of Skolem translations since there is a ressemblance with Skolem functions in first-order logic.

You may have noticed that in the proofs of the translation theorems we omitted all type-theoretical details. Discussing them on paper would just be too boring. As always, we recommend stepping through the accompanying Coq scripts. The scripts in turn hide considerable type-theoretic detail since they rely on Coq's destructuring facilities. If you want to see more, do the proofs following the scripts but do all destructuring with explicit eliminator applications.

10.4 Projections

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

$$\begin{array}{lllcllll}
\pi_1 : & \mathsf{sig}\,p \to X & & \qquad & \pi_2 : & \forall a^{\mathsf{sig}\,p}.\ p(\pi_1 a) & \\
\pi_1\,(x,y) & := & x & & \pi_2\,(x,y) & := & y
\end{array}$$

Note that the type of π_2 is given using the projection π_1. The use of π_1 is necessary since the type of the second component of a depends on the first component of a. Type checking the defining equation of π_2 requires a conversion step applying the defining equation of π_1.

Fact 10.4.1 (Eta Law) $\forall a^{\mathsf{sig}\,p}.\ a = (\pi_1 a, \pi_2 a)$.

Proof By computational equality after destructuring of a. ∎

Using the projections we can describe the Skolem translations from Fact 10.3.2 concisely as terms.

$$\begin{aligned}
\lambda F.\,(\lambda x.\pi_1(Fx),\,\lambda x.\pi_2(Fx)) &: (\forall x\,\Sigma y.\ pxy) \to (\Sigma f.\,\forall x.\ px(fx))\\
\lambda ax.\,(\pi_1 ax,\ \pi_2 ax) &: (\Sigma f.\,\forall x.\ px(fx)) \to (\forall x\,\Sigma y.\ pxy)
\end{aligned}$$

We emphasize that in practice one would establish the Skolem translations in tactic mode (see Fact 10.3.2). Still writing the term descriptions using a proof assistant is fun and demonstrates the smooth working of implicit argument inference and type conversion.

Exercise 10.4.2 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 10.4.3 Construct the eliminator for sigma types as a certifying function using the projections and the eta law (Fact 10.4.1).

Exercise 10.4.4 Express the projections π_1 and π_2 for sigma types with terms t_1 and t_2 using the eliminator E_Σ such that $\pi_1 \approx t_1$ and $\pi_2 \approx t_2$.

Exercise 10.4.5 (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.

Exercise 10.4.6 (Propositional Skolem) Due to the propositional discrimination 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 10.4.7 (Existential quantification) Existential quantifications $\mathsf{ex}\, Xp$ are subject to the propositional discrimination 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 $\mathsf{ex}\, Xp$ where X is a proposition.

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

Exercise 10.4.8 (Injectivity laws)
One would think that the injectivity laws for dependent pairs

$$\begin{aligned} \mathsf{E}xy = \mathsf{E}x'y' &\rightarrow x = x' \\ \mathsf{E}xy = \mathsf{E}xy' &\rightarrow y = y' \end{aligned}$$

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 and conversion. In Chapter 29, we will show that the second injectivity law holds if we assume proof irrelevance, or if the type of the first component discrete.

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 $\pi_2(\mathsf{E}xy)$ to $\pi_2(\mathsf{E}xy')$ does not work since there is no well-typed rewrite predicate validating the rewrite.

10.5 Truncation

We define a type constructor $\Box : \mathbb{T} \to \mathbb{P}$ mapping a type X to a proposition $\Box X$ such that the proposition $\Box X$ is provable if and only if the type X is inhabited:

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

We refer to a proposition $\Box X$ as **truncation of X**. We may read a truncation $\Box X$ as "X is inhabited". If X is a computational type, a truncation $\mathsf{T}x : \Box X$ disables the computational facilities coming with the inhabitant $x : X$. This is due to the propositional discrimination restriction for $\Box X$ and matters if X is a sum type or a sigma type.

As one would expect, conjunction, disjunction, and existential quantification can be characterized by the truncations of their computational counterparts (pair types, sum types, and sigma types).

Fact 10.5.1 (Logical truncations)

1. $(P \wedge Q) \longleftrightarrow \Box(P \times Q)$.
2. $(P \vee Q) \longleftrightarrow \Box(P + Q)$.

We obtain truncation with an inductive predicate. Alternatively, truncation can be obtained with existential quantification or impredicatively with function types.

Fact 10.5.2 (Characterizations of truncation)

1. $\forall X^{\mathbb{T}}.\ \Box X \longleftrightarrow \exists x^X.\top$
2. $\forall X^{\mathbb{T}}.\ \Box X \longleftrightarrow \forall Z^{\mathbb{P}}.\ (X \to Z) \to Z$
3. $\forall P^{\mathbb{P}}.\ \ \Box(P) \longleftrightarrow P$

Exercise 10.5.3 Define a simply typed match function for inhabitation types and prove Facts 10.5.1 and 10.5.2. Moreover, prove $\forall P^{\mathbb{P}}.\ \ \Box(P) \longleftrightarrow P$.

Exercise 10.5.4 Prove the following truncation laws for all types X:

a) $X \to \Box X$

b) $\Box X \to (X \to \Box Y) \to \Box Y$

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

d) $\Box X \to \neg\neg X$

e) $\mathsf{XM} \to \Box X \lor \neg X$

Exercise 10.5.5 (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 $\Box(\forall X^{\mathbb{T}}.\,\mathsf{witness}\ X)$. The translation from choice to witness needs to navigate cleverly around the propositional discrimination restriction. The presence of the inhabitation operator is essential for this direction.

a) Prove $\Box(\forall X^{\mathbb{T}}.\,\mathsf{witness}\ X) \to (\forall XY^{\mathbb{T}}.\,\mathsf{choice}\ X\,Y)$.

b) Prove $(\forall XY^{\mathbb{T}}.\,\mathsf{choice}\ X\,Y) \to \Box(\forall X^{\mathbb{T}}.\,\mathsf{witness}\ X)$.

c) Convince yourself that the equivalence

$$(\forall XY^{\mathbb{T}}.\,\mathsf{choice}\ X\,Y) \longleftrightarrow \Box(\forall X^{\mathbb{T}}.\,\mathsf{witness}\ X)$$

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.

10.6 Notes

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 the 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 informative types we can describe computational situations often lacking adequate descriptions in set-theoretic language.

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 the type-theoretic 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. Often, one 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.

Computational type theory originated with Martin-Löf type theory [23]. In contrast to the type theory we are working in, Martin-Löf type theory does not have a special universe for propositions but accommodates propositions as ordinary types. So there is no propositional discrimination restriction and there are no propositional variants of product, sum and sigma types. This simplicity comes at the price that assumptions like excluded middle can only be formulated for all types, which is not meaningful (but consistant). Having an impredicative universe of propositions is a key feature of the computational type theory underlying the Coq proof assistant [9].

11 More Computational Types

The main tool for relating types are injections and bijections. We consider injections and bijections that come with inverse functions. For the types of injections and bijections dependent tuple types are needed. We shall use customised inductive types for injection and bijection types. We show Cantor's theorem for injections. We also show that sigma types can represent sum types and product types up to bijection.

We discuss option types $\mathcal{O}(X)$, which extend a type X with an extra element. Option types $\mathcal{O}(X)$ may be defined as sum types $X + \mathbb{1}$, but we shall use a dedicated inductive type constructor. For option types, we construct a function that from a bijection between $\mathcal{O}(X)$ and $\mathcal{O}(Y)$ obtains a bijection between X and Y. For the construction a clever use of a certifying function is essential.

We then discuss two prominent type families N_n and $\mathsf{V}_n X$ known as numeral types and vector types. We obtain both families by structural recursion on the index n using option types and product types. Numeral types are finite types $\mathcal{O}^n\bot$ obtained by applying the option type constructor to void. Vector types are sequence types $(\lambda Y.X{\times}Y)^n\,\mathbb{1}$ obtained by nesting pair types starting with unit. Both constructions make full use of type conversion.

We remark that the basic notions discussed in this chapter (injections, bijections, numerals, vectors) are computational refinements of notions that have been studied in set theory for a long time.

11.1 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 11.1.1 (Inverse functions)

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

Proof The proofs are straightforward but interesting. Exercise. ▪

We define **injection types**

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

and call their inhabitants **injections**. An injection $\mathcal{I}XY$ is 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 $\Sigma f^{X\to Y}\, \Sigma g^{Y\to X}.\, \mathsf{inv}\, gf$.

Fact 11.1.2 (Reflexivity and Transitivity)
$\mathcal{I}XX$ and $\mathcal{I}XY \to \mathcal{I}YZ \to \mathcal{I}XZ$.

Proof Exercise. ▪

Fact 11.1.3 (Transport)
$\mathcal{I}XY \to \mathcal{E}(Y) \to \mathcal{E}(X)$.

Proof By Fact 9.5.2 it suffices to have an injective function $X \to Y$, which exists by Fact 11.1.1. ▪

Fact 11.1.4 (Cantor) $\mathcal{I}(X \to \mathsf{B})X \to \bot$. That is, $X \to \mathsf{B}$ does not embed into X.

Proof Follows from Fact 6.3.4 since the inverse function of the embedding function is surjective. ▪

A bijection is a symmetric injection where the two functions invert each other in either direction. We define **bijection types**

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

and call their inhabitants **bijections**.

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. This is not necessarily the case for an injection $\mathcal{I}XY$ where the target type Y may have elements not appearing as images of elements of the source type X.

Fact 11.1.5 $\mathcal{B}XY \to \mathcal{I}XY \times \mathcal{I}YX$.

Fact 11.1.6 (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 11.1.7 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 11.1.8 $\mathsf{N} \times \mathsf{N}$ and N are in bijection.

Proof Arithmetic pairing as developed in Chapter 7. ∎

Fact 11.1.9 (Sigma types can express product and sum types)
Sigma types can express product types $X \times Y$ and sum types $X + Y$ up to bijection.

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

Proof The functions for the bijections can be defined with the simply typed match function for sigma types. The proofs of the roundtrip equations, however, require the dependently typed eliminator (with one exception). ∎

Exercise 11.1.10 Prove the claims of all facts stated without proofs.

Exercise 11.1.11 Give a function $f^{\mathsf{N}\to\mathsf{N}}$ that has two non-agreeing inverse functions.

Exercise 11.1.12 (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:
$\forall f^{\mathsf{B}\to\mathsf{B}}.\ \mathsf{inv}\,ff \land (\forall g.\ \mathsf{inv}\,gf \to \mathsf{agree}\,gf)$.

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

Exercise 11.1.13 Show that $\mathsf{N} \to \mathsf{N} \to X$ and $\mathsf{N} \to X$ are in bijection.

Exercise 11.1.14 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 11.1.15
Show that $\mathcal{B}XY$ and $\Sigma f^{X \to Y} \Sigma g^{Y \to X}.\ \mathsf{inv}\, gf \wedge \mathsf{inv}\, fg$ are in bijection:

a) $\mathcal{B}XY \ \Leftrightarrow\ \Sigma f^{X \to Y} \Sigma g^{Y \to X}.\ \mathsf{inv}\, gf \wedge \mathsf{inv}\, fg$.

b) $\mathcal{B}\ (\mathcal{B}XY)\ (\Sigma g^{Y \to X}.\ \mathsf{inv}\, gf \wedge \mathsf{inv}\, fg)$.

Exercise 11.1.16 Prove $\mathcal{B}\,\mathsf{N}\,\mathsf{B} \to \bot$.

Exercise 11.1.17
Prove that bijections transport equality deciders: $\mathcal{B}XY \to \mathcal{E}(X) \to \mathcal{E}(Y)$.

11.2 Option Types

Given a type X, we may see the sum type $X + \mathbb{1}$ as a type extending X with an 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 $\mathcal{O}(X)$.

Fact 11.2.1 (Equality deciders)
Option types transport equality deciders in both directions:
$\forall X^{\mathbb{T}}.\ \mathcal{E}(X) \Leftrightarrow \mathcal{E}(\mathcal{O}(X))$.

Proof Direction $\to$ follows by discrimination on options and both constructor laws for options. Direction $\leftarrow$ follows with the injectivity of the some constructor. ∎

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 clear result needs a clever proof using a certifying function through a Skolem translation.

Suppose f and g provide a bijection between $\mathcal{O}(X)$ and $\mathcal{O}(Y)$. We first define a bijective function $f'^{X \to Y}$. 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 $f'^{X \to Y}$ as described above involves a computational falsity elimination. For this reason we work with a cleverly designed certifying function so that the definition of the function is not needed for proving the required properties. We work with the following specification:

$$\mathsf{lower}\, f^{\mathcal{O}(X) \to \mathcal{O}(Y)}\, f'^{X \to Y} := \forall x.\ \text{MATCH}\ f(^\circ x)\ [\,^\circ y \Rightarrow f'x = y \mid \emptyset \Rightarrow {}^\circ f'x = f\emptyset\,]$$

Lemma 11.2.2 $\forall f^{\mathcal{O}(X) \to \mathcal{O}(Y)}.\ \mathsf{injective}\, f \to \mathsf{sig}\,(\mathsf{lower}\, f)$.

Proof By the Skolem translation (Fact 10.3.2) it suffices to prove $\forall x\, \Sigma z.\ \text{MATCH}\ f(^\circ x)\ [\,^\circ y \Rightarrow z = y \mid \emptyset \Rightarrow {}^\circ z = f\emptyset\,]$. Straightforward. ■

Lemma 11.2.3 $\forall f^{\mathcal{O}(X) \to \mathcal{O}(Y)}.\ \mathsf{lower}\, f f' \to \mathsf{lower}\, g g' \to \mathsf{inv}\, g f \to \mathsf{inv}\, f g \to \mathsf{inv}\, g' f'$.

Proof Case analysis following $\mathsf{lower}\, f f'$ and $\mathsf{lower}\, g g'$. ■

Theorem 11.2.4 (Bijection) $\forall XY.\ \mathcal{B}\,(\mathcal{O}(X))\,(\mathcal{O}(Y)) \to \mathcal{B} XY$.

Proof Follows with Lemmas 11.2.2 and 11.2.3. ■

Exercise 11.2.5 (Eliminator and constructor laws)
Define an eliminator for option types and use it to prove the constructor laws. Follow the techniques used for numbers in §4.3.

Exercise 11.2.6 Prove $\forall a^{\mathcal{O}(X)}.\ a \neq \emptyset \Leftrightarrow \Sigma x.\ a = {}^\circ x$.
The direction $\to$ needs computational falsity elimination.

Exercise 11.2.7 Prove $\forall f^{X \to \mathcal{O}(Y)}.\ (\forall x.\ fx \neq \emptyset) \to \forall x\, \Sigma y.\ fx = {}^\circ y$.
Note the need for computational falsity elimination. Show that assuming the above claim yields computational falsity elimination in the form $\forall X^{\mathbb{T}}.\ \bot \to X$ (instantiate with $X := \bot$, $Y := X$, and $f = \lambda\_.\emptyset$).

Exercise 11.2.8 Prove $\forall x^{\mathcal{O}^3 \bot}.\ x = \emptyset \lor x = {}^\circ\emptyset \lor x = {}^{\circ\circ}\emptyset$.

Exercise 11.2.9 (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 11.2.10 Prove $\mathcal{B}XY \to \mathcal{B}(\mathcal{O}X)(\mathcal{O}Y)$.

Exercise 11.2.11 Prove the bijection theorem 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 11.2.4.

Exercise 11.2.12 (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 11.2.13 (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 11.2.14 (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.

11.3 Numeral Types

We define **numeral types** recursively:

$$\begin{aligned} \mathsf{N}_0 &:= \bot \\ \mathsf{N}_{\mathsf{S}n} &:= \mathcal{O}(\mathsf{N}_n) \end{aligned}$$

By construction, N_n is a finite type with n elements. For instance, the elements of N_3 are $\emptyset$, ${}^\circ\emptyset$, ${}^{\circ\circ}\emptyset$. We may think of the elements of $\mathsf{N}_{\mathsf{S}n}$ as the numbers $0, \dots, n$. Formally, numeral types are obtained with an inductive function $\mathsf{N} \to \mathbb{T}$.

Fact 11.3.1 (Equality Deciders) $\mathcal{E}(\mathsf{N}_n)$.

Proof By induction on n using Facts 9.5.1(1) and 11.2.1. ■

We can now give a formal definition of what it means that a type has n elements.

Definition 11.3.2 A type X has **cardinality** n if it is in bijection with N_n. Moreover, a type X is **finite** if it is in bijection with some numeral type N_n.

We would like to prove that a type X has at most one cardinality. Suppose X is in bijection with N_m and N_n. Then N_m and N_n are in bijection.

Fact 11.3.3 (Cardinality) $\mathcal{B}\,\mathsf{N}_m\,\mathsf{N}_n \to m = n$.

Proof By induction on m and discrimination on n with n quantified in the inductive hypothesis. There are three nontrivial cases, two of which follows by contradiction since N_0 and $\mathsf{N}_{\mathsf{S}k}$ cannot be in bijection. The remaining case $\mathcal{B}\,\mathsf{N}_{\mathsf{S}m}\,\mathsf{N}_{\mathsf{S}n} \to \mathsf{S}m = \mathsf{S}n$ follows with the bijection theorem 11.2.4 for option types and the inductive hypothesis. ■

Exercise 11.3.4 (Decidable quantification)
Let d be a certifying decider for $p : \mathsf{N}_n \to \mathbb{T}$. Prove the following:

a) $(\Sigma x.px) + (\forall x.px \to \bot)$

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

c) $\mathcal{D}(\Sigma x.px)$

Hint: Use induction on the cardinality n.

11.4 Vector Types

Vectors are sequences $\langle x_1, \dots, x_n \rangle$ of values from a common base type. We define **vector types** $\mathsf{V}_n X$ recursively:

$$\begin{aligned} \mathsf{V}_0 X &:= \mathbb{1} \\ \mathsf{V}_{\mathsf{S}n} X &:= X \times \mathsf{V}_n X \end{aligned}$$

Formally, vector types are obtained with an inductive function $\mathsf{V} : \mathsf{N} \to \mathbb{T} \to \mathbb{T}$ recursing on numbers and representing vectors as nested pairs.[1] As an example, we offer

$$(1, (2, (3, \mathsf{I}))) \ : \ \mathsf{V}_3 \mathsf{N} \ \approx \ \mathsf{N} \times (\mathsf{N} \times (\mathsf{N} \times \mathbb{1}))$$

The single element of $\mathsf{V}_0 X \approx \mathbb{1}$ is I, and the elements of $\mathsf{V}_{\mathsf{S}n} X \approx X \times \mathsf{V}_n X$ are pairs (x, v) where $x : X$ and $v : \mathsf{V}_n X$.

Fact 11.4.1 (Equality deciders) $\mathcal{E}(X) \to \mathcal{E}(\mathsf{V}_n X)$.

Proof By induction on n. We have a decider $\mathcal{E}(\mathsf{V}_0 X)$ since $\mathsf{V}_0 X \approx \bot$. Moreover, we have a decider $\mathcal{E}(\mathsf{V}_{\mathsf{S}n} X)$ since $\mathsf{V}_{\mathsf{S}n} X \approx X \times \mathsf{V}_n X$, we have a decider $\mathcal{E}(X)$, and the inductive hypothesis gives us a decider $\mathcal{E}(\mathsf{V}_n X)$. ■

[1] The nested pair representation is well-known from lists. It appeared in set theory where it us used to represent tuples of arbitrary length.

Vector types can be seen as refinement of list types where in addition to the element type the length is prescribed. We define a dedicated constant nil for the empty vector and list-style operations cons, hd, and tl:

$$\begin{aligned} &\mathsf{nil} : \forall X.\ \mathsf{V}_0 X && \mathsf{hd} : \forall X n.\ \mathsf{V}_{\mathsf{S}n} X \to X \\ &\mathsf{nil} := \mathsf{I} && \mathsf{hd}\, X n\, (x, v) := x \\[1ex] &\mathsf{cons} : \forall X n.\ X \to \mathsf{V}_n X \to \mathsf{V}_{\mathsf{S}n} X && \mathsf{tl} : \forall X n.\ \mathsf{V}_{\mathsf{S}n} X \to \mathsf{V}_n X \\ &\mathsf{cons}\, X n\, x v := (x, v) && \mathsf{tl}\, X n\, (x, v) := v \end{aligned}$$

Note that we have defined hd and tl as inductive functions and that conversion is needed for type checking the patterns of the defining equations. We remark that for list types hd cannot be defined as a total function since we don't have a default value for the empty list. We will treat X and n as implicit arguments.

Fact 11.4.2 (Eta law) $\mathsf{cons}\, (\mathsf{hd}\, a)(\mathsf{tl}\, a) = a$.

Proof Discrimination on $a : \mathsf{V}_{\mathsf{S}n} X$ and computational equality. ■

We define a function returning arithmetic vectors:

$$\begin{aligned} \mathsf{enum} &: \mathsf{N} \to \forall n.\ \mathsf{V}_n \mathsf{N} \\ \mathsf{enum}\, k\, 0 &:= \mathsf{nil} \\ \mathsf{enum}\, k\, (\mathsf{S}n) &:= \mathsf{cons}\, k\, (\mathsf{enum}\, (\mathsf{S}k)\, n) \end{aligned}$$

For instance, we have $\mathsf{enum}\, 1\, 3 \approx \mathsf{cons}\, 1\, (\mathsf{cons}\, 2\, (\mathsf{cons}\, 3\, \mathsf{nil}))$.

Exercise 11.4.3 (Tuple types) Tuple types generalize vector types in that they determine their component types with a type function $\mathsf{N} \to \mathbb{T}$:

$$\begin{aligned} \mathsf{tup} &: (\mathsf{N} \to \mathbb{T}) \to \mathsf{N} \to \mathbb{T} \\ \mathsf{tup}\, p\, 0 &:= \mathsf{I} \\ \mathsf{tup}\, p\, (\mathsf{S}n) &:= pn \times \mathsf{tup}\, p\, n \end{aligned}$$

Construct functions as follows:

a) $\forall p^{\mathsf{N} \to \mathbb{T}}.\ (\forall n.\ \mathsf{tup}\, pn \to pn) \to \forall n.\ \mathsf{tup}\, pn$

b) $\forall p^{\mathsf{N} \to \mathbb{T}}.\ (\forall n.\ \mathsf{tup}\, p\, n \to pn) \to \forall n.\ pn$

11.4.1 Position Element Maps

We may number the positions of a vector from left to right starting with 0. We define a function that given a vector and a position returns the element at the position:

$$\begin{aligned} \mathsf{sub} :\ & \forall X n.\ \mathsf{V}_{\mathsf{S}n} X \to \mathsf{N} \to X \\ \mathsf{sub}\, X\, 0\, v\, k \ &:=\ \mathsf{hd}\, v \\ \mathsf{sub}\, X\, (\mathsf{S}n)\, v\, 0 \ &:=\ \mathsf{hd}\, v \\ \mathsf{sub}\, X\, (\mathsf{S}n)\, v\, (\mathsf{S}k) \ &:=\ \mathsf{sub}\, X n (\mathsf{tl}\, v)\, k \end{aligned}$$

Note that the primary discrimination is on n. Also note that the type of sub ensures that in case the given position k is too large the last element of the vector can be returned.

Note that the numeral type Nn has exactly as many elements as the vectors of type V_n have positions. Using numerals, we can define a safe position element map always returning the element at the given position:

$$\begin{aligned} \mathsf{sub}' :\ & \forall X n.\ \mathsf{V}_n X \to \mathsf{N}_n \to X \\ \mathsf{sub}' X\, 0\, v\, a \ &:=\ \mathsf{E}_\perp X a \\ \mathsf{sub}' X\, (\mathsf{S}n)\, v\, {}^\circ a \ &:=\ \mathsf{sub}' X n\, (\mathsf{tl}\, v)\, a \\ \mathsf{sub}' X\, (\mathsf{S}n)\, v\, \emptyset \ &:=\ \mathsf{hd}\, v \end{aligned}$$

Note the use of the eliminator for void in the contradictory case handled by the first defining equation.

Exercise 11.4.4 We treat X and n as implicit arguments of sub and sub′. Prove the following equations. For each equation, first determine the most general type for v.

$$\begin{aligned} \mathsf{sub}\, v\, 0 &= \mathsf{hd}\, v & \qquad \mathsf{sub}'\, v\, \emptyset &= \mathsf{hd}\, v \\ \mathsf{sub}\, v\, 1 &= \mathsf{hd}(\mathsf{tl}\, v) & \qquad \mathsf{sub}'\, v\, {}^\circ\emptyset &= \mathsf{hd}(\mathsf{tl}\, v) \end{aligned}$$

Exercise 11.4.5

a) Define a function $\mathsf{last} : \forall X n.\ \mathsf{V}_{\mathsf{S}n} X \to X$ returning the element at the last position of a vector.

b) Prove $\forall v^{\mathsf{V}_{\mathsf{S}n} X}.\ \mathsf{sub}\, v n = \mathsf{last}\, v$.

c) Define a function $\mathsf{snoc} : \forall X n.\ \mathsf{V}_n \to X \to \mathsf{V}_{\mathsf{S}n}$ appending an element at the end of a vector.

d) Prove $\mathsf{last}(\mathsf{snoc}\, v x) = x$.

e) Define a function $\mathsf{rev} : \forall X n.\ \mathsf{V}_n X \to \mathsf{V}_n X$ reversing a vector.

f) Prove $\mathsf{rev}\, (\mathsf{rev}\, v) = v$.

11.4.2 Concatenation

We define a concatenation operation for vectors.

$$\begin{aligned} \mathsf{con} :\ & \forall Xmn.\, \mathsf{V}_m X \to \mathsf{V}_n X \to \mathsf{V}_{m+n} X \\ \mathsf{con}\, X\, 0\, n v w \ &:=\ w \\ \mathsf{con}\, X\, (\mathsf{S}m)\, n v w \ &:=\ \mathsf{cons}\,(\mathsf{hd}\, v)\,(\mathsf{con}\, X m n (\mathsf{tl}\, v) w) \end{aligned}$$

We treat X, m, n as implicit arguments and write $v +\!\!\!+\, w$ for $\mathsf{con}\, v w$.

Now suppose we have three vectors $u : \mathsf{V}_m X$, $v : \mathsf{V}_n X$, $w : \mathsf{V}_k X$. Then the two concatenations

$$\begin{aligned} (u +\!\!\!+\, v) +\!\!\!+\, w &: \mathsf{V}_{(n+m)+k} X \\ u +\!\!\!+\,(v +\!\!\!+\, w) &: \mathsf{V}_{n+(m+k)} X \end{aligned}$$

have incompatible types (that is, nonconvertible types) although they yield the same value. Thus the associativity law for vector concatenation cannot be expressed directly. The problem goes away if m, n, and k are concrete numbers rather than variables.

The type checking problem can be bypassed with a dedicated cast function:

$$\begin{aligned} \mathsf{cast} :\ & \forall Xmnk.\, \mathsf{V}_{(m+n)+k} X \to \mathsf{V}_{m+(n+k)} X \\ \mathsf{con}\, X\, 0\, n k v \ &:=\ v \\ \mathsf{con}\, X\, (\mathsf{S}m)\, n k v \ &:=\ \mathsf{cons}\,(\mathsf{hd}\, v)\,(\mathsf{cast}\, X m n k\,(\mathsf{tl}\, v)) \end{aligned}$$

We treat X, m, n, k as implicit arguments.

Fact 11.4.6 (Associativity)
$\forall u^{\mathsf{V}_m X}\ \forall v^{\mathsf{V}_n X}\ \forall w^{\mathsf{V}_k X}.\ \mathsf{cast}\,((u +\!\!\!+\, v) +\!\!\!+\, w) = u +\!\!\!+\,(v +\!\!\!+\, w)$.

Proof By induction on m with u quantified. Straightforward. ∎

The formulation and proof of the associativity law for vector concatenation come with massive type conversion and with massive elaboration of implicit arguments. This is quite feasible with a proof assistant but too tedious to be done with pen and paper.

Exercise 11.4.7 Prove $\mathsf{V}_{(n+m)+k} X = \mathsf{V}_{n+(m+k)} X$.

Exercise 11.4.8 Convince yourself that the equation

$$(\mathsf{enum}\, 0\, 3 +\!\!\!+\, \mathsf{enum}\, 3\, 3) +\!\!\!+\, \mathsf{enum}\, 6\, 3 = \mathsf{enum}\, 0\, 3 +\!\!\!+\,(\mathsf{enum}\, 3\, 3 +\!\!\!+\, \mathsf{enum}\, 6\, 3)$$

type checks and holds by computational equality.

12 Linear Arithmetic

Numbers $0, 1, 2, \ldots$ are the basic recursive data structure. Starting from the inductive definition of numbers, we study the algebraic properties of addition and truncating subtraction. Comparisons $x \leq y$ will be obtained as equations $x - y = 0$.

The system obtained with addition and subtraction of numbers is called linear arithmetic. Proof assistants come with automatic provers for linear arithmetic freeing users from knowing the basic lemmas for numbers. Linear arithmetic provers realize an abstraction level that is commonly used in informal proofs.

We also define multiplication and prove its basic algebraic properties.

Studying linear arithmetic in computational type theory starting from first principles is fun. As always, the thrill is in finding the right definitions and the right theorems in the right order. There is beauty and elegance in the development presented here.

12.1 Inductive Definition of Numbers

The type of numbers $0, 1, 2 \ldots$ is obtained 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 inductive functions. A basic inductive function is an eliminator providing for inductive proofs:

$$\begin{aligned} \mathsf{E_N} :\ & \forall p^{\mathsf{N} \to \mathbb{T}}.\ p\,0 \to (\forall x.\ px \to p(\mathsf{S}x)) \to \forall x.px \\ \mathsf{E_N}\, p e_1 e_2\, 0 \ &:= \ e_1 \\ \mathsf{E_N}\, p e_1 e_2\, (\mathsf{S}x) \ &:= \ e_2 x (\mathsf{E_N}\, p a f x) \end{aligned}$$

A discussion of the eliminator appears in §5.7. Matches for numbers can be obtained as applications of the eliminator where no use of the inductive hypothesis is made. Or more directly, a specialized elimination function for matches omitting the inductive hypothesis can be defined.

We shall often use the if-then-else notation for zero testing:

$$\text{IF } s \text{ THEN } s_1 \text{ ELSE } s_2 \quad \rightsquigarrow \quad \text{MATCH } s\ [\,0 \Rightarrow s_1 \mid \mathsf{S}\ \_ \Rightarrow s_2\,]$$

Fact 12.1.1 (Constructor laws)

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

Proof The results follow with matches, rewriting, and conversion. The proofs are discussed in §4.3. ■

Exercise 12.1.2 Prove $\mathsf{S}x \neq x$.

12.2 Addition

We accommodate addition of numbers with an inductive function:

$$+ : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$$

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

Fact 12.2.1 (Commutativity) $x + y = y + x$.

Proof By induction on y using the lemmas $x + 0 = x$ and $x + \mathsf{S}y = \mathsf{S}x + y$. Both lemmas follow by induction on x. ■

We remark that addition is not symmetric at the level of computational equality although it is commutative at the level of propositional equality. This unpleasant situation cannot be avoided in the type theory we work in.

Fact 12.2.2 (Associativity) $(x + y) + z = x + (y + z)$.

Proof By induction on x. ■

Associativity and commutativity of addition are used tacitly in informal proofs. moreover, parentheses may be omitted; For instance, $x + y + z \rightsquigarrow (x + y) + z$. The symmetric versions $x + 0 = x$ and $x + \mathsf{S}y = \mathsf{S}(x + y)$ of the defining equations may also be used tacitly.

Fact 12.2.3 (Zero propagation) $x + y = 0 \longleftrightarrow x = 0 \land y = 0$.

Proof By discrimination on x and constructor disjointness. ■

Fact 12.2.4 (Injectivity) $x + y = x + y' \to y = y'$

Proof By induction on x. ■

Corollary 12.2.5 (Injectivity) $x + y = x \to y = 0$.

Proof Follows with injectivity since $x = x + 0$. ■

Corollary 12.2.6 (Contradiction) $x + \mathsf{S}y \neq x$.

Proof Follows with injectivity since $x = x + 0$. ■

12.3 Subtraction

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

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

The recursion is on the first argument. The successor case is realized with a secondary discrimination on the second argument.

Fact 12.3.1 $x - 0 = x$.

Proof Discrimination of x and computational equality. ∎

Fact 12.3.2

1. $x + y - x = y$
2. $x - (y + z) = x - y - z$

Proof Both claims follow by induction on x and discrimination on y. ∎

Corollary 12.3.3

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

12.4 Comparisons

We define **comparisons** using truncating subtraction:

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

We define the usual notational variants for comparisons:

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

We refer to the predicate $(\le) : \mathsf{N} \to \mathsf{N} \to \mathbb{P}$ as **order relation**.

Fact 12.4.1 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. $0 < \mathsf{S}x$

Fact 12.4.2

1. $x \leq x + y$
2. $x \leq x$
3. $x \leq \mathsf{S}x$
4. $x - y \leq x$

Proof (1) follows with Fact 12.3.3(2). (2) and (3) follow from (1). (4) follows with Fact 12.3.2(2) and (1). ■

Fact 12.4.3 (Additive characterization) $x \leq y \longleftrightarrow x + (y - x) = y$.

Proof By induction on x with y quantified followed by discrimination on y. ■

Fact 12.4.4 (Antisymmetry) $x \leq y \to y \leq x \to x = y$.

Proof Follows by equational reasoning after $x \leq y$ is replaced with the additive characterization. ■

Fact 12.4.5 (Transitivity)

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

Proof For (1), replace z and then y in the claim with the additive characterization for the assumptions. The reduced claim follows with Fact 12.4.2(1).
(2) is an instance of (1), and (3) is an instance of (1) modulo conversion. ■

Fact 12.4.6 $x - y = \mathsf{S}z \to y < x$.

Proof By induction on x with y quantified followed by discrimination on y in the successor case. ■

Fact 12.4.7 (Contradictions)

1. $(x < 0) \to \bot$
2. $(x + y < x) \to \bot$
3. $(x < x) \to \bot$

Proof (1) follows by the disjointness constructor law. (2) follows by Fact 12.3.2(1). (3) follows from (2). ■

Fact 12.4.8 (Negative Characterizations)

1. $x \le y \longleftrightarrow (y < x \to \bot)$
2. $x < y \longleftrightarrow (y \le x \to \bot)$

Proof (2) is a conversion instance of (1). (1) follows by induction on y followed by discrimination on y with y quantified in the inductive hypothesis. ■

Exercise 12.4.9 (Equality by contradiction)
Prove $(x < y \to \bot) \to (y < x \to \bot) \to x = y$.

Exercise 12.4.10 Prove the following:

a) $x \le y \;\to\; x \le \mathsf{S}y$

b) $x < y \to x \le y$

c) $x + y \le z \to x \le z$

d) $x \le y \;\to\; x \le y + z$

e) $x + y \le x + z \longleftrightarrow y \le z$.

f) $0 < x \;\to\; x - \mathsf{S}y < x$.

Exercise 12.4.11 (Brute force proofs)
Most facts about comparisons can be proved without lemmas using a single induction and case analysis on variables. A notable exception is $x - y \le x$.

Prove the following types not using lemmas.

a) $x \le x$

b) $x \le y \to y \le z \to x \le z$

c) $x < y \to y \le z \to x < z$

d) $x \le y \to y < z \to x < z$

e) $x \le y \to y \le x \to x = y$

f) $x \le x + y$

g) $x \le 0 \to x = 0$

h) $x \le y \to x + (y - x) = y$

i) $(x \le y) + (y < x)$

j) $x \le y \longleftrightarrow \neg(y < x)$

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

l) IF $x - y$ THEN $x \le y$ ELSE $y < x$

m) IF $(x - y) + (y - x)$ THEN $x = y$ ELSE $x \ne y$

12.5 Arithmetic Testers and Deciders

We say that a function $f : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ is **reducible** if fst reduces to a canonical term whenever s and t are canonical terms. Addition and subtraction are reducible functions.

We say that a reducible function $f : \mathsf{N} \to \mathsf{N} \to \mathsf{N}$ is an **arithmetic tester** for a predicate $p : \mathsf{N} \to \mathsf{N} \to \mathbb{P}$ if the proposition $\text{IF } fxy \text{ THEN } px \text{ ELSE } \neg px$ is provable.

Fact 12.5.1 Predicates that have an arithmetic tester have a certifying decider.

Proof Trivial. ∎

Fact 12.5.2 (Arithmetic testers)
The predicates $x \le y$, $x < y$, and $x = y$ have arithmetic testers:

$$\text{IF } x - y \text{ THEN } x \le y \text{ ELSE } \neg(x \le y)$$
$$\text{IF } \mathsf{S}x - y \text{ THEN } x < y \text{ ELSE } \neg(x < y)$$
$$\text{IF } (x - y) + (y - x) \text{ THEN } x = y \text{ ELSE } x \neq y$$

Proof All three propositions follow by discrimination on the guard of the conditional. For the third proposition we need memorizing discrimination. In the zero case, zero propagation (Fact 12.2.3) gives us $x \le y$ and $y \le x$, so the claim follows with antisymmetry. In the successor case, we assume $x = y$ and obtain a contradiction with constructor disjointness since $(x - x) + (x - x) = 0$ with Fact 12.3.3(1). ∎

Corollary 12.5.3 (Certifying deciders)
The predicates $x \le y$, $x < y$, and $x = y$ have certifying deciders.

Based on Fact 12.5.2 we shall use the notations

$$\begin{array}{lcl} \text{IF } s \le t \text{ THEN } u \text{ ELSE } v & \rightsquigarrow & \text{IF } s - t \text{ THEN } u \text{ ELSE } v \\ \text{IF } s < t \text{ THEN } u \text{ ELSE } v & \rightsquigarrow & \text{IF } \mathsf{S}\,s - t \text{ THEN } u \text{ ELSE } v \\ \text{IF } s = t \text{ THEN } u \text{ ELSE } v & \rightsquigarrow & \text{IF } (s - t) + (t - s) \text{ THEN } u \text{ ELSE } v \end{array}$$

Fact 12.5.4 (Certifying Deciders)

a) $\forall xy.\ (x \le y) + (y < x)$

b) $\forall xy.\ (x \le y) \to (x < y) + (x = y)$

c) $\forall xy.\ (x \le y) \to (y \le \mathsf{S}x) \to (x = y) + (y = \mathsf{S}x)$ (tightness)

Proof All claims follow by induction on x and discrimination on y. ∎

Exercise 12.5.5 Define a boolean decider for $x \le y$ and prove its correctness.

12.6 Linear Arithmetic Prover

Proof assistants come with terminating provers that for linear arithmetic propositions construct a proof whenever the proposition is provable. Linear arithmetic propositions are obtained with $=$, $\leq$, $\bot$, $\to$, $\wedge$, and $\vee$ from linear arithmetic expressions obtained with $+$, $-$, and S from numbers and variables. In Coq a linear arithmetic prover is available through the automation tactic lia (linear integer arithmetic). A linear arithmetic prover frees the user from knowing the lemmas for linear arithmetic, a service making a dramatic difference with more involved proofs where the details of linear arithmetic would be overwhelming. A linear arithmetic prover will for instance find a proof for $\neg(x > y) \longleftrightarrow \neg(x \geq y) \vee \neg(x \neq y)$.

From now on we will write proofs involving numbers assuming the abstraction level provided by linear arithmetic.

Coq defines comparisons $x \leq y$ with an inductive type constructor, a definition that is quite different from our definition. The difference doesn't matter if comparisons are handled with lia, and we will do this from now on.

Coq's automation tactic lia cannot do type sums. Still, a certifying decider like $\forall xy.\ (x \leq y) + (y < x)$ can be constructed with a single memorizing arithmetic discrimination and linear arithmetic. The trick is matching on the number $x - y$, which determines the result decision. The certificates are then obtained with the linear arithmetic prover. Similar ideas work for the other deciders in § 12.5.

Exercise 12.6.1 Define the certifying deciders in § 12.5 in Coq using in each case a single memorizing arithmetic discrimination and the linear arithmetic prover lia.

12.7 Multiplication

We accommodate addition of numbers with a recursively defined function:

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

With this definition the equations

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

hold by computational equality.

Fact 12.7.1 (Distributivity)

a) $(x + y) \cdot z = x \cdot z + y \cdot z$

b) $(x - y) \cdot z = x \cdot z - y \cdot z$

Proof Both equations follow by induction on x and linear arithmetic. For the equation with subtraction discrimination on y is needed in the successor case. Hence y must be quantified in the inductive hypothesis. ■

Fact 12.7.2 (Associativity) $(x \cdot y) \cdot z = x \cdot (y \cdot z)$.

Proof By induction on x. The successor case follows with distributivity over "+". ■

Fact 12.7.3 (Commutativity) $x \cdot y = y \cdot x$.

Proof By induction on y using $x \cdot 0 = 0$ in the zero case and $x \cdot \mathsf{S}y = x + x \cdot y$ in the successor case. Both lemmas follow by induction on x. The successor case of the successor lemma needs linear arithmetic. ■

12.8 Notes

The inclined reader may compare the computational development of linear arithmetic given here with Landau's [22] classical set-theoretic development from 1929.

13 Least Witness Operators

A least witness operator (LWO) decides for a decidable predicate $p^{\mathsf{N} \to \mathbb{P}}$ and a bound n whether p is satisfied by some $k \leq n$. In this is the case, the operator returns the least such k called the least witness of p.

The most principled way to obtain an LWO is, as usual, to construct it as a certifying function in tactic mode. Based on the certifying LWO, we construct certifying deciders for finite quantifications $\forall k{<}n.\ pk$ and $\exists k{<}n.\ pk$.

We also present and verify a number of simply typed reducible least witness functions. The correctness proofs for these functions provide us the with opportunity to demonstrate basic program verification techniques in the context of computational type theory.

We define divisibility and primality of numbers with finite quantification and obtain deciders for the corresponding predicates with the general deciders for finite quantification.

On the foundational side, and related to LWOs, we prove that satisfiable predicates on numbers have least witnesses if and only if the law of excluded middle holds.

13.1 Least Witness Predicate

In this chapter, p will denote a predicate $\mathsf{N} \to \mathbb{P}$ and n and k will denote numbers. We will use the notations (with $\leq$)

$$\begin{aligned} \forall k{<}n.\ pk &:= \forall k.\ k < n \to pk \\ \exists k{<}n.\ pk &:= \exists k.\ k < n \land pk \\ \Sigma k{<}n.\ pk &:= \Sigma k.\ k < n \land pk \end{aligned}$$

and speak of **finite quantifications**. We say that n is a **witness of p** if pn is provable, and that **p is satisfiable** if $\exists n. pn$ is provable. We define a **least witness predicate** as follows:

$$\begin{aligned} \mathsf{safe}\ p\ n &:= \forall k{<}n.\ \neg pk \\ \mathsf{least}\ p\ n &:= pn \land \mathsf{safe}\ pn \end{aligned}$$

Note that $\mathsf{safe}\ pn$ says that no number below n is a witness.

Fact 13.1.1 (Uniqueness)
Least witnesses are unique: $\mathsf{least}\, p\, n \to \mathsf{least}\, p\, n' \to n = n'$.

Proof Follows with antisymmetry. ■

Fact 13.1.2 (Safety propagation)

1. $\mathsf{safe}\, p 0$ (init)
2. $\mathsf{safe}\, pn \to \neg pn \to \mathsf{safe}\, p(\mathsf{S}n)$ (upgrade)

Proof (1) follows with linear arithmetic. For (2), we have the assumption $k < \mathsf{S}n$. Thus $k < n$ or $k = n$. Both cases are straightforward. ■

Exercise 13.1.3 Prove the following:

a) $\mathsf{safe}\ pn \longleftrightarrow \forall k.\ pk \to k \geq n$
b) $\mathsf{safe}\ p(\mathsf{S}n) \longleftrightarrow \mathsf{safe}\ pn \land \neg pn$
c) $\mathsf{safe}\, pn \to k \leq n \to pk \to k = n$

Exercise 13.1.4 (Subtraction) Prove $x - y = z \longleftrightarrow \mathsf{least}\,(\lambda z.\, x \leq y + z)\, z$.

Exercise 13.1.5 (Extensionality)
Prove $(\forall n.\ pn \longleftrightarrow p'n) \to (\forall n.\ \mathsf{least}\, pn \longleftrightarrow \mathsf{least}\, p'n)$.

13.2 Certifying Least Witness Operators

A least witness operator (LWO) decides for a decidable predicate $p^{\mathsf{N}\to\mathbb{P}}$ and a bound n whether p is satisfied by some $k < n$. In the positive case the operator yields the least such k (that is, the least witness of p). In the negative case the operator yields a certificate that there is no witness below n.

Fact 13.2.1 (Certifying LWO)
There is a function $\mathsf{dec}\, p \to \forall n.\ (\Sigma k{<}n.\ \mathsf{least}\, pk) + \mathsf{safe}\, pn$.

Proof By induction on n using the propagation rules for safety (Fact 13.1.2). The zero case is trivial since we have $\mathsf{safe}\, p\, 0$. In the successor case we have $(\Sigma k{<}n.\ \mathsf{least}\, pk) + \mathsf{safe}\, pn$ and need

$$(\Sigma k{<}\mathsf{S}n.\ \mathsf{least}\, pk) + \mathsf{safe}\, p(\mathsf{S}n)$$

The case $\Sigma k{<}n.\ \mathsf{least}\, pk$ is straightforward. Otherwise, we have $\mathsf{safe}\, pn$. If pn, we have $\mathsf{least}\, pk$ and the claim follows. Otherwise, we have $\neg pn$ and hence $\mathsf{safe}\, p(\mathsf{S}n)$ by the safety upgrade rule (Fact 13.1.2(2)). ■

Corollary 13.2.2 (Certifying LWO)
There is a function $\mathsf{dec}\, p \to \forall n.\ (\Sigma k{\le}n.\ \mathsf{least}\, pk) + \mathsf{safe}\, p(\mathsf{S}n)$.

Proof Instantiate the LWO from Fact 13.2.1 with $\mathsf{S}n$. ■

Given a witness, we can obtain a least witness.

Corollary 13.2.3 (Informed LWO)
There is a function $\mathsf{dec}\, p \to \mathsf{sig}\, p \to \mathsf{sig}\, (\mathsf{least}\, p)$.

Proof Straightforward with the LWO from Fact 13.2.1. ■

Corollary 13.2.4 There is a function $\mathsf{dec}\, p \to \mathsf{ex}\, p \to \mathsf{ex}\, (\mathsf{least}\, p)$.

Proof Straightforward with the LWO from Fact 13.2.1. ■

Fact 13.2.5 (Existential least witness operator)
There is a function $\mathsf{dec}\, p \to \mathsf{ex}\, p \to \mathsf{sig}(\mathsf{least}\, p)$.

Proof Immediate with Corollary 13.2.3 and Fact 14.2.2. ■

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 13.2.6 (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. ■

Exercise 13.2.7 (Greatest witness operator)
Let $\mathsf{greatest}\, pnk := pk \wedge (k < n) \wedge (\forall i.\ k{<}i{<}n \to \neg pi)$. Construct a certifying function $\forall n.\ \mathsf{sig}(\mathsf{greatest}\, pn) + (\forall k{<}n.\ \neg pk)$ computing the greatest witness below n of p if there is one.

13.3 Decidability Results

Fact 13.3.1 (Decidability)

1. $\mathsf{dec}\, p \to \mathsf{dec}(\mathsf{safe}\, p)$
2. $\mathsf{dec}\, p \to \mathsf{dec}(\mathsf{least}\, p)$

Proof The construction of the decider (1) is straightforward with the LWO from Fact 13.2.1. The construction of (2) is straightforward with the decider (1) ■

Finite quantifications over decidable predicates satisfy both de Morgan laws.

Fact 13.3.2 (De Morgan laws for finite quantifications)

1. $\neg(\exists k{<}n.\ pk) \longleftrightarrow (\forall k{<}n.\ pk)$
2. $\mathsf{dec}\,p \to \neg(\forall k{<}n.\ pk) \longleftrightarrow (\exists k{<}n.\ pk)$

Proof (1) is straightforward and holds for all quantifications. This is also true for the direction "$\leftarrow$" of (2). In contrast, the direction "$\rightarrow$" of (2) depends on the decidability of p. If follows with the certifying LWO from Fact 13.2.1 instantiated with $\lambda k.\neg pk$ and exploits the double negation law for p. ■

Finite quantifications over decidable predicates are decided.

Fact 13.3.3 (Deciders for finite quantifications)

1. $\mathsf{dec}\,p \to \mathsf{dec}(\lambda n.\forall k{<}n.\ pk)$
2. $\mathsf{dec}\,p \to \mathsf{dec}(\lambda n.\exists k{<}n.\ pk)$

Proof (1) follows with the decider for the safeness predicate (Fact 13.3.1) instantiated with $\lambda k.\neg pk$ exploiting the double negation law for p.
(2) follows with the certifying LWO from Fact 13.2.1 and the de Morgan law for negated finite existential quantification (Fact 13.3.2(1)). ■

Exercise 13.3.4 Obtain deciders for finite quantification with "$\le$" using the deciders for "$<$" instantiated with $\mathsf{S}n$.

Exercise 13.3.5 (Decidability of Primality)
We define divisibility and primality using finite quantification:

$$\begin{aligned} n \mid x \;&:=\; x \neq 0 \to n \neq 1 \to n \neq x \to \exists k{<}x.\ x = k \cdot n \\ \mathsf{prime}\,x \;&:=\; x \geq 2 \wedge \forall k{<}x.\ k \mid x \to k = 1 \end{aligned}$$

a) Prove that both predicates are decidable.

b) Prove $n \mid x \longleftrightarrow \exists k.\ x = k \cdot n$.

Remark. In Coq we can define a reducible certifying decider for primality computing proofs for $\mathsf{prime}\,101$ and $\neg\mathsf{prime}\,117$, for instance. This approach requires that the underlying proof scripts carefully separate the computational level from the propositional level where automation tactics will insert abstract proof functions. See the accompanying Coq file to learn more.

13.4 Reducible LWOs

The deciders and LWOs obtained so far are not reducible since they are derived from the abstract certifying LWO provided by Fact 13.2.1. We can obtain a reducible

LWO by defining a simply typed recursive function following the construction of the certifying LWO.

We assume a predicate $p^{\mathsf{N}\to\mathbb{P}}$ with a boolean decider $\hat{p}^{\mathsf{N}\to\mathsf{B}}$ such that

$$\forall n.\ \text{IF } \hat{p}n \text{ THEN } pn \text{ ELSE } \neg pn$$

and define an inductive function

$$\begin{aligned} G &: \mathsf{N} \to \mathcal{O}(\mathsf{N}) \\ G\,0 &:= \emptyset \\ G\,(\mathsf{S}n) &:= \text{MATCH } Gn\ [\,{}^{\circ}k \Rightarrow {}^{\circ}k \mid \emptyset \Rightarrow \text{IF } \hat{p}n \text{ THEN } {}^{\circ}n \text{ ELSE } \emptyset\,] \end{aligned}$$

We also define an inductive predicate

$$\begin{aligned} \varphi &: \mathsf{N} \to \mathcal{O}(\mathsf{N}) \to \mathbb{P} \\ \varphi n\,\emptyset &:= \mathsf{safe}\,pn \\ \varphi n\,{}^{\circ}k &:= \mathsf{least}\,pk \wedge k < n \end{aligned}$$

Fact 13.4.1 (Correctness)
$\forall n.\ \varphi n(Gn)$.

Proof By induction on n. In the zero case, the claim reduces to $\mathsf{safe}\,p0$, which holds by Fact 13.1.2(1). In the successor case, we have the inductive hypothesis $\varphi n(Gn)$ and the claim $\varphi(\mathsf{S}n)(G(\mathsf{S}n))$. We discriminate on Gn. If $Gn = {}^{\circ}k$, the inductive hypothesis simplifies to $k < n \wedge \mathsf{least}\,pk$ and the claim to $k < \mathsf{S}n \wedge \mathsf{least}\,pk$. The claim follows.
If $Gn = \emptyset$, the inductive hypothesis is $\mathsf{safe}\,pn$. We discriminate on $\hat{p}n$ in the claim. If pn, the claim reduces to $n < \mathsf{S}n \wedge \mathsf{least}\,pn$ and follows. If $\neg pn$, the claim reduces to $\mathsf{safe}\,p(\mathsf{S}n)$ and follows with Fact 13.1.2(2). ■

Step-indexed linear search

The canonical algorithm for computing least witnesses is linear search: Test p on a *counter* $m = 0, 1, 2, \ldots$ until the first m satisfying p is found. Linear search terminates if and only if p is satisfiable. Realizing linear search with a terminating function in computational type theory requires a modification. The trick is to switch to a *step-indexed* linear search function Lnm testing p starting from m for at most n steps:

$$\begin{aligned} L &: \mathsf{N} \to \mathsf{N} \to \mathcal{O}(\mathsf{N}) \\ L\,0\,m &:= \emptyset \\ L\,(\mathsf{S}n)\,m &:= \text{IF } \hat{p}m \text{ THEN } {}^{\circ}m \text{ ELSE } L\,n\,(\mathsf{S}m) \end{aligned}$$

Note that L tail-recurses on the *step index* n.

We would like to prove $Ln0 = Gn$. To do so, we need to understand the linear search function Lnm for all n and m.

Fact 13.4.2 (Correctness) $\forall nm.\ \mathsf{safe}\, pm \to \varphi(n+m)(Lnm)$.

Proof By induction on n with m quantified. The zero case is trivial. In the successor case we discriminate on $\hat{p}m$. If pm, we have the trivial proof obligation

$$pm \to \mathsf{safe}\, pm \to m < \mathsf{S}(n+m) \wedge \mathsf{least}\, pm$$

If $\neg pm$, we have the proof obligation

$$\neg pm \to \mathsf{safe}\, pm \to \varphi(\mathsf{S}n+m)(Ln(\mathsf{S}m))$$

which reduces to an instance of Fact 13.1.2(2) using the inductive hypothesis for $\mathsf{S}m$ and rewriting with $\mathsf{S}n + m = n + \mathsf{S}m$. ■

Fact 13.4.3 (Correctness) $\forall n.\ \varphi n(Ln0)$.

Proof Fact 13.4.2 instantiated with $m = 0$. ■

Fact 13.4.4 (Agreement) $Ln0 = Gn$.

Proof Given the correctness theorems for G and L (Facts 13.4.1 and 13.4.3), the claim follows with the uniqueness of φn which follows with the uniqueness of $\mathsf{least}\, p$ after discrimination on the option arguments. ■

The proofs we have just seen for L are typical for program verifications. While the underlying ideas are clear, the detailed execution of the proofs is tedious and calls for the help of a proof assistant.

Exercise 13.4.5 (Invariant Puzzle) We have one more reducible LWO in the offer. Consider the function

$$\begin{aligned} &W : \mathsf{N} \to \mathsf{N} \\ &W\,0 := 0 \\ &W\,(\mathsf{S}n) := \text{LET } k = Wn \text{ IN IF } \hat{p}k \text{ THEN } k \text{ ELSE } \mathsf{S}n \end{aligned}$$

Verify that $Fn := \text{IF } n \le Wn \text{ THEN } \emptyset \text{ ELSE } {}^\circ Wn$ agrees with G.

Hint: The worker function W computes the largest $k \le n$ that is safe for p.

Hint: You need a specification δ for W such that $\forall n.\ \delta n(Wn)$. For the correctness proof to go through, δ must be an invariant for W's recursion, which imposes the proof obligation $\forall nk.\ \delta nk \to \text{IF } \hat{p}n \text{ THEN } \delta(\mathsf{S}n)k \text{ ELSE } \delta(\mathsf{S}n)(\mathsf{S}n)$.

Exercise 13.4.6 (Reducible deciders for finite quantifications)

a) Obtain the deciders for finite quantifications as reducible functions using G or L.

b) Verify the correctness of your deciders.

c) Prove prime 101 in the proof assistant using a boolean decider for primality.

13.5 Least Witness Existence and Excluded Middle

It turns out that the existence of least witnesses for satisfiable predicates on numbers is equivalent to the law of excluded middle:

$$\begin{aligned}\mathsf{ELW} &:= \forall p^{\mathsf{N}\to\mathbb{P}}.\ \mathsf{ex}\,p \to \mathsf{ex}(\mathsf{least}\,p)\\ \mathsf{XM} &:= \forall P^{\mathbb{P}}.\ P \lor \neg P\end{aligned}$$

The direction $\mathsf{XM} \to \mathsf{ELW}$ is unsurprising since XM gives us decidability of predicates at the propositional level, which means that we can use the construction for LWOs if we lower them to the propositional level.

Fact 13.5.1 $\mathsf{XM} \to \mathsf{ELW}$.

Proof Assume XM. Then every predicate is logically decidable: $\forall n.\ pn \lor \neg pn$. Hence we can carry out the constructions of Facts 13.2.1 and 13.2.4 at the propositional level:

$$\begin{aligned}&\forall p^{\mathsf{N}\to\mathbb{P}}\ \forall n.\ (\exists k < n.\ \mathsf{least}\,pk) \lor \mathsf{safe}\,pn\\ &\forall p^{\mathsf{N}\to\mathbb{P}}.\ \mathsf{ex}\,p \to \mathsf{ex}\,(\mathsf{least}\,p)\end{aligned}$$

∎

The other direction $\mathsf{ELW} \to \mathsf{XM}$ is very easy to prove.

Fact 13.5.2 $\mathsf{ELW} \to \mathsf{XM}$.

Proof We pick a proposition P and prove $P \lor \neg P$. Using the assumption, we discriminate on the least witness n of the satisfiable predicate $pn := \textsc{if}\ n\ \textsc{then}\ P\ \textsc{else}\ \top$. We have $p0 \longleftrightarrow P$. Thus if the least witness is 0, we have P, and otherwise $\neg P$. ∎

Exercise 13.5.3 (Boolean least witness existence)
Prove $\mathsf{XM} \longleftrightarrow \forall p^{\mathsf{N}\to\mathsf{B}}.\ \mathsf{ex}\,p \to \exists x.\ px \land (p\mathsf{true} \to x = \mathsf{true})$.

Exercise 13.5.4 Prove that the following propositions are equivalent.

1. XM
2. $\forall pn.\ \mathsf{ex}(\mathsf{least}\,p) \lor \mathsf{safe}\,pn$
3. $\forall p.\ \mathsf{ex}\,p \to \mathsf{ex}(\mathsf{least}\,p)$.

14 EWOs

In computational type theory, functional recursion comes as structural recursion based on recursive inductive types. A canonical recursive type is the type of numbers. In this chapter we will consider a recursive type constructor

$$T(p:\mathsf{N}\to\mathbb{P})(n:\mathsf{N}):\mathbb{P} \;::=\; C\,(\neg pn \to Tp(\mathsf{S}n))$$

featuring a higher-order structural recursion through the right-hand side of a function type serving as argument type of a single proof constructor C. We will see that with the inductive predicate T we can construct an *existential witness operator*

$$\forall p^{\mathsf{N}\to\mathbb{P}}.\ \mathsf{dec}\,p \to \mathsf{ex}\,p \to \mathsf{sig}\,p$$

testing p on $n = 0, 1, 2, \dots$ until a number satisfying p is found. For this construction the *higher-order recursion* of T and the fact that T is a computational predicate (no PDR) are essential. Recall that a naive witness operator just unpacking the given existential witness is not possible since it would violate the PDR.

EWOs (existential witness operators) exist for many computational types, not just the type of numbers. In particular, finite types and types embedding into the numbers do have EWOs. Moreover, EWOs transport through option types. With EWOs we can construct inverses for bijective functions and co-inverses for surjective functions.

14.1 Linear Search Types

We start with the definition of an inductive predicate[1]

$$T(p:\mathsf{N}\to\mathbb{P})(n:\mathsf{N}):\mathbb{P} \;::=\; C\,(\neg pn \to Tp(\mathsf{S}n))$$

The first point to notice about the definition of T is that there is recursion through the right-hand side of the function type serving as argument type of the single proof constructor C. Such **higher-order structural recursions** are legal in computational type theory. The second point to notice about the definition of T is that T is a computational predicate (§ 5.2) since the type $\neg pn \to Tp(\mathsf{S}n)$ of the proper argument

[1] The letter T derives from the intuition that propositions obtained with the computational T provide for a transfer from the propositional level to the computational level.

of the proof constructor C is propositional. This means that inductive propositions obtained with T are exempted from the discrimination restriction (§ 5.2).

There is a third aspect of the inductive type definition of T deserving discussion: The parameter n of Tpn is *nonuniform* in that it is changed to $\mathsf{S}n$ in the recursive application of T. We remark that nonuniform parameters are fine in computational type theory, and in fact are needed for meaningful higher-order structural recursion.

Recall that proofs of computational propositions can be decomposed at the computational level although they have been constructed at the propositional level. Recursive computational propositions thus provide for computational recursion.

The next essential step is the definition of an eliminator for T making use of the higher-order recursion provided by T:

$$\begin{aligned} &E : \forall p^{\mathsf{N}\to\mathbb{P}}\ \forall q^{\mathsf{N}\to\mathbb{T}}.\ (\forall n.\ (\neg pn \to q(\mathsf{S}n)) \to qn) \to \forall n.\ Tpn \to qn \\ &E\,pqe\ n\,(C\varphi)\ :=\ e\,n\,(\lambda a.\ E\,pqe\ (\mathsf{S}n)\,(\varphi a)) \end{aligned}$$

Note that the discrimination of the defining equation is a propositional transfer discrimination. It is admissible since T is computational. As it comes to the guard condition for termination, the argument is that every application of the *continuation function* φ (appearing as proper argument of the constructor C in the pattern of the defining equation) qualifies for (higher-order) structural recursion.

Why does this liberal guard condition for higher-order structural recursion preserve termination of reduction in computational type theory? We can offer two answers to this question:

1. Computational type theory is designed such that reduction terminates in the presence of higher-order recursion.
2. Before we can construct a value $C\varphi$, we need to construct the function φ. Hence all values φa are in some sense smaller than $C\varphi$.

We remark that the eliminator E is the only inductive function we will consider for T. So, all the magic of higher-order structural recursion is in the definitions of T and E.

We will refer to the propositions Tpn as **linear search types**.

14.2 EWO for Numbers

We will construct an EWO for N based on an abstract interface that we realize with the inductive predicate T. The interface provides an abstract computational predicate T' with two constructors.

Fact 14.2.1 (First-order interface)
For every predicate $p^{\mathsf{N}\to\mathbb{P}}$ there are functions as follows:

$$\begin{aligned}
T' &: \mathsf{N} \to \mathbb{P} \\
I &: \forall n.\ pn \to T'n \\
D &: \forall n.\ T'(\mathsf{S}n) \to T'n \\
E' &: \forall q^{\mathsf{N}\to\mathbb{T}}.\ \mathsf{dec}\,p \to (\forall n.\ pn \to qn) \to (\forall n.\ q(\mathsf{S}n) \to qn) \to (\forall n.\ T'n \to qn)
\end{aligned}$$

We refer to I and D as **abstract constructors** and to E' as **abstract eliminator**.

Proof Assume $p^{\mathsf{N}\to\mathbb{P}}$. We set $T' := Tp$. Now the abstract constructors I and D are easily obtained from the concrete constructor C. The abstract eliminator E' can be obtained with the concrete eliminator E. Having a decider for p is essential to separate the base case pn from the recursive case $\neg pn$. ■

Note that the abstract constructors I and D and the abstract eliminator E' make the predicate T' appear as a computational inductive predicate obtained with two first-order constructors. A proof of $T'pn$ verifies that there is $k \geq n$ such that pk.

Fact 14.2.2 (EWO for numbers)
There is a function $\forall p^{\mathsf{N}\to\mathbb{P}}.\ \mathsf{dec}\,p \to \mathsf{ex}\,p \to \mathsf{sig}\,p$.

Proof We assume $d : \mathsf{dec}\,p$, $H : \mathsf{ex}\,p$, and the functions from Fact 14.2.1. We obtain $\mathsf{sig}\,p$ with E' instantiated with $qn := \mathsf{sig}\,p$, d, and $n := 0$. This results in three proof obligations:

1. $\forall n.\ pn \to \mathsf{sig}p$
2. $\forall n.\ \mathsf{sig}p \to \mathsf{sig}p$
3. $T'0$.

Obligations (1) and (2) are trivial. For (3) we note that $T'0$ is a proposition. Thus we can unpack H and obtain pn and $T'n$ using I. We now close the proof by proving $\forall n.\ T'n \to T'0$ by induction on n using D. ■

Exercise 14.2.3 (Computational characterization of linear search predicate)
Assume a predicate $p^{\mathsf{N}\to\mathbb{P}}$ and prove the following:

a) $m \leq n \to Tpn \to Tpm$

b) $\mathsf{dec}\,p \to Tpn \to \Sigma m.\ m \geq n \wedge pm$

c) $\mathsf{dec}\,p \to (Tpn \Leftrightarrow \Sigma m.\ m \geq n \wedge pm)$

Exercise 14.2.4 (Empty search type) 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 14.2.5 With higher-order recursion we can define an empty propositional type allowing for computational elimination as follows:

$$V : \mathbb{P} \ ::= \ C(\top \to V)$$

Define an elimination function $V \to \forall X^{\mathbb{T}}. X$.

Exercise 14.2.6 Let p be a decidable predicate on numbers. Construct a function $\forall n.\, Tpn \to \Sigma k.\, k \geq n \wedge pk$.

Exercise 14.2.7 (Strict positivity condition)
We remark that computational type theory admits recursion only through the right-hand side of function types, a restriction known as **strict positivity condition**.

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 assuming that the illegal definition of B provides constants as follows:

$$\begin{aligned} B &: \mathbb{T} \\ C &: (B \to \bot) \to B \\ E &: \forall Z.\, B \to ((B \to \bot) \to Z) \to B \to Z \end{aligned}$$

First define a function $f : B \to \bot$ using the constant E.

14.3 General EWOs

We define the type of EWOs for a type X as follows:

$$\mathsf{EWO}\, X^{\mathbb{T}} \ := \ \forall p^{X \to \mathbb{P}}.\, \mathsf{dec}\, p \to \mathsf{ex}\, p \to \mathsf{sig}\, p$$

Fact 14.3.1 The types $\bot$, $\mathbb{1}$, B, and N have EWOs.

Proof The EWO for N was obtained with Fact 14.2.2. For the other three types 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 14.3.2 (Disjunctive EWOs)
Let p and q be decidable predicates on a type X 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 14.3.3 (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 14.3.4 (Numeral types have EWOs)
The numeral types $\mathcal{O}^n\bot$ have EWOs.

Proof By induction on n using Facts 14.3.1 and 14.3.3. ∎

It turns out that injections transport EWOs backwards.

Fact 14.3.5 (Injections transport EWOs)
$\mathcal{I}XY \to \mathsf{EWO}\,Y \to \mathsf{EWO}\,X$.

Proof Let $\mathsf{inv}_{XY}fg$. 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 14.3.6 Every type that embeds into N has an EWO: $\mathcal{I}X\mathsf{N} \to \mathsf{EWO}\,X$.

Proof Facts 14.3.5 and 14.2.2. ∎

Fact 14.3.7 $\mathsf{N}\times\mathsf{N}$ has an EWO.

Proof Follows with Fact 14.3.6 since $\mathsf{N}\times\mathsf{N}$ and N are in bijection (Fact 11.1.8). ∎

Fact 14.3.8 (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 14.3.9 (EWO for boolean deciders)
We will consider EWOs for boolean deciders on numers:

$$\forall f^{\mathsf{N}\to\mathsf{B}}.\ (\exists n.\ fn = \mathsf{true}) \to (\Sigma n.\ fn = \mathsf{true})$$

a) Define an EWO for boolean deciders (i) using the EWO for numbers, and (ii) from scratch using customized linear search types.
b) Using an EWO for boolean deciders, define an EWO for decidable predicates on numbers.
c) Using an EWO for boolean deciders, define 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

$$\mathsf{enum}\ p^{X\to\mathbb{P}}\ f^{\mathsf{N}\to\mathcal{O}(X)}\ :=\ \forall x.\ px \longleftrightarrow \exists n.\ fn = {}^{\circ}x$$

14.4 EWO Applications

Fact 14.4.1 (Co-inverse for surjective functions)
Let $f^{X\to Y}$ be a surjective function. Then there is a function
$\mathsf{EWO}\ X \to \mathcal{E}\,Y \to \Sigma g.\ \mathsf{inv}\,fg$ yielding a co-inverse function for f.

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 14.4.2 (Inverse for bijective functions)
Let $f^{X\to Y}$ be a bijective function. Then there is a function
$\mathsf{EWO}\ X \to \mathcal{E}\,Y \to \Sigma g.\ \mathsf{inv}\,gf \wedge \mathsf{inv}\,fg$ yielding an inverse function for f.

Proof Fact 14.4.1 gives us g with $\mathsf{inv}\,fg$. Now $\mathsf{inv}\,gf$ follows since f is injective (Fact 11.1.1). ∎

The following fact was discovered by Andrej Dudenhefner in March 2020.

Fact 14.4.3 (Discreteness via step-indexed boolean equality decider)
Let $f^{X\to X\to\mathsf{N}\to\mathsf{B}}$ be a function such that $\forall xy.\ x = y \longleftrightarrow \exists n.\ fxyn = \mathsf{true}$.
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 $fxxn = \mathsf{true}$. If $fxyn = \mathsf{true}$, we have $x = y$. If $fxyn = \mathsf{false}$, we have $x \neq y$. ∎

Exercise 14.4.4 (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 14.4.5 Let $f : \mathsf{N} \to \mathsf{B}$. Prove the following:

a) $(\exists n.\ fn = \mathsf{true}) \Leftrightarrow (\Sigma n.\ fn = \mathsf{true})$.

b) $(\exists n.\ fn = \mathsf{false}) \Leftrightarrow (\Sigma n.\ fn = \mathsf{false})$.

14.5 Notes

With linear search types we have seen an inductive predicate going beyond of the inductive type 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.

15 Indexed Inductives

Indexed inductive type definitions are generalized inductive type definitions where the target types of value constructors may instantiate nonparametric arguments (called indices) of their type constructors. Indexed inductive type definitions provide for the formalization of derivation systems as they appear in the study of proof systems, computational systems, and programming languages.

We discuss indexed inductives at the example of two derivation systems, one for the reflexive transitive closure of relations, and one for arithmetic comparisons. We also model numeral types and vector types as indexed inductive types and show that the inductive formalizations are in bijection with non-inductive formalizations employing arithmetic recursion.

It turns out that the constants for equality can be defined as indexed inductives, and that the inductive definition adds properties to the Leibniz characterization of equality that are of foundational interest.

15.1 Inductive Equality

We start with inductive equality since its indexed inductive definition is particularly simple. Recall that we have accommodated equality with three constants (§ 4.2):

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

We can obtain eq and Q with an **indexed inductive definition**:

$$\begin{aligned} &\mathsf{eq}\,(X:\mathbb{T},\ x:X) :\ X \to \mathbb{P}\ ::= \\ &\mid \mathsf{Q} :\ \mathsf{eq}\ X x x \end{aligned}$$

Note that the target type of the constructor Q is $\mathsf{eq}\, X\, x\, x$ and not $\mathsf{eq}\, X\, x\, y$. This means that only X and x are parametric arguments of $\mathsf{eq}\, X\, x\, y$, and that y is a nonparametric argument. Following common speak we will refer to nonparametric arguments of type constructors as **indices**. The extra freedom coming with indexical arguments of type constructors is that they can be freely instantiated in the target types of value constructors.

We can now obtain the rewriting constant R with an inductive function definition:

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

The definition discriminates on the inductive argument of type $\mathsf{eq}\,Xxy$. Note that the value constructor Q equates the variables x and y in the defining equation. Also note that the pattern of the defining equation of R gives the argument y with an underline. The accounts for the fact that the argument y of R is determined by the discrimination with Q.

We have defined R with a general type function $p^{X\to\mathbb{T}}$ rather than just a predicate $p^{X\to\mathbb{P}}$. This is possible since the inductive predicate eq is computational since the value Q has no proper argument.

There is a single but important restriction on the types of inductive functions discriminating on arguments of indexed inductive types: The indexed argument type must be given with variables in the index positions that don't occur otherwise in the argument type. We speak of the **index condition**. The index condition disallows an inductive function discriminating on an argument of type $\mathsf{eq}\,Xxx$, for instance.

Inductive equality has a number of properties Leibniz equality doesn't have, an issue we will study in Chapter 29.

15.2 Reflexive Transitive Closure

We may see a relation $R^{X\to X\to\mathbb{P}}$ as a description of a graph with vertices in X and edges Rxy. We will write R^*xy if there is a path $x \to \cdots \to y$ from x to y in the graph described by R. We may characterize R^*xy with two *derivation rules*:

$$\frac{}{R^*xx} \qquad\qquad \frac{Rxy \qquad R^*yz}{R^*xz}$$

The first rule says that for every $x : X$ there is a path R^*xx. The second rule says that there is a path R^*xz if there is an edge Rxy and a path R^*yz.

We formalize the notation R^*xy and the two derivation rules with an indexed inductive predicate star defined as follows:

$$\begin{aligned}&\mathsf{star}\,(X:\mathbb{T},\ R : X\to X\to\mathbb{P},\ x:X) :\ X\to\mathbb{P}\ ::=\\ &\mid \mathsf{Nil} : \mathsf{star}\,XRxx\\ &\mid \mathsf{Cons} : \forall yz.\ Rxy \to \mathsf{star}\,XRyz \to \mathsf{star}\,XRxz\end{aligned}$$

We now write R^* for the relation $\mathsf{star}\,XR$. Note that a proof of R^*xy with the constructors Nil and Cons describes a path from x to y. We may see an inductive proposition R^*xy as the type of all paths from x to y. We remark that the inductive formalization of R^* is minimal in that it doesn't presuppose numbers or lists.

Considering the inductive predicate $\mathsf{star}\,XRxy$ from a formal point of view, the arguments X and R are uniform parameters, x is a nonuniform parameter, and y is an index. That y is an index is forced by the type of the constructor Nil.

Let p and p' be predicates $X \to X \to \mathbb{P}$. We define **inclusion of predicates** as one would expect:

$$p \subseteq p' \quad := \quad \forall xy.\, pxy \to p'xy$$

The relation R^* is known as the reflexive transitive closure of the relation R. This speak is justified since every reflexive and transitive relation that contains R also contains R^*. We are going to prove this fact.

Fact 15.2.1 (Inclusion) $R \subseteq R^*$.

Proof Let Rxy. We show R^*xy. We have R^*yy with Nil. Thus R^*xy with Cons. ■

We would like to prove that the relation R^* is transitive; that is, if there are paths R^*xy and R^*yz, then there is a path R^*xz. We prove this fact by induction on the first path R^*xy.

Fact 15.2.2 (Transitivity) $R^*xy \to R^*yz \to R^*xz$.

Proof By induction on the path R^*xy. If R^*xy is obtained with Nil, we have $x = y$ and the claim is trivial. If R^*xy is obtained with Cons, we have an edge Rxx' and a shorter path $R^*x'y$. The inductive hypothesis gives us a path $R^*x'z$. Now we obtain a path R^*xz with Cons. ■

To formalize the path induction in the transitivity proof, we need an eliminator function for the inductive predicate star. An eliminator function with the type

$$\begin{aligned}\mathsf{E} \;:\; & \forall X\, \forall R\, \forall p^{X \to X \to \mathbb{P}}.\\ & (\forall x.\, pxx) \to \\ & (\forall xyz.\, Rxy \to pyz \to pxz) \to \\ & \forall xy.\, R^*xy \to pxy\end{aligned}$$

will suffice. For the transitivity proof E can be used with the target predicate

$$pxy \quad := \quad R^*yz \to R^*xz$$

assuming that z is introduced outside. The formalization of the proof is now straightforward. While checking the details with paper and pencil is tedious, constructing the formal proof with the proof assistant is pleasant. See the accompanying Coq file.

We define the eliminator function E following its type

$$\begin{aligned}\mathsf{E} \;:\; & \forall X\, \forall R\, \forall p^{X\to X\to\mathbb{P}}.\\ & (\forall x.\, pxx) \to\\ & (\forall xyz.\, Rxy \to pyz \to pxz) \to\\ & \forall xy.\, R^*xy \to pxy\end{aligned}$$

as an inductive function discriminating on paths R^*xy:

$$\begin{aligned}\mathsf{E}\, XRp\, e_1e_2\, x\, \_\, \mathsf{Nil} \;&:=\; e_1\\ \mathsf{E}\, XRp\, e_1e_2\, x\, \_\, (\mathsf{Cons}\, x'zra) \;&:=\; e_2xx'zr(\mathsf{E}\, XRp\, e_1e_2\, x'za)\end{aligned}$$

The equation for Cons may look complicated but it is just a routine construction following the types of E and Cons.

Fact 15.2.3 (Reflexive transitive closure)
Let $p^{X\to X\to\mathbb{P}}$ be a reflexive and transitive relation. Then $R \subseteq p \to R^* \subseteq p$.

Proof We prove $\forall xy.\, R^*xy \to pxy$ by induction on the path R^*xy. If R^*xy is obtained with Nil, we have $x = y$ and the claim holds since p is reflexive. If R^*xy is obtained with Cons, we have an edge Rxx' and a shorter path $R^*x'y$. Now pxx' since $R \subseteq p$ and $px'y$ by the inductive hypothesis. Hence pxy since p is transitive. ∎

Exercise 15.2.4 Prove $R^*(R^*) \subseteq R^*$.

Exercise 15.2.5 (Definition with arithmetic recursion)
There is an elegant definition of the relation R^* as a recursive predicate:

$$\begin{aligned}&\mathsf{path} :\; \forall X.\, (X \to X \to \mathbb{P}) \to \mathsf{N} \to X \to X \to \mathbb{P}\\ &\mathsf{path}\, XR0xy \;:=\; (x = y)\\ &\mathsf{path}\, XR(\mathsf{S}n)xy \;:=\; (\exists x'.\, Rxx' \wedge \mathsf{path}\, XRnxy)\end{aligned}$$

Prove $\mathsf{star}\, XRxy \longleftrightarrow \exists n.\, \mathsf{path}\, XRnxy$.

Exercise 15.2.6 Here is another inductive definition of reflexive transitive closure:

$$\begin{aligned}&\mathsf{star}'\,(X : \mathbb{T},\; R : X \to X \to \mathbb{P},\; x : X) :\; X \to \mathbb{P} \;::=\\ &\mid \mathsf{Incl} : \forall xy.\, Rxy \to \mathsf{star}'\, XRxy\\ &\mid \mathsf{Refl} : \mathsf{star}'\, XRxx\\ &\mid \mathsf{Trans} : \forall xyz.\, \to \mathsf{star}'\, XRxy \to \mathsf{star}'\, XRyz \to \mathsf{star}'\, XRxz\end{aligned}$$

Prove $\mathsf{star}\, XRxy \longleftrightarrow \mathsf{star}'\, XRxy$.

15.3 Inductive Comparisons

Two derivation rules for arithmetic comparisons $x \le y$ are

$$\frac{}{x \le x} \qquad\qquad \frac{x \le y}{x \le \mathsf{S}y}$$

The rules are obviously correct. We will show that they are also complete in the sense that they can derive every valid comparison.

We formalize the derivation system comprised by the two rules with an inductive type definition:

$$\begin{aligned}
&\mathsf{le}\,(x : \mathsf{N}) : \ \mathsf{N} \to \mathbb{P} \ ::= \\
&\mid \mathsf{leE} : \mathsf{le}\, x x \\
&\mid \mathsf{leS} : \forall y.\, \mathsf{le}\, x y \to \mathsf{le}\, x\, (\mathsf{S}y)
\end{aligned}$$

Note that in $\mathsf{le}\,xy$ the argument x is a uniform parameter and the argument y is an index.

Completeness of the derivation system le can now be shown with induction on numbers.

Fact 15.3.1 (Completeness) $x \le y \to \mathsf{le}\,xy$.

Proof By induction on y.
Let $y = 0$. The $x = 0$ and hence $\mathsf{le}\,xy$ with leE.
In the successor case we assume $x \le \mathsf{S}y$ and show $\mathsf{le}\,x(\mathsf{S}y)$. If $x = \mathsf{S}y$, the claim follows with leE. If $x \le y$, the inductive hypothesis gives us $\mathsf{le}\,xy$. The claim follows with leS. ■

The other direction is intuitively obvious since both derivation rules represent valid facts about arithmetic comparisons. To do the proof formally, we need induction on derivations of comparisons $\mathsf{le}\,xy$, which can be provided with an inductive function

$$\begin{aligned}
\mathsf{E} :\ & \forall x\, \forall p^{\mathsf{N}\to\mathbb{P}}. \\
& pxx \to \\
& (\forall y.\, py \to p(\mathsf{S}y)) \to \\
& \forall y.\, x \le y \to py
\end{aligned}$$

$$\begin{aligned}
\mathsf{E}\,xpe_1e_2\,\_\,\mathsf{leE} &:= e_1 \\
\mathsf{E}\,xpe_1e_2\,\_\,(\mathsf{leS}\,ya) &:= e_2y(\mathsf{E}\,xpe_1e_2ya)
\end{aligned}$$

Fact 15.3.2 (Soundness) $\mathsf{le}\,xy \to x \le y$.

Proof By induction on the derivation of $\mathsf{le}\,xy$. If $\mathsf{le}\,xy$ is obtained with leE, we have $x = y$ and hence $x \le y$. If $\mathsf{le}\,xy$ is obtained with leS, we have a smaller derivation $\mathsf{le}\,xy'$ and $y = Sy'$. The inductive hypothesis gives us $x \le y'$, which give us the claim $x \le \mathsf{S}y'$. ■

Exercise 15.3.3 Give two derivation rules for comparisons $x < y$ and show their soundness and completeness.

15.4 Inductive Numeral Types

Numeral types are a family of finite types providing a canonical finite type for every cardinality. In §11.3, we obtained numeral types recursively as $\mathcal{O}^n\bot$. Numeral types may also be obtained as indexed inductive types:

$$\begin{aligned}
&\mathsf{fin} : \mathsf{N} \to \mathbb{T} ::= \\
&\mid \mathsf{Old} : \forall n.\ \mathsf{fin}\,n \to \mathsf{fin}\,(\mathsf{S}n) \\
&\mid \mathsf{New} : \forall n.\ \mathsf{fin}\,(\mathsf{S}n)
\end{aligned}$$

the constructors Old and New are in correspondence with the option type constructor some and none. The definition suggests that fin_0 is empty and that $\mathsf{fin}_{\mathsf{S}n}$ has one more element than fin_n. We prove this fact by establishing a bijection between fin_n and N_n:

$$\begin{aligned}
f &: \forall n.\ \mathsf{fin}_n \to \mathsf{N}_n \\
f\,\_\,(\mathsf{Old}\,na) &:= {}^\circ fna \\
f\,\_\,(\mathsf{New}\,n) &:= \emptyset
\end{aligned}
\qquad\qquad
\begin{aligned}
g &: \forall n.\ \mathsf{N}_n \to \mathsf{fin}_n \\
g\,0\,a &:= \mathsf{E}_\bot \mathsf{fin}_0\ a \\
g\,(\mathsf{S}n)\,{}^\circ a &:= \mathsf{Old}\,n(gna) \\
g\,(\mathsf{S}n)\,\emptyset &:= \mathsf{New}\,n
\end{aligned}$$

The definitions of the inductive functions f and g clarify how the numeral types fin_n and N_n relate to each other. While the definition of f is straightforward, the definition of g is quite involved. In the zero case void elimination $\mathsf{E}_\bot$ is needed. In the successor case a secondary discrimination on options is needed. Note that in the patterns of the equations defining f the first argument n is given as an underline, as is required by the fact that n provides the index argument of fin_n in the type of f.

Fact 15.4.1 $gn(fna) = a$.

Proof By induction on $a : \mathsf{fin}_n$. For the constructor Old we have the proof obligation $g(\mathsf{S}n)(f(\mathsf{S}n)(\mathsf{Old}\,na)) = \mathsf{Old}\,na$, which simplifies to $\mathsf{Old}\,n(gn(fna)) = \mathsf{Old}\,na$, which holds by the inductive hypothesis $gn(fna) = a$. For the constructor New we have the proof obligation $g(\mathsf{S}n)(f(\mathsf{S}n)(\mathsf{New}\,n)) = \mathsf{New}\,n$, which simplifies to $\mathsf{New}\,n = \mathsf{New}\,n$. ∎

The proof does an induction on values of the inductive type fin_n. The induction can be formalized with an eliminator function defined as follows:

$$\begin{aligned}
\mathsf{E} :\ & \forall p^{\forall n.\, \mathsf{fin}_n \to \mathbb{T}}.\\
& (\forall na.\ pna \to p(\mathsf{S}n)(\mathsf{Old}\,na)) \to\\
& (\forall n.\ p(\mathsf{S}n)(\mathsf{New}\,n)) \to\\
& \forall na.\ pna
\end{aligned}$$

$$\begin{aligned}
\mathsf{E}\,pe_1e_2\,\_\,(\mathsf{Old}\,na) &:= e_1na(\mathsf{E}\,pe_1e_2\,na)\\
\mathsf{E}\,pe_1e_2\,\_\,(\mathsf{New}\,n) &:= e_2n
\end{aligned}$$

Fact 15.4.2 $fn(gna) = a$.

Proof By induction on n^{N} and discrimination on a^{N_n}. The proof obligation $f0(g0a) = a$ is trivial since $a : \bot$. The proof obligation $f(\mathsf{S}n)(g(\mathsf{S}n)(^\circ a)) = {^\circ a}$ reduces to $^\circ fn(gna) = {^\circ a}$ and follows with the inductive hypothesis. The proof obligation $f(\mathsf{S}n)(g(\mathsf{S}n)\emptyset) = \emptyset$ reduces to $\emptyset = \emptyset$. ∎

Corollary 15.4.3 The types fin_n and N_n and are in bijection.

Exercise 15.4.4 Use the bijection between inductive and recursive numeral types to show that the inductive numeral types fin_n have equality deciders and EWOs.

Exercise 15.4.5 We want to prove that the inductive numeral type fin_0 is empty. Formally, we represent this statement with the proposition $\mathsf{fin}_0 \to \bot$.

a) Prove $\mathsf{fin}_0 \to \bot$ using the eliminator for inductive numeral types.

b) Prove $\mathsf{fin}_0 \to \bot$ using the bijection between inductive and recursive numeral types.

Hint: For (a) prove the equivalent proposition $\forall n.\ \mathsf{fin}_n \to n \neq 0$.

15.5 Inductive Vector Types

Vector types as introduced in §11.4 have an elegant definition as indexed inductive types:

$$\begin{aligned}&\mathsf{vec}\,(X:\mathbb{T}):\ \mathsf{N}\to\mathbb{T}\ ::=\\ &\mid \mathsf{Nil}:\ \mathsf{vec}\,X\,0\\ &\mid \mathsf{Cons}:\ \forall n.\,X\to\mathsf{vec}\,X n\to\mathsf{vec}\,X(\mathsf{S}n)\end{aligned}$$

The constructors Nil and Cons give us the functions nil and cons we defined for the recursive representation with products and unit.

We define a bijection between inductive and recursive vector types. For simplicity we don't write the base type X.

$$\begin{aligned}f:\ &\forall n.\,\mathsf{vec}_n\to\mathsf{V}_n & g:\ &\forall n.\,\mathsf{V}_n\to\mathsf{vec}_n\\ f\,\_\,\mathsf{Nil}\ &:=\ \mathsf{I} & g\,0\,a\ &:=\ \mathsf{Nil}\\ f\,\_\,(\mathsf{Cons}\,nxa)\ &:=\ (x,fna) & g\,(\mathsf{S}n)\,(x,a)\ &:=\ \mathsf{Cons}\,nx(gna)\end{aligned}$$

The definitions demonstrate that the structure of inductive and recursive vector types is in full agreement.

Fact 15.5.1 $gn(fna) = a$.

Proof By induction on $a : \mathsf{vec}_n$.

The proof obligation $g0(f0\mathsf{Nil}) = \mathsf{Nil}$ simplifies to $\mathsf{Nil} = \mathsf{Nil}$.

The proof obligation $g(\mathsf{S}n)(f(\mathsf{S}n)(\mathsf{Cons}\,nxa)) = \mathsf{Cons}\,nxa$ simplifies to the equation $\mathsf{Cons}\,nx(gn(fna)) = \mathsf{Cons}\,nxa$, which follows with the inductive hypothesis. ∎

The proof does an induction on values of inductive vector types vec_n. The induction can be formalized with an eliminator function defined as follows:

$$\begin{aligned}\mathsf{E}:\ &\forall p^{\forall n.\,\mathsf{vec}_n\to\mathbb{T}}.\\ &p\,0\,\mathsf{Nil}\to\\ &(\forall nxa.\,pna\to p(\mathsf{S}n)(\mathsf{Cons}\,nxa))\to\\ &\forall na.\,pna\end{aligned}$$

$$\begin{aligned}\mathsf{E}\,pe_1e_2\,\_\,\mathsf{Nil}\ &:=\ e_1\\ \mathsf{E}\,pe_1e_2\,\_\,(\mathsf{Cons}\,nxa)\ &:=\ e_2nxa(\mathsf{E}\,pe_1e_2\,na)\end{aligned}$$

Fact 15.5.2 $fn(gna) = a$.

Proof By induction on n^{N} and discrimination on a^{V_n}. The proof obligation $f0(g\,0\,\mathsf{I}) = \mathsf{I}$ reduces to $\mathsf{I} = \mathsf{I}$. The proof obligation $f(\mathsf{S}n)(g(\mathsf{S}n)(x,a)) = (x,a)$ reduces to $(x,fn(gna)) = (x,a)$ and follows with the inductive hypothesis. ∎

Corollary 15.5.3 The types $\mathsf{vec}_n X$ and $\mathsf{V}_n X$ are in bijection.

Exercise 15.5.4 Use the bijection between inductive and recursive vector types to show that inductive vector types have equality deciders and EWOs.

15.6 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 of cards in C (possibly containing duplicates) 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

$$\forall C.\ \mathcal{D}(\exists A.\ \mathsf{post}\,CAA) \tag{15.1}$$

Since Coq's type theory can only define computable functions, we can conclude that no function of type (15.1) is definable.

As it comes to the arguments of the inductive predicate $\mathsf{post}\,CAB$, the defining rules establish C as a uniform parameter and A and B as indices.

15.7 Index Elimination

Inductive type definitions with indices can be translated into inductive type definitions without indices using equations in the premises. For instance, the initial rule for reflexive transitive closure may be written as

$$\frac{x = y}{R^*xy} \qquad\qquad \frac{Rxx' \qquad R^*x'y}{R^*xy}$$

which yields an equivalent inductive predicate

$$\begin{aligned}&\mathsf{star}\,(X : \mathbb{T},\ R : X \to X \to \mathbb{P},\ x : X,\ y : X) : \mathbb{P}\ ::=\\ &\mid \mathsf{Nil} :\ x = y \to \mathsf{star}\,XRxy\\ &\mid \mathsf{Cons} :\ \forall x'.\ Rxx' \to \mathsf{star}\,XRx'y \to \mathsf{star}\,XRxy\end{aligned}$$

with no index and a single nonuniform parameter x.

Index elimination will not work for inductive equality. Here the best we can do is to express the coreference in the target type with Leibniz equality:

$$\frac{\forall p^{X\to\mathbb{P}}.\ px \to py}{\mathsf{eq}\,Xxy}$$

Formally, this yields the inductive predicate definition

$$\begin{aligned}&\mathsf{eq}\,(X : \mathbb{T},\ x : X,\ y : X) : \mathbb{P}\ ::=\\ &\mid \mathsf{L} :\ (\forall p^{X\to\mathbb{P}}.\ px \to py) \to \mathsf{eq}\,Xxy\end{aligned}$$

Based on this definition we can define

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

What we cannot do, however, is define R with a type function $p^{X\to\mathbb{T}}$. The only way to obtain rewriting of types is to work with the indexed inductive equality predicate.

We remark that rewriting of types is not needed for the usual things we do in computational type theory. We may thus say that indexed inductive types are a convenience not essential for our purposes. The benefits of inductive equality only show at an internal technical level we will explore in Chapter 29.

15.8 Notes

The index condition is a restriction on the types of inductive function discriminating on values of indexed inductive types. It simply says that the index positions of an discriminating argument type must be given with variables not appearing otherwise in the argument type.

Recall that the proof assistant Coq simulates inductive functions as plain functions discriminating with native matches. There are situations called *inversions* in the literature where a match for an indexed inductive type does not type check although intuition says it should. An example appears as Exercise 15.4.5. The reason for the mismatch with intuition is that the type of the inductive function explaining the match would violate the index condition.

As it comes to expressivity added by indexed inductives, inductive equality gives us something we didn't have before (see Chapter 29). Numeral types and vector types can be obtained elegantly with type functions recursing on numbers not involving indexed inductives. There is a noninductive definition of a reflexive transitive closure predicate using recursion on the path length (Exercise 15.2.5), but the representation with indexed inductives is simpler and doesn't involve numbers, which need to be erased anyway (with existential quantification) to state the transitivity law (see the discussion for vectors in § 11.4.2).

Indexed inductives really pay when it comes to derivation systems as they are common in proof theory and programming languages. Examples appear in Chapters 24 (propositional deduction systems) and Chapter 26 (regular expression matching).

Section § 20.6 gives an indexed inductive predicate characterizing greatest comon divisors with three computation rules.

16 Extensionality

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

16.1 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**
  $\mathsf{FE} := \forall X^{\mathbb{T}}\, \forall p^{X \to \mathbb{T}}.\ \forall f g^{\forall x.px}.\ (\forall x. fx = gx) \to f = g$
- **Propositional extensionality**
  $\mathsf{PE} := \forall P Q^{\mathbb{P}}.\ (P \longleftrightarrow Q) \to P = Q$
- **Proof irrelevance**
  $\mathsf{PI} := \forall Q^{\mathbb{P}}.\ \forall a b^{Q}.\ a = b$

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 $\mathsf{sig}\, p$ 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 $f^{\mathsf{B}\to\mathsf{B}}$ as boolean pairs $(f\,\mathsf{true}, f\,\mathsf{false})$. Under FE, the boolean function can be fully recovered from the pair.

Fact 16.1.1 $\mathsf{FE} \to \forall f^{\mathsf{B}\to\mathsf{B}}.\ \ f = (\lambda ab.\ \text{IF } b \text{ THEN } \pi_1 a \text{ ELSE } \pi_2 a)\,(f\,\mathsf{true}, f\,\mathsf{false})$.

Exercise 16.1.2 Prove the following:

a) $\mathsf{FE} \to \forall fg^{\mathsf{B}\to\mathsf{B}}.\ f\mathsf{true} = g\mathsf{true} \to f\mathsf{false} = g\mathsf{false} \to f = g$.

b) $\mathsf{FE} \to \forall f^{\mathsf{B}\to\mathsf{B}}.\ (f = \lambda b.\,b) \lor (f = \lambda b.\,!b) \lor (f = \lambda b.\,\mathsf{true}) \lor (f = \lambda b.\,\mathsf{false})$.

Exercise 16.1.3 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) $\mathsf{FE} \to \mathcal{E}(\mathsf{B} \to \mathsf{B})$.

16.2 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:

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

Exercise 16.2.1 Prove $x \in (p - q) \longleftrightarrow x \in p \land x \notin q$. Check that the equation $(x \in (p - q)) = (x \in p \land x \notin q)$ holds by computational equality.

Exercise 16.2.2 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) $\mathsf{SE} \to p - (q \cup r) = (p - q) \cap (p - r)$.

16.3 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 16.3.1 $\bot$ 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 16.3.2 $\mathsf{PE} \to \mathsf{PI}$.

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 16.3.3 Prove $\mathcal{D}(\mathsf{unique}(\top + \bot))$ and $\mathcal{D}(\mathsf{unique}(\top + \top))$.

Exercise 16.3.4 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 16.3.5 Prove the following:

a) Uniqueness propagates forward through surjective functions:
$\forall XY^{\mathbb{T}}\forall f^{X\to Y}.\ \mathsf{surjective}\,\mathsf{f} \to \mathsf{unique}(X) \to \mathsf{unique}(Y)$.

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 16.3.6 Prove $\mathsf{FE} \to \mathsf{unique}\,(\top \to \top)$.

Exercise 16.3.7 Assume PI and $p^{X\to\mathbb{P}}$. Prove $\forall xy\ \forall ab.\ x = y \to (x,a)_p = (y,b)_p$.

Exercise 16.3.8 Suppose there is a function $f : (\top \vee \top) \to \mathsf{B}$ such that $f(\mathsf{L\,I}) = \mathsf{true}$ and $f(\mathsf{R\,I}) = \mathsf{false}$. Prove $\neg\,\mathsf{PI}$. Convince yourself that without the propositional discrimination restriction you could define a function f as assumed.

Exercise 16.3.9 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 propositional discrimination restriction you could define a function f as assumed.

Exercise 16.3.10 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 (31.4.1), which says that $\mathbb{P}$ embeds into no proposition.

16.4 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.

17 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.

17.1 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 17.1.1 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 17.1.2 (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 17.1 shows prominent equivalences whose left-to-right directions are only provable with XM. Note the de Morgan laws for conjunction and universal quantification. Recall that the de Morgan laws for disjunction and existential quantification

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

have constructive proofs.

Exercise 17.1.3

a) Prove the right-to-left directions of the equivalences in Figure 17.1.

b) Prove the left-to-right directions of the equivalences in Figure 17.1 using XM.

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

Figure 17.1: Prominent equivalences only provable with XM

Exercise 17.1.4 Prove the following equivalences possibly using XM. In each case find out which direction needs XM.

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

Exercise 17.1.5 Make sure you can prove the de Morgan laws for disjunction and existential quantification (not using XM).

Exercise 17.1.6 Prove that $\forall PQR^{\mathbb{P}}.\ (P \to Q) \lor (Q \to R)$ is equivalent to XM.

Exercise 17.1.7 Explain why Peirce's law and the double negation law are independent in Coq's type theory.

Exercise 17.1.8 Prove the equivalence of Peirce's law and the double negation law directly without going through other laws or disjunction. In each case only a single instance of the other law is needed, which can be taken right at the beginning. Use Coq's tactic tauto to convince yourself that you have chosen the right instance.

Exercise 17.1.9 (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 $P \lor \neg P$ with $X := P \lor \neg P$ and $\lambda\_.\bot$, and exploit that $\neg\neg(P \lor \neg P)$ is provable.

Exercise 17.1.10 (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 17.1.2.

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 17.1.11 (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 [31] discussing the drinker paradox and suggesting it does not imply excluded middle in a more conventional logical framework.

Exercise 17.1.12 (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.

17.2 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 24.7.2).

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 17.2.1 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 17.2.2 Prove the equivalences of Fact 17.2.1.

Exercise 17.2.3 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 17.2.4 Prove $\neg(\forall x.\neg px) \longleftrightarrow \neg\neg\exists x.px$.

Exercise 17.2.5 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 17.2.6 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 17.2.7 (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) $\forall p^{\mathbb{B} \to \mathbb{P}}.\, (\forall x.\, \neg\neg px) \to \neg\neg\forall x.\, px$

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 17.2.8 (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.

17.3 Definite Propositions

We define **definite propositions** as propositions for which excluded middle holds:

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

Fact 17.3.1 $\mathsf{XM} \longleftrightarrow \forall P^{\mathbb{P}}.\, \mathsf{definite}\, P$.

We may see definite propositions as propositionally decided propositions. Computationally decided propositions are always propositionally decided, but not necessarily vice versa.

Fact 17.3.2

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 17.3.3 (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 17.3.4 (Definite de Morgan) $\mathsf{definite}\, P \vee \mathsf{definite}\, Q \to \neg(P \wedge Q) \longleftrightarrow \neg P \vee \neg Q$.

Exercise 17.3.5 Prove the above facts.

17.4 Stable Propositions

We define **stable propositions** as propositions where double negation elimination is possible:

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

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

Fact 17.4.1 $\mathsf{XM} \longleftrightarrow \forall P^{\mathbb{P}}.\ \mathsf{stable}\, P$.

Definite propositions are stable, but not necessarily vice versa.

Fact 17.4.2 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 17.4.3 (Characterization) $\mathsf{stable}\, P \ \longleftrightarrow\ \exists Q^{\mathbb{P}}.\ P \longleftrightarrow \neg Q$.

Corollary 17.4.4 Negated propositions are stable: $\forall P^{\mathbb{P}}.\ \mathsf{stable}(\neg P)$.

Fact 17.4.5 $\top$ and $\bot$ are stable.

Fact 17.4.6 (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 17.4.7 (Extensionality) Stability is invariant under propositional equivalence: $(P \longleftrightarrow Q) \to \mathsf{stable}\, P \to \mathsf{stable}\, Q$.

Fact 17.4.8 (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 17.4.9 Prove the above facts.

Exercise 17.4.10 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 17.4.11 Prove $(\forall x.\ \mathsf{stable}\,(px)) \to \neg(\forall x.px) \longleftrightarrow \neg\neg\exists x.\neg px$.

Exercise 17.4.12 Prove $\mathsf{FE} \to \forall f g^{\mathsf{N}\to\mathsf{B}}.\ \mathsf{stable}(f = g)$.

Exercise 17.4.13 Prove $\mathsf{XM} \longleftrightarrow \forall P^{\mathbb{P}}\, \exists Q^{\mathbb{P}}.\ P \longleftrightarrow \neg Q$.

Exercise 17.4.14 We define **classical variants** of conjunction, disjunction, and existential quantification:

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

The definitions are obtained from the impredicative 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.

17.5 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 17.5.1 $\mathsf{TVS} \longleftrightarrow \mathsf{XM} \land \mathsf{PE}$.

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:

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

Altogether we have the following hierarchy: TVS ⇒ XM ⇒ IXM ⇒ WXM.

Exercise 17.5.2 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 17.5.3 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 17.5.4 Prove $(\forall X^{\mathbb{T}}.\ X = \top \vee X = \bot) \to \bot$.

Exercise 17.5.5 (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 17.5.6 (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.

17.6 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.

18 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 16 and the different variants of excluded middle from Chapter 17 are all claimed independent. These claims are backed up by model-theoretic studies in the literature.

18.1 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:

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

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

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

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

Fact 18.1.1 (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 18.1.2 (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 18.1.1. ■

Exercise 18.1.3 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 18.1.4 Let $P \to Q$ be provable. Show that P and Q are both independent if P is consistent and Q is unprovable.

Exercise 18.1.5 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 $\neg\mathsf{provable}\,(\bot) \longleftrightarrow \neg\forall P^{\mathbb{P}}.\ \mathsf{provable}\,(P)$.

18.2 Consistency

Fact 18.2.1 (Consistency) The following propositions are equivalent:

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

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

2 → 3. Trivial.

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

1 → 4. We assume that $\bot$ is unprovable, P is provable, and $\neg P$ is provable. By PMP we have $\mathsf{provable}\,\bot$. Contradiction.

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

1 → 5. Follows with PMP.

5 → 1. Assume $\mathsf{disprovable}\,(\bot) \to \neg\mathsf{provable}\,(\bot)$. It suffices to show $\mathsf{disprovable}(\neg\bot)$, which follows with PI. ∎

Exercise 18.2.2 We may consider more abstract provability predicates

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

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

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

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

Exercise 18.2.3 (Hilbert style assumptions) The assumptions PI, PK, and PC can be obtained from the simpler assumptions

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

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

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

Part II

More Basics

19 Arithmetic Recursion

This chapter is about functions recursing on numeric arguments in a non-structural manner. The examples we consider are Euclidean division, greatest common divisors, and the Fibonacci numbers. We describe the functions with procedural specifications and construct satisfying reducible functions using structural recursion on a step index formalizing the arithmetic termination argument. We show uniqueness of the procedural specifications using complete induction and size induction, two induction operators obtained by structural induction on a step index.

We also define course-of-values recursion as structural recursion on the length of vectors. With course-of-values recursion we obtain a Fibonacci function from the step function generating the Fibonacci sequence.

19.1 Complete Induction

Complete induction is a well-known proof rule saying that to prove px it is ok to assume py holds for all $y < x$:

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

We will establish complete induction as a certifying function with the above type.

Complete induction can be used to construct functions since p may be a type function.

When we use complete induction to construct a proof, we will refer to the hypothesis

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

as the *inductive hypothesis*. When we use complete induction to construct a function, we will refer to the function of the type

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

as the *step function*. Moreover, we will refer to the certifying function for complete induction as the complete induction operator.

Compared to structural induction

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

complete induction introduces only one proof obligation. Moreover, the inductive hypothesis of complete induction is stronger than the inductive hypothesis of structural induction in that it provide py for all $y < x$, not just the predecessor.

Computationally, complete induction says that when we compute a function f for a number x, we can obtain fy for all $y < x$ by recursion.

Constructing a complete induction operator with a structural induction operator is straightforward. The trick is to replace the claim $\forall x.px$ with the equivalent claim $\forall nx.\ x < n \to px$ and do structural induction on the introduced upper bound n.

Definition 19.1.1 (Complete induction operator)
$\forall p^{\mathsf{N}\to\mathbb{T}}.\ (\forall x.\ (\forall y.\ y < x \to py) \to px) \to \forall x.px.$

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 follows withe computational falsity elimination since we have the hypothesis $x < 0$. 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 linear arithmetic ∎

Exercise 19.1.2 (Uniqueness of procedural specification)
We call a procedural specification **unique** if all functions satisfying the specification agree. Prove with complete induction that the procedural specification (of the Fibonacci function)

$$\begin{aligned} &\mathsf{Fib} : (\mathsf{N} \to \mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N} \\ &\mathsf{Fib}\, f\, n \ := \ \text{IF } n \le 1 \text{ THEN } n \text{ ELSE } f(n-2) + f(n-1) \end{aligned}$$

is unique: $\forall ff'.\ (\forall n.\ fn = \mathsf{Fib}\, f\, n) \to (\forall n.\ f'n = \mathsf{Fib}\, f'\, n) \to (\forall n.\ fn = f'n).$

Exercise 19.1.3 (Uniqueness of procedural specification)
Prove that the procedural specification (of the Ackermann function)

$$\begin{aligned} \mathsf{Ack} :\ & (\mathsf{N} \to \mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ \mathsf{Ack}\, f\, 0\, y \ &:= \ \mathsf{S}y \\ \mathsf{Ack}\, f\, (\mathsf{S}x)\, 0 \ &:= \ fx1 \\ \mathsf{Ack}\, f\, (\mathsf{S}x)(\mathsf{S}y) \ &:= \ fx(f(\mathsf{S}x)\, y) \end{aligned}$$

is unique.

Exercise 19.1.4 (Double induction) Prove the following double induction principle for numbers (from Smullyan and Fitting [27]):

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

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.

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.

19.2 Size Induction

Size induction is a generalization of complete induction working for all types that have a numeric size function:

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

With size induction, when we prove px, we can assume py for all y whose size is smaller than the size of x. The construction of the size induction operator is a straightforward adaption of the construction of the complete induction operator.

Definition 19.2.1 (Size induction operator)

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

Proof We assume X, σ, p and the step function $F : \forall x.\ (\forall y.\ y < x \to py) \to px$ and show $\forall x.px$. We prove the equivalent claim

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

by structural induction on the upper bound n. For $n = 0$, the claim follows withe computational falsity elimination since we have the hypothesis $x < 0$. In the successor case, we assume $\sigma x < \mathsf{S}n$ and prove px. We apply the step function F, which gives us the assumption $\sigma y < \sigma x$ and the claim py. By the inductive hypothesis it suffices to show $\sigma y < \sigma n$, which follows by linear arithmetic ∎

There will be applications of size induction where the size function has two arguments. We establish a binary size induction operator adapted to this situation.

Definition 19.2.2 (Binary size induction 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 Straightforward adaption of the construction of the unary size induction operator (Fact 19.2.1). ∎

Exercise 19.2.3 Assume a size induction operator and construct a binary size induction operator and a structural induction operator not using structural induction.

19.3 Euclidean Quotient

The Euclidean quotient of two numbers x and $\mathsf{S}y$ is the number of times $\mathsf{S}y$ can be subtracted from x without truncation. This computational characterization of the Euclidean quotient can be formalized with a procedural specification

$$Dxy = \text{IF } x \le y \text{ THEN } 0 \text{ ELSE } \mathsf{S}(D(x - \mathsf{S}y)y)$$

The computation captured by the equation terminates since the first argument is decreased upon recursion. Thus if D is a function satisfying the equation, Dxy is the Euclidean quotient of x and $\mathsf{S}y$ for all numbers x and y. There are obvious questions about the procedural specification of Euclidean division:

- *Existence:* Is there a function satisfying the specification?
- *Uniqueness:* Do all functions satisfying the specification agree?
- *Formalization:* How can the specification be expressed in type theory?

We will answer the first two questions positively. Existence will be shown with a general technique *called step indexing* that applies to all procedural specifications whose recursion is guarded by an arithmetic size function. Uniqueness will follow with complete induction on the first argument x.

First we take care of the formalization of the procedural specification. We express the procedural specification with an unfolding function (§ 1.12)

$$\begin{aligned}&\Delta :\ (\mathsf{N}\to\mathsf{N}) \to \mathsf{N}\to\mathsf{N}\to\mathsf{N}\\&\Delta fxy \ :=\ \text{IF } x \le y \text{ THEN } 0 \text{ ELSE } \mathsf{S}(f(x-\mathsf{S}y)y)\end{aligned}$$

We prepare the uniqueness statement with a notation for function agreement:

$$f \equiv f' \quad \rightsquigarrow \quad \forall xy.\ fxy = f'xy$$

Fact 19.3.1 (Uniqueness)
All functions satisfying Δ agree:
$\forall f f'.\ f \equiv \Delta f \to f' \equiv \Delta f' \to f \equiv f'$.

Proof We assume $H_1 : f \equiv \Delta f$ and $H_2 : f' \equiv \Delta f'$ and prove $fxy = f'xy$ by complete induction on x. By rewriting with H_1 and H_2 the claim reduces to

$$\begin{aligned} & (\text{IF } x \le y \text{ THEN } 0 \text{ ELSE } \mathsf{S}(f(x - \mathsf{S}y)y)) \\ = \ & (\text{IF } x \le y \text{ THEN } 0 \text{ ELSE } \mathsf{S}(f'(x - \mathsf{S}y)y)) \end{aligned}$$

We now discriminate on $x - y$. If $x \le y$, the claim reduces to $0 = 0$. If $x > y$, the claim reduces to

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

which follows with the inductive hypothesis instantiated with $x - \mathsf{S}y < x$. ■

Exercise 19.3.2 Let D be a function satisfying the procedural specification for the Euclidean quotients: $D \equiv \Delta D$. Prove $Dxy \cdot \mathsf{S}y \le x < Dxy \cdot \mathsf{S}y + \mathsf{S}y$.

Exercise 19.3.3 (Euclidean remainder) The Euclidean remainder of x and $\mathsf{S}y$ is the number that remains if $\mathsf{S}y$ is subtracted from x as long as this is possible without truncation.

a) Give a procedural specification Γ formalizing the specification of the remainder.
b) Show that Γ is unique.
c) Show that every function M satisfying Γ satisfies $Mxy \le y$.
d) Let D be a function satisfying δ and M be a function satisfying Γ.
Show $x = Dxy \cdot \mathsf{S}y + Mxy$.

Exercise 19.3.4 We have shown some results about the procedural specification Δ using complete induction. It turns out that the results can also be shown using structural induction on an upper bound for the variable used for complete induction. For instance, if D satisfies Γ, one may prove

$$\forall nx.\ x < n \to Dxy \cdot \mathsf{S}y \le x < Dxy \cdot \mathsf{S}y + \mathsf{S}y$$

by structural induction on n to obtain $\forall x.\ Dxy \cdot \mathsf{S}y \le x < Dxy \cdot \mathsf{S}y + \mathsf{S}y$.

Exercise 19.3.5 Construct an induction operator

$$\begin{aligned} & \forall y^{\mathsf{N}}\, \forall p^{\mathsf{N} \to \mathbb{T}}. \\ & (\forall x \le y.\ px) \to \\ & (\forall x > y.\ p(x - \mathsf{S}y) \to px) \to \\ & \forall x.\ px \end{aligned}$$

19.4 Step-Indexed Function Construction

We now introduce a technique called *step indexing* providing for the direct construction of functions satisfying procedural specifications. Step indexing works whenever the termination of the procedural specification can be argued with an arithmetic size function.

Suppose we have a procedural specification whose termination can be argued with an arithmetic size function. Then we can define an helper 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 helper function with a sufficiently large step index.

We demonstrate the technique with the procedural specification δ of Euclidean quotients. The step-indexed helper function comes out as follows:

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

The essential result about Div is *index independence*: $\mathsf{Div}\,nxy = \mathsf{Div}\,n'xy$ whenever the step indices are large enough.

Lemma 19.4.1 (Index independence)
$\forall nn'xy.\ n > x \to n' > x \to \mathsf{Div}\,nxy = \mathsf{Div}\,n'xy.$

Proof By induction on n with n' and x 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 $n' = \mathsf{S}n_1'$, the claim reduces to

$$\begin{aligned} &\text{IF } x \le y \text{ THEN } 0 \text{ ELSE } \mathsf{S}(\mathsf{Div}\,n_1\,(x - \mathsf{S}y)\,y) \\ = \ &\text{IF } x \le y \text{ THEN } 0 \text{ ELSE } \mathsf{S}(\mathsf{Div}\,n_2\,(x - \mathsf{S}y)\,y) \end{aligned}$$

The claim now follows by discrimination on $x - y$ using the inductive hypothesis for $n_1 > x - \mathsf{S}y$ and $n_2 > x - \mathsf{S}y$. ■

Fact 19.4.2 (Existence) Let $Dx := \mathsf{Div}(\mathsf{S}x)x$. Then $D \equiv \Delta D$.

Proof The claim simplifies to

$$\begin{aligned} &\text{IF } x \le y \text{ THEN } 0 \text{ ELSE } \mathsf{S}(\mathsf{Div}\,x\,(x - \mathsf{S}y)\,y) \\ = \ &\text{IF } x \le y \text{ THEN } 0 \text{ ELSE } \mathsf{S}(\mathsf{Div}\,(\mathsf{S}(x - \mathsf{S}y))\,(x - \mathsf{S}y)\,y) \end{aligned}$$

The reduced claim follows by discrimination on $x - y$ using index independence (Lemma 19.4.1) for $x > x - \mathsf{S}y$ and $\mathsf{S}(x - \mathsf{S}y) > x - \mathsf{S}y$. ■

19.5 Greatest Common Divisor

The techniques we have seen for Euclidean quotients also work for GCDs (greatest common divisors). Recall that the GCD of two numbers can be computed by repeated non-truncating subtraction. We formalize the algorithm with the procedural specification

$$\begin{aligned}&\Gamma : \ (\mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N} \to \mathsf{N}\\ &\Gamma f x y \ := \ \text{IF } x \text{ THEN } y \text{ ELSE IF } x \le y \text{ THEN } f x (y - x) \text{ ELSE } f y x\end{aligned}$$

The recursion of the specification terminates since it decreases the binary *size function* $\sigma x y := 2 \cdot x + y$.

Fact 19.5.1 (Uniqueness)
All functions satisfying Γ agree:
$\forall f f'.\, f \equiv \Gamma f \to f' \equiv \Gamma f' \to f \equiv f'$.

Proof We assume $H_1 : f \equiv \Gamma f$ and $H_2 : f' \equiv \Gamma f'$ and prove $f x y = f' x y$ by binary size induction on $\sigma x y$. Following Γ, we consider three cases: (1) $x = 0$, (2) $0 < x \le y$, and (3) $y > x > 0$. The base case follows by computational equality, and the recursive cases follow with the inductive hypothesis. ∎

To construct a function satisfying Γ, we define a step-indexed helper function:

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

Lemma 19.5.2 (Index independence)
$\forall n n' x y.\ n > \sigma x y \to n' > \sigma x y \to \mathsf{Gcd}\, n x y = \mathsf{Gcd}\, n' x y$.

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. Let $n = \mathsf{S}n_1$ and $n' = \mathsf{S}n_1'$. Following Γ, we consider three cases: (1) $x = 0$, (2) $0 < x \le y$, and (3) $y > x > 0$. The base case follows by computational equality, and the recursive cases follow with the inductive hypothesis. ∎

Fact 19.5.3 (Existence) Let $G x y := \mathsf{Gcd}(\mathsf{S}(\sigma x y)) x y$. Then $G \equiv \Gamma G$.

Proof Following Γ, we consider three cases: (1) $x = 0$, (2) $0 < x \le y$, and (3) $y > x > 0$. The base case follows by computational equality, and the recursive cases follow with index independence (Lemma 19.5.2). ∎

Exercise 19.5.4 Construct a GCD induction operator

$$\begin{aligned}
&\forall p^{\mathsf{N}\to\mathsf{N}\to\mathbb{T}}.\\
&(\forall y.\ p0y) \to\\
&(\forall xy.\ x \le y \to px(y-x) \to pxy) \to\\
&(\forall xy.\ pxy \to pyx) \to\\
&\forall xy.\ pxy
\end{aligned}$$

19.6 Course-of-values Recursion

Recall the definition of vector types in §11.4 with a type function recursing on numbers:

$$\begin{aligned}
&\mathsf{V} : \mathbb{T} \to \mathsf{N} \to \mathbb{T}\\
&\mathsf{V}\,X\,0 \ := \ \mathbb{1}\\
&\mathsf{V}\,X\,(\mathsf{S}n) \ := \ X \times \mathsf{V}\,X\,n
\end{aligned}$$

Also recall that we are using the notation $\mathsf{V}_n X$ for $\mathsf{V}\,X\,n$. The values of a vector type $\mathsf{V}_n X$ represent sequences $\langle x_1, \dots, x_n \rangle$ whose elements are of type X. For instance, we have

$$(1,(2,(3,\mathsf{I}))) \ : \ \mathsf{V}_3\mathsf{N} \ \approx \ \mathsf{N} \times (\mathsf{N} \times (\mathsf{N} \times \mathbb{1}))$$

Given a *step function* $f : \forall n.\ \mathsf{V}_n\,X \to X$, we can start from the *empty vector* and recursively compute vectors $(fnv, v) : \mathsf{V}_{\mathsf{S}n}\,X$. We speak of **vector recursion** and formalize the computation scheme with a function

$$\begin{aligned}
&\mathsf{vecrec} : \forall X^{\mathbb{T}}.\ (\forall n.\ \mathsf{V}_n\,X \to X) \to \forall n.\ \mathsf{V}_n\,X\\
&\mathsf{vecrec}\,X\,f\,0 \ := \ \mathsf{I}\\
&\mathsf{vecrec}\,X\,f\,(\mathsf{S}n) \ := \ \text{LET } v = \mathsf{vecrec}\,X\,f\,n \text{ IN } (fnv, v)
\end{aligned}$$

Building on vector recursion, we define **course-of-values recursion** as follows:

$$\begin{aligned}
&\mathsf{covrec} : \forall X^{\mathbb{T}}.\ (\forall n.\ \mathsf{V}_n\,X \to X) \to \mathsf{N} \to X\\
&\mathsf{covrec}\,X\,f\,n \ := \ \pi_1(\mathsf{vecrec}\,X\,f\,(\mathsf{S}n))
\end{aligned}$$

An untyped form of course by value recursion appeared with the study of primitive recursive functions in the 1920s.

Course-of-values recursion can be used to construct the functions representing sequences like those of the square numbers, factorials, and Fibonacci numbers:

$$\begin{aligned}
&1, 4, 9, 16, 25, 36, \dots\\
&1, 1, 2, 6, 24, 120, \dots\\
&0, 1, 1, 2, 3, 5, \dots
\end{aligned}$$

The function for the square number sequence is

$$\mathsf{covrec}\,\mathsf{N}\,(\lambda n\_.\,n^2)$$

The function for the factorial sequence is obtained with the step function

$$\begin{aligned}
&\mathsf{facstep} : \forall n.\ \mathsf{V}_n\mathsf{N} \to \mathsf{N}\\
&\mathsf{facstep}\ 0\ \_ \ := \ 1\\
&\mathsf{facstep}\ (\mathsf{S}n)\ v \ := \ \mathsf{S}n \cdot \pi_1 v
\end{aligned}$$

Note that type checking of the second defining equation requires conversion to validate the application of the projection π_1. Finally the function for the Fibonacci sequence is obtained with the step function.

$$\begin{aligned}
&\mathsf{fibstep} : \forall n.\ \mathsf{V}_n\mathsf{N} \to \mathsf{N}\\
&\mathsf{fibstep}\ 0\ \_ \ := \ 0\\
&\mathsf{fibstep}\ 1\ \_ \ := \ 1\\
&\mathsf{fibstep}\ (\mathsf{SS}n)\ v \ := \ \pi_1(\pi_2 v) + \pi_1 v
\end{aligned}$$

Informally, the step function says that the first two Fibonacci numbers are 0 and 1, and that higher Fibonacci numbers are obtained as the sum of the two preceding Fibonacci numbers.

We prove that the Fibonacci function obtained with course-of-values recursion satisfies the standard specification of the Fibonacci sequence.

Fact 19.6.1 (Correctness) Let $\mathsf{fib} := \mathsf{covrec}\,\mathsf{N}\,\mathsf{fibstep}$.
Then $\forall n.\ \mathsf{fib}\,n = \text{IF } n \le 1 \text{ THEN } n \text{ ELSE } \mathsf{fib}\,(n-2) + \mathsf{fib}\,(n-1)$.

Proof The claim follows by case analysis on n. If $n = 0$ or $n = 1$, the claim follows by computational equality. If $n = \mathsf{SS}n'$, the claim follows by computational equality after $\mathsf{SS}n' - 2$ is replaced with n'. ■

20 Euclidean Division

We study functions for Euclidean division. We start with a relational specification and show with a certifying function that Euclidean quotients and remainders uniquely exist. The certifying function gives us abstract quotient and remainder functions satisfying a defining property. We show that the abstract functions can be computed with repeated subtraction.

Following the ideas used for quotient and remainder, we construct an abstract GCD function satisfying a relational specification of greatest common divisors. We show that the abstract GCD function can be computed by taking remainders. We also give an inductive characterization of greatest common divisors.

20.1 Existence of Quotient and Remainder

Suppose you have a chocolate bar of length x and you want to cut it in pieces of length $\mathsf{S}y$. Then the number of pieces you can obtain is the Euclidean quotient of x and $\mathsf{S}y$ and the short piece possibly remaining is the Euclidean remainder of x and $\mathsf{S}y$.

Mathematically speaking, the Euclidean quotient of x and $\mathsf{S}y$ is the maximal number a such that $a \cdot \mathsf{S}y \leq x$. Following this characterization, we define a **relational specification**

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

Given $\delta xyab$, we say that a is the **quotient** and b is the **remainder** of x and $\mathsf{S}y$. For instance, we have $8 = 2 \cdot 3 + 2$ saying that 2 is the quotient and the remainder of 8 and 3.

Note that we consider Euclidean division for x and $\mathsf{S}y$ to avoid the division-by-zero problem.

We will construct functions that for x and y compute a and b such that $\delta xyab$. This tells us that Euclidean quotients and remainders always exist. We will also show that quotients and remainders uniquely exist.

Given x and y, we compute a and b such that $\delta xyab$ by structural recursion on x. The zero case is trivial:

$$0 = 0 \cdot \mathsf{S}y + 0$$

In the successor case, we obtain

$$x = a \cdot \mathsf{S}y + b \quad \text{and} \quad b \le y$$

by recursion. If $b < y$, then

$$\mathsf{S}x = a \cdot \mathsf{S}y + \mathsf{S}b \quad \text{and} \quad \mathsf{S}b \le y$$

Otherwise, $b = y$ and we have

$$\mathsf{S}x = \mathsf{S}a \cdot \mathsf{S}y + 0$$

Fact 20.1.1 (Certifying division function) $\forall xy.\Sigma ab.\ \delta xyab$.

Proof By induction on x with y fixed following the arguments given above. ∎

Definition 20.1.2 (D and M)
We fix two function $D^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$ and $M^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$ such that $\forall xy.\ \delta xy(Dxy)(Mxy)$.

Proof Let F be a function $\forall xy.\Sigma ab.\ \delta xyab$ as provided by Fact 20.1.1. We define the functions 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). ∎

Note that Definition 20.1.2 provides D and M as abstract constants. The only thing we know about the functions D and M is that they yield quotients and remainders as asserted by the **defining property** $\forall xy.\ \delta xy(Dxy)(Mxy)$. We know have examples for simply typed abstract functions that are specified with a defining property.

Corollary 20.1.3 $Mxy < \mathsf{S}y$.

Corollary 20.1.4 $Mxy = 0 \longleftrightarrow x = Dxy \cdot \mathsf{S}y$.

Exercise 20.1.5
Convince yourself that the following statements follow by linear arithmetic:

a) $\delta xyab \;\to\; b = x - a \cdot \mathsf{S}y$

b) $Mxy = x - Dxy \cdot \mathsf{S}y$

Exercise 20.1.6 Prove $\mathsf{least}\,(\lambda k.\, x \le \mathsf{S}k \cdot \mathsf{S}y)\,(Dxy)$.

20.2 Uniqueness of Quotient and Remainder

We now show that quotients and remainders are unique.

Fact 20.2.1 (Uniqueness of δ)
$\delta xyab \to \delta xya'b' \to a = a' \land b = b'$.

Proof Using linear arithmetic, it suffices to prove

$$(b \le y) \to (b' \le y) \to (a \cdot \mathsf{S}y + b = a' \cdot \mathsf{S}y + b') \to a = a'$$

We prove the claim by induction on a with a' quantified, followed by discrimination on a'.
The three cases where $a = 0$ or $a' = 0$ follow by linear arithmetic.
Suppose $a = \mathsf{S}a_1$ and $a' = \mathsf{S}a_2$. Then the inductive hypothesis reduces the claim to $a_1 \cdot \mathsf{S}y + b = a_2 \cdot \mathsf{S}y + b'$, which follows by linear arithmetic (injectivity of S and injectivity of $+$ (Fact 12.2.4)). ∎

To obtain a solid proof, the above outline needs elaboration and verification. This is best done with a proof assistant, where the reasoner for linear arithmetic takes care of all arithmetic details.

The uniqueness of δ has important applications.

Fact 20.2.2 $x = a \cdot \mathsf{S}y \longleftrightarrow Dxy = a \land Mxy = 0$.

Proof Follows with linear arithmetic from uniqueness of δ (20.2.1) and the defining property of D and M 20.1.2. ∎

Example 20.2.3 To show the ground equations $D\,100\;3 = 25$ and $M\,100\;3 = 0$, it suffices to show $\delta\,100\;3\;25\;0$ (because of uniqueness of δ and the defining property of D and M). The reduced claim $\delta\,100\;3\;25\;0$ follows with computational equality.

Exercise 20.2.4 Let $b \le y$. Prove $D(a \cdot \mathsf{S}y + b)\,y = a$ and $M(a \cdot \mathsf{S}y + b)\,y = b$.

Exercise 20.2.5 Prove the following:

a) $Dxy = a \longleftrightarrow \exists b.\,\delta xyab$

b) $Mxy = b \longleftrightarrow \exists a.\,\delta xyab$

Exercise 20.2.6 There is a specification ofthe Euclidean quotient not mentioning the remainder: $\gamma xya := a \cdot \mathsf{S}y \le x < a \cdot \mathsf{S}y + \mathsf{S}y$. Prove the following:

a) $\gamma xya \longleftrightarrow \exists b.\,\delta xyab$

b) $Dxy = a \longleftrightarrow \gamma xya$

c) $\gamma xy(Dxy)$

Exercise 20.2.7 Prove $x \cdot \mathsf{SS}z + 1 \neq y \cdot \mathsf{SS}z + 0$.
Hint: Use uniqueness of δ.

Exercise 20.2.8 Let $\mathsf{even}\, n := \exists k.\ n = k \cdot 2$. Prove the following:

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

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

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

Hint: Characterize even with the remainder function M.

20.3 Quotient and Remainder with Repeated Subtraction

Euclidean quotient and reminder can be computed by **repeated subtraction**. The geometric intuition tells us that the Euclidean quotient of x and $\mathsf{S}y$ is the number of times $\mathsf{S}y$ can be subtracted from x without truncation, and that the remainder of x and $\mathsf{S}y$ is the number that remains.

How can we formalize this insight in computational type theory? We face the difficulty that repeated subtraction is a recursive process whose recursion is not structurally recursive. What we can do in this situation is to verify that the functions D and M satisfy the equations for repeated subtraction. Moreover, we can formulate and verify the subtraction rules for the relational specification δ.

Fact 20.3.1 (Repeated subtraction, relational version)
The following rules hold for all numbers x, y, a, b:

1. $x \leq y \rightarrow \delta x y 0 x$
2. $x > y \rightarrow \delta(x - \mathsf{S}y) y a b \rightarrow \delta x y (\mathsf{S}a) b$

Proof Both rules follow by linear arithmetic. ■

The rules justify an algorithm taking x and y as input and computing a and b such that $\delta x y a b$ by repeated subtraction. The relational algorithm can be factorized into two functional algorithms computing quotient and remainder separately. The functional algorithms can be formalized with unfolding functions as follows:

$$\mathsf{DIV} : (\mathsf{N} \rightarrow \mathsf{N}) \rightarrow \mathsf{N} \rightarrow \mathsf{N} \rightarrow \mathsf{N}$$
$$\mathsf{DIV}\, f x y \;:=\; \text{IF } x \leq y \text{ THEN } 0 \text{ ELSE } \mathsf{S}(f(x - \mathsf{S}y)y)$$

$$\mathsf{MOD} : (\mathsf{N} \rightarrow \mathsf{N}) \rightarrow \mathsf{N} \rightarrow \mathsf{N} \rightarrow \mathsf{N}$$
$$\mathsf{MOD}\, f x y \;:=\; \text{IF } x \leq y \text{ THEN } 0 \text{ ELSE } \mathsf{S}(f(x - \mathsf{S}y)y)$$

Fact 20.3.2 (Repeated subtraction, functional version)
The following equations hold for all numbers x, y:
$D x y = \mathsf{DIV}\, D x y$ and $M x y = \mathsf{MOD}\, M x y$.

Proof By the uniqueness of δ (Fact 20.2.1) and the definition of D and M (Definition 20.1.2) it suffices to show

$$\delta xy\,(\text{IF } x \le y \text{ THEN } 0 \text{ ELSE } \mathsf{S}(D(x - \mathsf{S}y)\,y))$$
$$(\text{IF } x \le y \text{ THEN } x \text{ ELSE } M(x - \mathsf{S}y)\,y)$$

We discriminate on $x - y$. If $x \le y$, the claim reduces to $\delta xy0x$, which follows with Fact 20.3.1(1). If $x > y$, the claim reduces to

$$\delta xy\,(\mathsf{S}(D(x - \mathsf{S}y)y))\,(M(x - \mathsf{S}y)y)$$

which with Fact 20.3.1(2) 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 the defining property of D and M. ■

Fact 20.3.2 is remarkable, both as it comes to the result and to the proof. It states that the abstract functions D and M satisfy procedural specifications employing repeated subtraction. The proof of the result hinges on the uniqueness of the relational specifications δ, the correctness of repeated subtraction for δ (Fact 20.3.1), and the definition of D and M.

Recall that we studied the procedural specifications DIV and Mod in §19.3. In particular we showed that both specifications are unique (Fact 19.3.1 and Exercise 19.3.3). Thus we now know that all functions satisfying the specifications agree with D and M.

Corollary 20.3.3 All functions satisfying DIV agree with D, and all functions satisfying MOD agree with M.

Exercise 20.3.4 Let $F : \forall xy.\Sigma ab.\ \delta xyab$ and $fxy := (\pi_1(Fxy),\ \pi_1(\pi_2(Fxy))$.

a) Prove that f satisfies $\delta xy(\pi_1(fxy))(\pi_2(fxy))$.

b) Prove that f satisfies the equation
$fxy = \text{IF } x \le y \text{ THEN } (0, x) \text{ ELSE LET } (a, b) = f\,(x - \mathsf{S}y)\,y \text{ IN } (\mathsf{S}a, b)$.

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 20.3.2 using uniqueness (Fact 20.2.1) and the rules for repeated subtraction (Fact 20.3.1). No induction is needed.

20.4 Divisibility

We define a **divisibility predicate** as follows:

$$n \mid x \ := \ \exists k.\ \ x = k \cdot n$$

We read $n \mid x$ as either n **divides** x, or n is a **divisor** of x, or n is a **factor** of x.

Fact 20.4.1 $Sn \mid x \longleftrightarrow Mxn = 0$.

Proof Follows with Facts 20.2.2 and 20.1.4. ■

Corollary 20.4.2 (Decidability) $\forall nx.\ \mathcal{D}(n \mid x)$

Fact 20.4.3 (Divisibility)

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. $(\forall n.\ n \mid x \longleftrightarrow n \mid y) \to x \leq y$.
6. $(\forall n.\ n \mid x \longleftrightarrow n \mid y) \to x = y$.

Proof Claims 1–4 have straightforward proofs unfolding the definition of divisibility. Claim 6 follows from claim 5.

For claim 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). ■

20.5 Greatest Common Divisors

The greatest common divisor of two numbers x and y is a number z such that the divisors of z are exactly the common divisors of x and y. Following this characterization, we define a **GCD predicate**:

$$\gamma xyz \ := \ \forall n.\ n \mid z \ \longleftrightarrow \ n \mid x \land n \mid y$$

We say z is the **GCD** of x and y if γxyz.

Fact 20.5.1 (Uniqueness) $\gamma xyz \to \gamma xyz' \to z = z'$.

Proof Straightforward with Fact 20.4.3(6). ■

Similar to Euclidean quotients, GCDs can be computed with repeated subtraction. This follows from the fact that non-truncating subtraction leaves the common divisors of two numbers unchanged.

Fact 20.5.2 $x \leq y \to n \mid x \to (n \mid y \longleftrightarrow n \mid y - x)$.

Proof Straightforward using distribution $a \cdot n + b \cdot n = (a + b) \cdot n$. ■

Corollary 20.5.3 (Subtraction rule) $x \leq y \to \gamma x(y - x)z \to \gamma xyz$.

Recall that Mxy subtracts $\mathsf{S}y$ from x as long as the subtraction doesn't truncate. Thus we may use the remainder function to compute GCDs.

Fact 20.5.4 (Remainder rule) $\gamma(\mathsf{S}x)(Myx)z \to \gamma(\mathsf{S}x)yz$.

Proof By complete induction on y using Facts 20.3.2 and 20.5.3. ▪

To compute GCDs, we need two further rules for γ. The correctness of both rules is obvious from the definition of δ.

Fact 20.5.5 (Symmetry rule) $\gamma yxz \to \gamma xyz$.

Fact 20.5.6 (Zero rule) $\gamma 0yy$.

We formulate a GCD algorithm using remainders with a procedural specification:

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

The recursion terminates since the first argument is decreased. We use the algorithm to show that GCDs exist.

Fact 20.5.7 (Existence) $\forall xy.\Sigma z.\ \gamma xyz$.

Proof We prove the claim by complete induction on x. If $x = 0$, the claim follows with the zero rule. Otherwise we prove $\Sigma z.\ \gamma(\mathsf{S}x)yz$. The inductive hypothesis gives us $\gamma(Myx)(\mathsf{S}x)z$ since $Myx < \mathsf{S}x$ (Fact 20.1.3). The claim follows with the symmetry and the remainder rule. ▪

Definition 20.5.8 (GCD function)
We fix a function gcd such that $\forall xy.\ \gamma xy(\mathsf{gcd}xy)$.

Proof Immediate with Fact 20.5.7 ▪

It remains to show that gcd satisfies the procedural specification.

Fact 20.5.9 $\mathsf{gcd} \equiv \Gamma\,\mathsf{gcd}$.

Proof By uniqueness of γ and the defining property of gcd it suffices to show $\gamma xy(\Gamma\,\mathsf{gcd}\,xy)$. If $x = 0$, the claim follows with the zero rule. Otherwise the claim reduces to $\gamma(\mathsf{S}x)y(\mathsf{gcd}(Myx)(\mathsf{S}x))$, which in turn reduces with the remainder and symmetry rule to an instance of the defining property of gcd. ▪

Exercise 20.5.10
Show that every function satisfying the procedural specification Γ agrees with gcd.

Exercise 20.5.11
Show $\mathsf{gcd}\,xy = \textsc{if}\ x\ \textsc{then}\ y\ \textsc{else if}\ x \le y\ \textsc{then}\ \mathsf{gcd}\,x(y-x)\ \textsc{else}\ \mathsf{gcd}\,yx$.

20.6 Inductive GCD Predicate

The three computation rules for GCDs

$$\mathsf{G}_1\ \frac{}{\mathsf{G}\,0yy} \qquad \mathsf{G}_2\ \frac{\mathsf{G}\,xyz}{\mathsf{G}\,yxz} \qquad \mathsf{G}_3\ \frac{x \le y \qquad \mathsf{G}\,x(y-x)z}{\mathsf{G}\,xyz}$$

yield an inductive predicate $G : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \to \mathbb{P}$. We accommodate all three arguments of G as non-parametric. We will show that G agrees with the GCD predicate γ. This establishes G as an inductive characterization of GCDs.

We will cary out the equivalence proof for abstract predicates $p^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}\to\mathbb{P}}$ satisfying the computation rules for GCDs:

1. $p0yy$ *zero rule*
2. $pxyz \to pyxz$ *symmetry rule*
3. $x \le y \to px(y-x)z \to pxyz$ *subtraction rule*

We call such predicates **gcd relations**.

Fact 20.6.1 (Soundness) For every gcd relation: $\mathsf{G}xyz \to pxyz$.

Proof By induction on the derivation of $\mathsf{G}xyz$. Straightforward since the defining rules of G agree with the properties required for gcd relations. ■

We may phrase soundness as saying that G is the least gcd relation. The other direction of the equivalence is more demanding.

Fact 20.6.2 (Totality) $\forall xy.\ \Sigma z.\ \mathsf{G}xyz$.

Proof By size induction of $x + y$. If $x = 0$ or $y = 0$, the claim follows with the zero and possibly the symmetry rule. Otherwise we have either $1 \le x \le y$ or $1 \le y \le x$. Using the symmetry rule the case $1 \le x \le y$ remains. The inductive hypothesis gives us $\mathsf{G}x(y-x)z$. The claim follows with the subtraction rule. ■

For the completeness direction we need a **functional** gcd relation:

$$\forall xyzz'.\ pxyz \to pxyz' \to z = z'$$

Fact 20.6.3 (Completeness) For every functional gcd relation: $pxyz \to \mathsf{G}xyz$.

Proof Let $pxyz$. By totality and soundness we have $\mathsf{G}xyz'$ and $pxyz'$. Now $z = z'$ by functionality of p. Thus $\mathsf{G}xyz'$. ■

Corollary 20.6.4 G agrees with every functional gcd relation.

Theorem 20.6.5 G agrees with the GCD predicate γ.

Proof We have shown in § 20.5 that γ is a functional gcd relation. ■

Assuming extensionality (PE and FE), our results say that $\mathsf{G} = \gamma$ and that γ is the only functional gcd relation.

Exercise 20.6.6 Give and verify an inductive characterization of the Euclidian division predicate $\lambda xyz.\ z \cdot \mathsf{S}y \le x < \mathsf{S}z \cdot \mathsf{S}y$.

Exercise 20.6.7 Give and verify an inductive characterization of the Euclidian remainder predicate.

Exercise 20.6.8 Define an eliminator for G justifying induction on derivations as used in the proof of the soundness result (Fact 20.6.1).

20.7 Reducible Quotient and Remainder Functions

We now define reducible functions $\mathsf{Div}\,xcy$ and $\mathsf{Mod}\,xcy$ for quotient and remainder by structural recursion on x. The auxiliary argument c will be decremented upon recursion. We will have $Dxy = \mathsf{Div}\,xyy$ and $Mxy = \mathsf{Mod}\,xyy$.

Definition 20.7.1

$$\begin{array}{rcl} \mathsf{Div} : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \to \mathsf{N} & & \\ \mathsf{Div}\ 0\ c\ y & := & 0 \\ \mathsf{Div}\ \mathsf{S}x\ 0\ y & := & \mathsf{S}(\mathsf{Div}\ xyy) \\ \mathsf{Div}\ \mathsf{S}x\ \mathsf{S}c\ y & := & \mathsf{Div}\ xcy \end{array} \qquad \begin{array}{rcl} \mathsf{Mod} : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \to \mathsf{N} & & \\ \mathsf{Mod}\ 0\ c\ y & := & y - c \\ \mathsf{Mod}\ \mathsf{S}x\ 0\ y & := & \mathsf{Mod}\ xyy \\ \mathsf{Mod}\ \mathsf{S}x\ \mathsf{S}c\ y & := & \mathsf{Mod}\ xcy \end{array}$$

The design of Div and Mod is explained with a correctness lemma relating the functions with the specification δ.

Lemma 20.7.2 (Correctness of Div and Mod)
$c \le y \to \delta\,(x + y - c)\,y\,(\mathsf{Div}\,xcy)\,(\mathsf{Mod}\,xcy)$.

Proof By induction on x with c quantified.

Let $x = 0$ and $c \le y$. We show $\delta(y - c)y0(y - c)$. Follows by linear arithmetic.

For the successor case we assume $c \le y$ and show $\delta(\mathsf{S}x + y - c)y(\mathsf{Div}\,(\mathsf{S}x)cy)\ (\mathsf{Mod}(\mathsf{S}x)cy)$. We discriminate on c.

Let $c = 0$. Then the claim simplifies to $\delta(x + \mathsf{S}y)\,y\,(\mathsf{S}(\mathsf{Div}\,xyy))\,(\mathsf{Mod}xyy)$. The inductive hypothesis instantiated with y gives us $\delta xy\,(\mathsf{Div}\,xyy)\,(\mathsf{Mod}xyy)$. The claim follows with linear arithmetic.

Let $c = \mathsf{S}c'$. Then the claim simplifies to $\delta(x + y - c)\,y\,(\mathsf{Div}\,xcy)\,(\mathsf{Mod}xcy)$, which is the inductive hypothesis instantiated with c. ■

Fact 20.7.3 (Correctness of Div and Mod)
$Dxy = \mathsf{Div}\,xyy$ and $Mxy = \mathsf{Mod}\,xyy$.

Proof Follows with the uniqueness of δ, the defining property of D and M, and Lemma 20.7.2 instantiated with $c = y$. ■

Exercise 20.7.4 (Remainder without subtraction) The first equation of Mod uses subtraction. The use of subtraction can be eliminated with an additional argument d acting as a counter initialized with 0 and being incremented such that $c + d = y$. Verify the correctness of such a remainder function.

20.8 Notes

Informally, one may start a development of Euclidean division with separate recursive functions for quotient and remainder applying subtraction. However, this is not an option in a computational type theory where recursion is restricted to structural recursion. Thus our development started with a certifying division function obtaining quotient and remainder with structural recursion. The certifying function gives us abstract functions for quotient and remainder. Using the uniqueness of the specification of Euclidean division, we showed that the abstract functions satisfy the equations for repeated subtraction and agree with reducible functions employing an auxiliary argument.

The proofs in this section often involve considerable formal detail, which is typical for program verification. They are examples of proofs whose construction and analysis profits much from working with a proof assistant, where a prover for linear arithmetic takes care of tedious arithmetic details. When done by hand, the details of the proofs can be overwhelming and their verification is error-prone.

21 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.

21.1 Inductive Definition

Lists represent finite sequences $[x_1, \dots, x_n]$ 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 representing the sequence $[x_1, \dots, x_n]$ a list representing the sequence $[x, x_1, \dots, x_n]$. Given a list cons x A, we call x the **head** and A the **tail** of the list. We say that lists provide a nested pair representation of sequences.

Formally, we obtain lists with an inductive type definition

$$\mathcal{L}(X : \mathbb{T}) : \mathbb{T} \ ::= \ \mathsf{nil} \mid \mathsf{cons}\,(X, \mathcal{L}(X))$$

The type constructor $\mathcal{L} : \mathbb{T} \to \mathbb{T}$ gives us a **list type** $\mathcal{L}(X)$ for every **base type** X. The value constructor $\mathsf{nil} : \forall X^{\mathbb{T}}.\ \mathcal{L}(X)$ gives us an **empty list** for every base type. Finally, the value constructor $\mathsf{cons} : \forall X^{\mathbb{T}}.\ X \to \mathcal{L}(X) \to \mathcal{L}(X)$ provides for the construction of nonempty lists by adding an element in front of a given list. Lists of type $\mathcal{L}(X)$ are called **lists over X**. Note that all elements of a list over X must have type X.

For nil and cons, we don't write the first argument X. We use the notations

$$\begin{aligned} [\,] &:= \mathsf{nil} \\ x :: A &:= \mathsf{cons}\, x\, A \end{aligned}$$

and omit parentheses as follows:

$$x :: y :: A \quad \rightsquigarrow \quad x :: (y :: A)$$

When convenient, we shall use the sequence notation $[x_1, \dots, x_n]$ for lists.

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.

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 **eliminator for lists** as follows:

$$\begin{aligned} \mathsf{E}_{\mathcal{L}} :\ & \forall X^{\mathbb{T}}\, \forall p^{\mathcal{L}(X)\to\mathbb{T}}.\ p\,[\,] \to (\forall x A.\ pA \to p(x :: A)) \to \forall A.\, pA \\ \mathsf{E}_{\mathcal{L}}\, X p e_1 e_2\, [\,] &:= e_1 \\ \mathsf{E}_{\mathcal{L}}\, X p e_1 e_2\, (x :: A) &:= e_2 x A (\mathsf{E}_{\mathcal{L}}\, X p e_1 e_2 A) \end{aligned}$$

The eliminator provides for inductive proofs, recursive function definitions, and structural case analysis.

Fact 21.1.1 (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 proofs for numbers in §4.3. Claim (4) corresponds to $\mathsf{S}n \neq n$ and follows by induction on A with x quantified. ■

Fact 21.1.2 (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 21.1.1. In case both lists are nonempty with heads x and y, an additional case analysis on $x = y$ is needed. ■

Exercise 21.1.3 Prove $\forall X^{\mathbb{T}}\, A^{\mathcal{L}(X)}.\ \mathcal{D}(A = [\,])$.

Exercise 21.1.4 Prove $\forall X^{\mathbb{T}}\, A^{\mathcal{L}(X)}.\ (A = [\,]) + \Sigma x B.\ A = x :: B$.

21.2 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:

$$\begin{aligned} \mathsf{len}\,[x_1,\dots,x_n] &= n && \textbf{length} \\ [x_1,\dots,x_m] +\!\!+ [y_1,\dots,y_n] &= [x_1,\dots,x_m,y_1,\dots,y_n] && \textbf{concatenation} \\ f@[x_1,\dots,x_n] &= [fx_1,\dots,fx_n] && \textbf{map} \end{aligned}$$

Formally, we define the operations as recursive functions:

$$\mathsf{len} : \forall X^{\mathbb{T}}.\, \mathcal{L}(X) \to \mathsf{N}$$
$$\begin{aligned} \mathsf{len}\,[] &:= 0 \\ \mathsf{len}\,(x :: A) &:= S\,(\mathsf{len}\,A) \end{aligned}$$

$$+\!\!+ : \forall X^{\mathbb{T}}.\, \mathcal{L}(X) \to \mathcal{L}(X) \to \mathcal{L}(X)$$
$$\begin{aligned} [] +\!\!+ B &:= B \\ (x :: A) +\!\!+ B &:= x :: (A +\!\!+ B) \end{aligned}$$

$$@ : \forall XY^{\mathbb{T}}.\, (X \to Y) \to \mathcal{L}(X) \to \mathcal{L}(Y)$$
$$\begin{aligned} f@[] &:= [] \\ f@(x :: A) &:= fx :: (f@A) \end{aligned}$$

We treat X and Y as implicit arguments.

Fact 21.2.1

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. ■

21.3 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 21.3.1 (Existential Characterization) $x \in A \longleftrightarrow \exists A_1 A_2.\, A = A_1 \mathbin{+\!\!+} x :: A_2$.

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 21.3.2 $\forall x a^X \forall A^{\mathcal{L}(X)}.\ \mathcal{E}(X) \to x \in a :: A \to (x = a) + (x \in A)$.

Proof Straightforward. ▪

Fact 21.3.3 (Factorization) $\forall x^X A^{\mathcal{L}(X)}.\ \mathcal{E}(X) \to x \in A \to \Sigma A_1 A_2.\, A = A_1 \mathbin{+\!\!+} x :: A_2$.

Proof By induction on A. The nil case is contradictory. In the cons case a case analysis using Fact 21.3.2 closes the proof. ▪

Fact 21.3.4 (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 21.3.5 (Membership laws)

1. $x \in A \mathbin{+\!\!+} B \ \longleftrightarrow\ x \in A \vee x \in B$.
2. $x \in f@A \ \longleftrightarrow\ \exists a.\, a \in A \wedge x = fa$.

Proof By induction on A. ▪

Fact 21.3.6 (Injective map)
injective $f \to fx \in f@A \to x \in A$.

Proof Follows with 21.3.5(2). ▪

Recall that finite quantification over numbers preserves decidability (Fact 13.3.3). Similarly, quantification over the elements of a list preserves decidability. In fact, quantification over the elements of a list is a form of finite 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 21.3.7 (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 21.3.8
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 21.3.9 (EWO) Let p be a decidable predicate on a type X.
Construct a function $\forall A.\ (\exists x \in A.\ px) \to (\Sigma x \in A.\ px)$.

21.4 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:

$$\begin{aligned} A \subseteq B &:= \forall x.\ x \in A \to x \in B \\ A \equiv B &:= A \subseteq B \land B \subseteq A \end{aligned}$$

Note that two lists are equivalent if and only if they represent the same set.

Fact 21.4.1 List inclusion $A \subseteq B$ is reflexive and transitive. List equivalence $A \equiv B$ is reflexive, symmetric, and transitive.

Fact 21.4.2 We have the following properties for membership, inclusion, and equivalence of lists.

$$\begin{array}{ll}
x \notin [] & x \in [y] \longleftrightarrow x = y \\
[] \subseteq A & A \subseteq [] \to A = [] \\
x \in y :: A \to x \neq y \to x \in A & x \notin y :: A \to x \neq y \land x \notin A \\
A \subseteq B \to x \in A \to x \in B & A \equiv B \to x \in A \longleftrightarrow x \in B \\
A \subseteq B \to x :: A \subseteq x :: B & A \equiv B \to x :: A \equiv x :: B \\
A \subseteq B \to A \subseteq x :: B & x :: A \subseteq B \longleftrightarrow x \in B \land A \subseteq B \\
x :: A \subseteq x :: B \to x \notin A \to A \subseteq B & x :: A \subseteq [y] \longleftrightarrow x = y \land A \subseteq [y] \\
x :: A \equiv x :: x :: A & x :: y :: A \equiv y :: x :: A \\
x \in A \to A \equiv x :: A & \\
x \in A +\!\!\!+ B \;\longleftrightarrow\; x \in A \lor x \in B & \\
A \subseteq A' \to B \subseteq B' \to A +\!\!\!+ B \subseteq A' +\!\!\!+ B' & A +\!\!\!+ B \subseteq C \;\longleftrightarrow\; A \subseteq C \land B \subseteq C
\end{array}$$

Proof Except for the membership fact for concatenation, which already appeared as Fact 21.3.5, all claims have straightforward proofs not using induction. ■

Fact 21.4.3 (Rearrangement)
$x \in A \to \exists B.\; A \equiv x :: B \land \mathsf{len}\, A = \mathsf{len}(x :: B)$.

Proof Follows with Fact 21.3.1. There is also a direct proof by induction on A. ■

Fact 21.4.4 (Rearrangement)
$\mathcal{E}(X) \to \forall x^X.\; x \in A \to \Sigma B.\; A \equiv x :: B \land \mathsf{len}\, A = \mathsf{len}(x :: B)$.

Proof Follows with Fact 21.3.3. There is also a direct proof by induction on A using Fact 21.3.2. ■

Fact 21.4.5 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 21.3.4) and bounded quantification preserves decidability (Fact 21.3.7). ■

21.5 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 21.5.1 (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 21.3.1. ▪

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 21.5.2 (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 21.3.4) for the cons case. ▪

Corollary 21.5.3 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 21.5.4 (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 21.5.6). 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 21.5.5 (Discriminating element)
Every nonrepeating list over a discrete type contains for every shorter list an element not in the shorter list: $\forall AB.\ \mathsf{nrep}\,A \to \mathsf{len}\,B < \mathsf{len}\,A \to \Sigma x.\ x \in A \wedge x \notin B$.

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 21.4.4 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 21.5.6 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 21.5.5, 21.5.1, and 21.5.3.

For (1), assume $\mathsf{len}\,A > \mathsf{len}\,B$ and derive a contradiction with Fact 21.5.5.

Claims (2) and (3) follow from Claim (1), where for (3) we assume $\mathsf{nrep}\,A$ and derive a contradiction (justified by Corollary 21.5.3).

For (4), we assume $\mathsf{rep}\,B$ and derive a contradiction (justified by Corollary 21.5.3). By Fact 21.5.1, 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 21.5.5 for $x :: A$ and B, we obtain $z \in x :: A$ and $z \notin B$, which is contradictory. ■

We remark that Corollary 21.5.6 (3) may be understood as a pigeonhole lemma.

Exercise 21.5.7 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 21.5.8 (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 21.5.6 and Exercise 21.5.7.

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 §22.6.

Exercise 21.5.9 (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 21.5.10 (Partition) The proof of Corollary 21.5.3 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 21.5.11 (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 21.5.12 (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 21.4.3.

Exercise 21.5.13 (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 $\mathsf{even}\,n + \mathsf{odd}\,n$.

21.6 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 21.6.1 The empty list is a segment.

Fact 21.6.2 Segments of equal length are equivalent.

Fact 21.6.3 Segments are serial.

Fact 21.6.4 (Segment existence)
$\forall n.\, \Sigma A.\, \mathsf{segment}\,A \wedge \mathsf{len}\,A = n \wedge \mathsf{nrep}\,A.$

Proof By induction on A. ■

Fact 21.6.5 Segments are nonrepeating.

Proof Let A be a segment. By Facts 21.6.4 and 21.6.2 we have an equivalent nonrepeating segment B of the same length. Hence A is nonrepeating by Fact 21.5.6(4). ■

Fact 21.6.6 (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 21.3.7(1) we can assume $\forall k{\in}A.\, k < n$ and derive a contradiction. Fact 21.6.4 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 21.5.6(1). ■

Fact 21.6.7 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 21.6.6 gives us $x \in A$ such that $x \geq n$. Fact 21.6.4 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 21.5.6(5). ■

Fact 21.6.8 (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 21.6.8 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 21.6.9 (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 21.6.8 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 21.6.10 (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:

$$\mathsf{sum}\,A > \mathsf{len}\,A \rightarrow \Sigma x.\ x \in A \wedge x \geq 2$$

First define the function sum.

Exercise 21.6.11 (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 21.6.12 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) $\mathsf{nrep}\,(\mathsf{seq}\,nk)$

21.7 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 22 for constructing injections and bijections between finite types and in Chapter 23 for constructing injections into N.

Here are the definitions of pos and sub we will use:

$$\begin{aligned} &\mathsf{pos} :\ \mathcal{L}(X) \to X \to \mathsf{N} \\ &\qquad \mathsf{pos}\, []\, x \ := \ 0 \\ &\mathsf{pos}\, (a :: A)\, x \ := \ \text{IF } \ulcorner a = x \urcorner \text{ THEN } 0 \text{ ELSE } \mathsf{S}(\mathsf{pos}\, A x) \end{aligned}$$

$$\begin{aligned} &\mathsf{sub} :\ X \to \mathcal{L}(X) \to \mathsf{N} \to X \\ &\qquad \mathsf{sub}\, y\, []\, n \ := \ y \\ &\quad \mathsf{sub}\, y\, (a :: A)\, 0 \ := \ a \\ &\mathsf{sub}\, y\, (a :: A)\, (\mathsf{S}n) \ := \ \mathsf{sub}\, y A n \end{aligned}$$

Fact 21.7.1 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 21.7.2 Prove $(\forall X^{\mathbb{T}}.\ \mathcal{L}(X) \to \mathsf{N} \to X) \to \bot$.

Exercise 21.7.3 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 21.7.4 One can realize pos and sub with option types

$$\begin{aligned}\mathsf{pos}:\ & \mathcal{L}(X) \to X \to \mathcal{O}(\mathsf{N})\\ \mathsf{sub}:\ & \mathcal{L}(X) \to \mathsf{N} \to \mathcal{O}(X)\end{aligned}$$

and this way avoid the use of escape values. Define pos and sub with option types for a discrete base type X and verify the following properties:

a) $x \in A \to \Sigma n.\ \mathsf{pos}\,Ax = {}^\circ n$

b) $n < \mathsf{len}\,A \to \Sigma x.\ \mathsf{sub}\,An = {}^\circ x$

c) $\mathsf{pos}\,Ax = {}^\circ n \to \mathsf{sub}\,An = {}^\circ x$

d) $\mathsf{nrep}\,A \to \mathsf{sub}\,An = {}^\circ x \to \mathsf{pos}\,Ax = {}^\circ n$

e) $\mathsf{sub}\,An = {}^\circ x \to x \in A$

f) $\mathsf{pos}\,Ax = {}^\circ n \to n < \mathsf{len}\,A$

21.8 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 21.5.5)

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

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 21.5.5.

Lemma 21.8.1 (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 21.4.3 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 17.4.8 (1)). Moreover, we will use the rule formulated by Fact 17.4.8 (2) to erase the double negation from the inductive hypothesis so that we can harvest the witness.

Lemma 21.8.2 (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 17.4.8 (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 21.4.3 yields some B' shorter than B such that $B \equiv a :: B'$. Using Fact 17.4.8 (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 21.8.1. ■

Exercise 21.8.3 Prove that the double negation of $\exists$ agrees with the double negation of Σ: $\neg\neg\mathsf{ex}\,p \longleftrightarrow ((\mathsf{sig}\,p \to \bot) \to \bot)$.

21.9 Element Removal

We assume a discrete type X and define a function $A \setminus x$ for **element removal** as follows:

$$\begin{aligned} \setminus :\ & \mathcal{L}(X) \to X \to \mathcal{L}(X) \\ [\,] \setminus \_ \ &:= [\,] \\ (x :: A) \setminus y \ &:= \text{IF } \ulcorner x = y \urcorner \text{ THEN } A \setminus y \text{ ELSE } x :: (A \setminus y) \end{aligned}$$

Fact 21.9.1

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 21.9.2 Prove $x \in A \to A \equiv x :: (A \setminus x)$.

Exercise 21.9.3 Prove the following equations, which are useful in proofs:

1. $(x :: A) \backslash x = A \backslash x$
2. $x \neq y \to (y :: A) \backslash x = y :: (A \backslash x)$

21.10 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 21.5.6 (3)), and every list is equivalent to a non-repeating list (Fact 21.5.4).

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 21.3.4. We prove that the cardinality function agrees with the cardinalities provided by equivalent nonrepeating lists.

Fact 21.10.1 (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 21.5.6 (2). ∎

Corollary 21.10.2

1. $\mathsf{card}\,A \le \mathsf{len}\,A$
2. $A \subseteq B \to \mathsf{card}\,A \le \mathsf{card}\,B$
3. $A \equiv B \to \mathsf{card}\,A = \mathsf{card}\,B$.
4. $\mathsf{rep}\,A \longleftrightarrow \mathsf{card}\,A < \mathsf{len}\,A$ (pigeonhole)
5. $\mathsf{nrep}\,A \longleftrightarrow \mathsf{card}\,A = \mathsf{len}\,A$
6. $x \in A \longleftrightarrow \mathsf{card}\,A = \mathsf{S}(\mathsf{card}(A \setminus x))$

Proof All facts follow without induction from Fact 21.10.1, Corollary 21.5.6, and Corollary 21.5.3. ∎

Exercise 21.10.3 Given direct proofs of (1), (4) and (5) of Corollary 21.10.2 by induction on A. Use (1) for (4) and (5).

Exercise 21.10.4 (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 21.10.5 (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 $\mathsf{disjoint}$.

21.11 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 21.11.1 (Compatibility laws for propositional equivalence)
Let $P \longleftrightarrow P'$ and $Q \longleftrightarrow Q'$. Then:

$$\begin{array}{ccc} P \land Q \longleftrightarrow P' \land Q' & P \lor Q \longleftrightarrow P' \lor Q' & (P \to Q) \longleftrightarrow (P' \to Q') \\ \neg P \longleftrightarrow \neg P' & & (P \longleftrightarrow Q) \longleftrightarrow (P' \longleftrightarrow Q') \end{array}$$

Fact 21.11.2 (Compatibility laws for list equivalence)
Let $A \equiv A'$ and $B \equiv B'$. Then:

$$\begin{array}{ccc} x \in A \longleftrightarrow x \in A' & A \subseteq B \longleftrightarrow A' \subseteq B' & A \equiv B \longleftrightarrow A' \equiv B' \\ x :: A \equiv x :: A' & A +\!\!+ B \equiv A' +\!\!+ B' & f@A \equiv f@A' \end{array}$$

Coq's setoid rewriting facility makes it possible to use the rewriting tactic for rewriting with equivalences, provided the necessary compatibility laws and equivalence relations have been registered with the facility. The compatibility laws for propositional equivalence are preregistered.

Exercise 21.11.3 Which of the compatibility laws are needed to justify rewriting the claim $\neg(x \in y :: (f@A) +\!\!+ B)$ with the equivalence $A \equiv A'$?

22 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 N_n 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.

22.1 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:

$$\mathsf{listing}_X\ A\ :=\ \mathsf{covering}\ A \land \mathsf{nrep}\ A$$

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 22.1.1 Given a covering of a discrete type, one can obtain a listing of the type: $\mathcal{E}X \to \mathsf{covering}_X\, A \to \Sigma B.\ \mathsf{listing}_X\, B \wedge \mathsf{len}\, B \leq \mathsf{len}\, A$.

Proof Follows with Facts 21.5.4. ■

Fact 22.1.2 All listings of a discrete type have the same length.

Proof Immediate with Fact 21.5.6 (2). ■

Fact 22.1.3 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 21.5.6 (4,5). ■

Exercise 22.1.4 Prove $\mathcal{I}XY \to \mathsf{covering}_Y B \to \Sigma A.\ \mathsf{covering}_X A$.

Exercise 22.1.5 Let A and B be lists over a discrete type X. Prove:

a) $\mathsf{covering}\, A \to \mathsf{nrep}\, B \to \mathsf{len}\, A \leq \mathsf{len}\, B \to \mathsf{listing}\, B$.

b) $\mathsf{listing}\, A \to \mathsf{covering}\, B \to \mathsf{len}\, B \leq \mathsf{len}\, A \to \mathsf{listing}\, B$.

22.2 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 22.2.1 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 22.1.2 and 22.1.1. ■

Fact 22.2.2 (Empty types)
Finite types with cardinality 0 are the empty types: $\mathsf{fin}_0\, X \Leftrightarrow (X \to \bot)$.

Proof Exercise. ▪

Fact 22.2.3 (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 11.2.1 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. ▪

Recall the recursive definition of numeral types N_n in §11.3.

Fact 22.2.4 (Numeral types) $\mathsf{fin}_n\, \mathsf{N}_n$.

Proof Induction on n using Facts 22.2.2 and 22.2.3. ▪

Finite types are also closed under sums and products.

Fact 22.2.5 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 9.5.1. 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 22.2.6 (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 21.3.7. ▪

Fact 22.2.7 (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 21.6.8. ▪

Exercise 22.2.8 Prove $\mathsf{fin}_0\,\bot$, $\mathsf{fin}_1\,\top$, and $\mathsf{fin}_2\,\mathsf{B}$.

Exercise 22.2.9 (Double negation shift)
Prove $\forall n\,\forall p^{\mathsf{N}_n\to\mathbb{P}}.\ (\forall x.\ \neg\neg px) \to \neg\neg\forall x.\ px$.

Exercise 22.2.10 (EWO) Give an EWO for finite types.
Hint: Use an EWO for lists as in Exercise 21.3.9.

Exercise 22.2.11 (Pigeonhole)
Prove $\mathsf{fin}_m\,X \to \mathsf{fin}_n\,Y \to m > n \to \forall f^{X\to Y}.\ \Sigma xx'.\ x \neq x' \wedge fx = fx'$.
Intuition: If we have m pigeons sitting in $n < m$ holes, there must be two pigeons sitting in the same hole.

22.3 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 22.3.1 (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 9.5.2 gives us an equality decider for X. Moreover, $g@B$ is a covering of X because of the inversion property. By Fact 22.1.1 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 22.3.2 (Alignment)
$\mathcal{I}XY \to \mathsf{fin}_m\,X \to \mathsf{fin}_n\,Y \to m \leq n$.

Proof Facts 22.3.1 and 22.2.1 (1). ∎

Corollary 22.3.3 (Transport)
$\mathcal{I}XY \to \mathcal{I}YX \to \mathsf{fin}_n\,X \to \mathsf{fin}_n\,Y$.

Proof Follows with Facts 22.3.1 and 22.3.2. ∎

Corollary 22.3.4 (Closure under bijection)
$\mathcal{B}XY \to \mathsf{fin}_n\,X \to \mathsf{fin}_n\,Y$.

Proof Follows with Facts 22.3.3 and 11.1.5. ∎

Corollary 22.3.5 The type N of numbers does not embed into finite types:
$\mathcal{I}\,\mathsf{N}\,X \to \mathsf{fin}\,X \to \bot$.

Proof Facts 22.3.1 and 22.2.7. ∎

22.4 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 22.4.1 (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 $b \in B$ (§21.7):

$$\begin{aligned} fx &:= \mathsf{sub}\, b\, B\ (\mathsf{pos}\, A\, x) \\ gy &:= \mathsf{sub}\, a\, A\ (\mathsf{pos}\, B\, y) \end{aligned}$$

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 22.4.2 $\mathsf{fin}_n\, X \to \mathsf{fin}_n\, Y \to \mathsf{inv}_{XY}\, gf \to \mathsf{inv}\, fg$.

Proof We show $f(gy) = y$ for arbitrary y. We choose a covering A of X and know by Fact 22.1.3 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 22.4.3 (Existence) $\mathsf{fin}_n\, X \to \mathsf{fin}_n\, Y \to \mathcal{B}XY$.

Proof For $n = 0$ the claim follows with Facts 22.2.2 and 11.1.7. For $n > 0$ the claim follows with Facts 22.4.1 and 22.4.2. ■

Corollary 22.4.4 (Existence) $\mathcal{I}XY \to \mathcal{I}YX \to \mathsf{fin}\, X \to \mathcal{B}XY$.

Proof Facts 22.2.1 (2), 22.3.3, and 22.4.3 ■

Corollary 22.4.5 (Listless Characterization) $\mathsf{fin}_n\, X \Leftrightarrow \mathcal{B}X\mathsf{N}_n$.

Proof Direction $\to$ follows with Facts 22.2.4 and 22.4.3. Direction $\leftarrow$ follows with Facts 22.2.4 and 22.3.4. ■

Corollary 22.4.6 $\mathsf{fin}\, X \to \Sigma n.\, \mathcal{B} X \mathsf{N}_n$

Proof Facts 22.2.1 and 22.4.5. ■

Corollary 22.4.7 (EWOs) Finite types have EWOs.

Proof Follows with Fact 22.4.5 since numeral types have EWOs (Fact 14.3.4) and injections transport EWOs (Fact 14.3.5). ■

Fact 22.4.8 (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 $a \in A$ (§21.7):

$$\begin{aligned} fx &:= \mathsf{pos}\, A\, x \\ gn &:= \mathsf{sub}\, a\, A\, n \end{aligned}$$

■

Exercise 22.4.9 Show the following facts:

a) $\mathsf{fin}_m\, \mathsf{N}_n \to m = n$.

b) $\mathcal{B}\, \mathsf{N}_m \mathsf{N}_n \to m = n$.

22.5 Upgrade Theorem

Fact 22.5.1 (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 22.1.3.

Let f be injective. Then $f@A$ is nonrepeating by Exercise 21.5.7 (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 21.5.7 (b). ■

Fact 22.5.2 (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 22.4.7). By Fact 22.5.1 we can assume a surjective function $f^{X \to Y}$. Fact 14.4.1 gives us a function g such that $\mathsf{inv}\, fg$. Fact 22.4.2 gives us $\mathsf{inv}\, gf$ as claimed. ■

Exercise 22.5.3
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 22.5.4 Show that all inverse functions of an injective function between finite types of the same cardinality agree.

22.6 Listless Development

Fact 22.4.5 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. $\mathcal{B}\,\mathsf{N}_m\mathsf{N}_n \to m = n$
2. $\forall fg^{\mathsf{N}_n\to\mathsf{N}_n}.\ \mathsf{inv}\,gf \to \mathsf{inv}\,fg$
3. $\mathcal{I}\,\mathsf{N}_n\mathsf{N}$
4. $\mathcal{I}\,\mathsf{N}\,\mathsf{N}_n \to \bot$

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 N_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:

$$L:\ \forall XY.\ (\mathcal{O}(X) \to \mathcal{O}^2(Y)) \to X \to \mathcal{O}(Y)$$
$$LXYfx\ :=\ \text{MATCH}\ f({}^\circ x)\ [\,{}^\circ b \Rightarrow b \mid \emptyset \Rightarrow \text{MATCH}\ f\emptyset\ [\,{}^\circ b \Rightarrow b \mid \emptyset \Rightarrow \emptyset\,]\,]$$

The idea behind L is simple: Given x, Lf checks whether f maps ${}^\circ x$ to ${}^\circ b$. If so, Lf maps x to b. Otherwise, Lf checks whether f maps $\emptyset$ to ${}^\circ b$. If so, Lf maps x to b. If not, Lf maps x to $\emptyset$.

Lemma 22.6.1 (Lowering) Let $f : \mathcal{O}^2(X) \to \mathcal{O}^2(Y)$ and $g : \mathcal{O}^2(Y) \to \mathcal{O}^2(X)$. Then $\mathsf{inv}\,gf \to \mathsf{inv}\,(Lg)(Lf)$.

Proof Let $\mathsf{inv}\,gf$. 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 22.6.2 (Self-injection)
Let f and g be functions $\mathsf{N}_n \to \mathsf{N}_n$. Then $\mathsf{inv}\, g f \to \mathsf{inv}\, f g$.

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 $\mathsf{inv}\, g f$. By Lemma 22.6.1 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 $\mathsf{inv}\, g f$. ∎

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 (§5.4).

We remark that the lowering operator L is related to the certifying lowering operator established by Lemma 11.2.2. However, there are essential differences. The lowering operator L uses a default value while Lemma 11.2.2 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 11.2.2, and the assumption is not available in the setting of the lowering operator.

Theorem 22.6.2 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 22.6.3 $\mathcal{B}X\mathsf{N}_n \to \mathcal{B}Y\mathsf{N}_n \to \mathsf{inv}_{XY}\, g f \to \mathsf{inv}\, f g$.

Proof Let f_1, g_1 form a bijection $\mathcal{B}X\mathsf{N}_n$ and f_2, g_2 form a bijection $\mathcal{B}X\mathsf{N}_n$. 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 22.6.2 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 22.6.4 (Cardinality)
$\mathcal{I}(\mathsf{N}_m)\mathsf{N}_n \to m \le n$.

Proof Let $f : \mathsf{N}_m \to \mathsf{N}_n$ 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 $m \le n$ 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 $\mathsf{S}m \le n$. 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 22.6.1 and the inductive hypothesis. ■

Exercise 22.6.5 Show $\mathcal{B}\,\mathsf{N}_m\mathsf{N}_n \to m = n$ using induction and the bijection theorem for option types (11.2.4).

Exercise 22.6.6 Try to do the proof of Theorem 22.6.2 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 22.6.7 (Pigeonhole)
Prove $\forall f^{\mathsf{N}_{\mathsf{S}n} \to \mathsf{N}_n}.\ \Sigma ab.\ a \ne b \wedge fa = fb$ not using lists.
Hint: A proof similar to the proof of Theorem 22.6.4 works, but the situation is simpler. The decision function from Fact 22.2.6 (c) is essential.

Exercise 22.6.8 (Double negation shift)
Prove $\forall n\, \forall p^{\mathsf{N}_n \to \mathbb{P}}.\ (\forall x.\ \neg\neg px) \to \neg\neg\forall x.\ px$ not using lists.

Exercise 22.6.9 (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 $E : \forall n.\ \mathsf{N}_n \to \mathsf{N}$.

b) Define a decoding function $D : \mathsf{N} \to \forall n.\ \mathsf{N}_{\mathsf{S}n}$.

c) Prove $Ena < n$.

d) Prove $D(E(\mathsf{S}n)a)n = a$.

e) Prove $k \le n \to E(\mathsf{S}n)(Dkn) = k$.

Hint: The definition of E needs computational falsity elimination.

Exercise 22.6.10 (Decidability)
Let p be a decidable predicate on N_n. Then:

1. $\mathcal{D}(\forall x.px)$
2. $\mathcal{D}(\exists x.px)$
3. $(\Sigma x.px) + (\forall x.\neg px)$

Exercise 22.6.11 (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) $\mathsf{choice}\perp Y$
b) $\mathsf{choice}\,X\,Y \to \mathsf{choice}\,(\mathcal{O}X)\,Y$
c) $\mathsf{choice}\,\mathsf{N}_n\,Y$
d) $\mathcal{B}\,X\,\mathsf{N}_n \to \mathsf{choice}\,XY$

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.

22.7 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 (22.6.4 and 22.6.2).

23 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.

23.1 Enumerable Types

An **enumerator** of a type X is a function $f^{\mathsf{N}\to\mathcal{O}(X)}$ that reaches all elements of X. Formally, 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 23.1.1 (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 22.4.6). ■

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 23.1.2 (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 23.1.3 (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. ■

23.2 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 23.2.1 (Countable types)

1. $\perp$ and N are countable types: $\mathsf{cty}\perp$ 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 23.1.1 and the concomitant closure facts for discrete types. ■

Fact 23.2.2 (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 23.1.3 and 23.1.2. ■

It turns out that a type X is countable if and only if $\mathcal{O}X$ is a retract of N.

Fact 23.2.3 (Retract characterization) $\mathsf{cty}\,X \Leftrightarrow \mathcal{I}(\mathcal{O}X)\mathsf{N}$.

Proof Suppose $\mathsf{cty}\,X$. Fact 23.2.2 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:

$$\begin{aligned} g'a &:= \textsc{match}\ a\ [\,{}^\circ x \Rightarrow \mathsf{S}(gx) \mid \emptyset \Rightarrow 0\,] \\ f'n &:= \textsc{match}\ n\ [\,0 \Rightarrow \emptyset \mid \mathsf{S}n \Rightarrow fn\,] \end{aligned}$$

The other direction follows with the transport lemmas for equality deciders 11.1.3 and 11.2.1 and the observation that the inverse function of the injection is an enumerator of X. ■

Fact 23.2.4 (EWOs) Countable types have EWOs.

Proof Follows with an EWO for N (Fact 14.2.2) and Facts 23.2.3, 14.3.5, and 14.3.3. ■

Fact 23.2.5 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

$$\begin{aligned} f'a &:= \textsc{match}\ a\ [\,{}^\circ x \Rightarrow \mathsf{S}(fx) \mid \emptyset \Rightarrow 0\,] \\ g'n &:= \textsc{match}\ n\ [\,0 \Rightarrow \emptyset \mid \mathsf{S}n \Rightarrow {}^\circ gn\,] \end{aligned}$$

yield an injection $\mathcal{I}(\mathcal{O}X)\mathsf{N}$. ■

Fact 23.2.6
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

$$\begin{aligned} f'x &:= f({}^\circ x) \\ g'y &:= \textsc{match}\ gy\ [\,{}^\circ x \Rightarrow x \mid \emptyset \Rightarrow x_0\,] \end{aligned}$$

yield an injection $\mathcal{I}XY$. ■

Fact 23.2.7 Nonempty countable types are exactly the retracts of N:
$X \to (\mathsf{cty}\,X \Leftrightarrow \mathcal{I}X\mathsf{N})$.

Proof Follows with Facts 23.2.3, 23.2.5, and 23.2.6. ■

Fact 23.2.8 (Uncountable type) The function type $\mathsf{N} \to \mathsf{B}$ is not countable.

Proof Suppose $\mathsf{N} \to \mathsf{B}$ is countable. Then Fact 23.2.7 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 6.3.4). ■

23.3 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 8 and 24. 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 23.3.1 $\mathsf{prefix}\, A\, B \to k < \mathsf{len}\, A \to \mathsf{sub}\, A\, k = \mathsf{sub}\, B\, k$.

Proof $\forall k.\ k < \mathsf{len}\, A \to \mathsf{sub}\, A\, k = \mathsf{sub}\, (A + C)\, k$ follows by induction on A. ∎

Theorem 23.3.2 (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 23.3.1 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 21.7.1(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 21.7.1(2). ∎

Given a type X, we call functions L and β as specified by Theorem 23.3.2 a **list enumeration** of X. Note that types are inhabited if they have a list enumeration.

23.4 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 23.4.1 $\mathcal{E}(\mathsf{tree})$.

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 23.4.2 (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 23.4.3 $\mathcal{I}\,\mathsf{tree}\,\mathsf{N}$.

Proof By Theorem 23.3.2 and Fact 23.4.1 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 23.4.2 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 23.4.4 $\mathsf{cty}\,\mathsf{tree}$.

Proof Follows with Facts 23.4.3, 23.2.1(2a), and 23.2.1(1). ∎

We can now show countability of a type by embedding it into tree. This is in many cases straightforward.

Fact 23.4.5 $\mathsf{cty}\,(\mathsf{N}\times\mathsf{N})$.

Proof By Facts 23.4.4 and 23.2.1(2a) it suffices to embed $\mathsf{N}\times\mathsf{N}$ into tree, which can be done as follows:

$$\begin{aligned} f(x,y) &:= \mathsf{T}(\mathsf{A}\,x)(\mathsf{A}\,y) \\[1ex] g(\mathsf{T}(\mathsf{A}\,x)(\mathsf{A}\,y)) &:= (x,y) \\ g\ \_ &:= (0,0) \end{aligned}$$

Now $g(f(x,y)) = (x,y)$ holds by computational equality. ∎

Note that Fact 23.4.5 can also be obtained with Cantor pairing (Fact 11.1.8). The point is that Fact 23.4.5 obtains countability of $\mathsf{N}\times\mathsf{N}$ with a routine method rather than the ingenuous Cantor pairing.

Fact 23.4.6 $\mathsf{cty}\,(\mathcal{L}\,\mathsf{N})$.

Proof By Facts 23.4.4 and 23.2.1(2a) it suffices to embed $\mathcal{L}\,\mathsf{N}$ into tree, which can be done as follows:

$$\begin{aligned} f\,[] &:= \mathsf{A}\,0 \\ f\,(x :: A) &:= \mathsf{T}(\mathsf{A}\,x)(f A) \\[1ex] g(\mathsf{T}(\mathsf{A}\,x)\,t) &:= x :: g t \\ g\ \_ &:= [] \end{aligned}$$

Now $g(fA) = A$ follows by induction on A. ∎

Fact 23.4.7 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 23.2.3) 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 23.4.6 and 23.2.1(2a) it suffices to embed $\mathcal{L}\,X$ into $\mathcal{L}\,\mathsf{N}$, which can be done as follows:

$$\begin{aligned} f\,A &:= (\lambda x.\,F(^\circ x))\,@\,A \\[1ex] g\,[] &:= [] \\ g\,(n :: A) &:= \textsc{match}\ Gn\ [\,^\circ x \Rightarrow x :: gA \mid \emptyset \Rightarrow []\,] \end{aligned}$$

Now $\forall A.\ g(fA) = A$ follows by induction on A. ∎

Exercise 23.4.8 Prove $\mathcal{I}X\mathsf{N} \to \mathcal{I}(\mathcal{L}X)(\mathcal{L}\mathsf{N})$.

Exercise 23.4.9 Prove that the type of expressions (§ 8.2) and the type of formulas (§ 24.1) are countable by embedding them into tree. No recursion is needed for the embedding.

Exercise 23.4.10 (General list product) Assume a function $f^{X \to Y \to Z}$. Construct a function $\forall AB.\, \Sigma C.\, \forall z.\, z \in C \longleftrightarrow \exists x{\in}A\, \exists y{\in}B.\, z = fxy$.

23.5 Alignments

We start with an informal presentation of ideas we are going to formalize. An *alignment* of a type X is a nonrepeating sequence $x_0, x_1, x_2, \dots$ 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 23.5.1 Cutoffs are unique. That is, all cutoffs of a function $X \to \mathsf{N}$ agree.

Fact 23.5.2 The surjective alignments are exactly the bijective functions $X \to \mathsf{N}$.

Fact 23.5.3 (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 14.4.2 since countable types have EWOs (Fact 23.2.4). ∎

Fact 23.5.4 (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 23.5.5 (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 $\mathsf{hit}\, f n$. 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 $fx = \mathsf{S}n$. ■

23.6 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 23.6.1 (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 13.3.1). That f' enumerates X and is nonrepeating follows with Facts 13.2.4 and 13.1.1 for least. ■

There is a second characterization of seriality that is useful for the second step of the alignment construction.

Fact 23.6.2 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 23.6.3 (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 23.6.1 and 23.1.3 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 $k < n$:

$$\begin{aligned} h(0) &:= 0 \\ h(\mathsf{S}n) &:= \text{IF } fn \neq \emptyset \text{ THEN } \mathsf{S}(hn) \text{ ELSE } hn \end{aligned}$$

By induction on n we show two intuitively obvious facts about h:

1. $\forall kn.\ hn = \mathsf{S}k \to \Sigma my.\ m < n \land fm = {}^\circ y \land hm = k.$
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 23.6.2.
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 $gx = gy$. ∎

23.7 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 23.7.1 (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 23.5.4 it suffices to show that g hits fx. With the Segment Lemma 23.5.5 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 21.6.6 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 23.7.2 (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 23.6.3. With the Transport Lemma 23.7.1 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 23.7.3 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 23.2.7. Now the bijection theorem 23.7.2 gives us a bijection $\mathcal{B}X\mathsf{N}$ since $\mathsf{cty}\,\mathsf{N}$. ■

23.8 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 23.8.1 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 23.5.5 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 23.8.2 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 21.6.7 since $f@A$ is serial and nonrepeating. ■

Corollary 23.8.3 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 23.8.2 we have a cutoff n of f. Thus $n < n$ if f hits n. ■

23.9 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 23.9.1 $\mathsf{XM} \to \mathsf{serial}\, f \to \mathsf{ex}(\mathsf{cutoff}\, f) \vee \mathsf{surjective}\, f$.

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 13.5.2, 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 23.9.2 $\mathsf{XM} \to \mathsf{cty}\, X \to \Box\,(\mathsf{fin}\, X + \mathsf{bijection}\, X\, \mathsf{N})$.

Proof Follows with Facts 23.6.3, 23.9.1, 23.8.1, and 23.5.3. ■

The truncation $\Box$ in Fact 23.9.2 is needed so that the disjunctive assumption XM can be harvested. Truncations are introduced in § 10.5.

Fact 23.9.3 $\mathsf{XM} \to \mathsf{cty}\, X \to \Box\, \mathcal{I}\, X \mathsf{N} \vee (X \to \bot)$.

Proof Follows with Facts 23.9.2, 22.2.2 and 22.4.8. ■

23.10 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 $°n$ represents the finite cardinality n and $\emptyset$ represents the single infinite cardinality.

Part III

Case Studies

24 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 three-valued and two-valued interpretations of formulas 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 which we obtain that classical provability is decidable and agrees with boolean entailment. We construct the certifying solver using size induction.

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.

24.1 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 24.1.1 (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.

24.2 Intuitionistic ND System

The deduction rules of the intuitionistic ND system we will consider are given in Figure 24.1 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 8.

$$\mathsf{A}\ \frac{s \in A}{A \vdash s} \qquad \mathsf{E}_\bot\ \frac{A \vdash \bot}{A \vdash s} \qquad \mathsf{I}_\to\ \frac{A, s \vdash t}{A \vdash s \to t} \qquad \mathsf{E}_\to\ \frac{A \vdash s \to t \qquad A \vdash s}{A \vdash t}$$

$$\mathsf{I}_\wedge\ \frac{A \vdash s \qquad A \vdash t}{A \vdash s \wedge t} \qquad \mathsf{E}_\wedge\ \frac{A \vdash s \wedge t \qquad A, s, t \vdash u}{A \vdash u}$$

$$\mathsf{I}^1_\vee\ \frac{A \vdash s}{A \vdash s \vee t} \qquad \mathsf{I}^2_\vee\ \frac{A \vdash t}{A \vdash s \vee t} \qquad \mathsf{E}_\vee\ \frac{A \vdash s \vee t \qquad A, s \vdash u \qquad A, t \vdash u}{A \vdash u}$$

Figure 24.1: Deduction rules of the intuitinistic ND system

All rules in Figure 24.1 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 24.1 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 24.1 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 24.1.

The rules in Figure 24.1 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 24.1 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 make use of 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 24.2.1 Below is a **derivation** for $s \vdash \neg\neg s$ depicted as a **derivation tree**:

$$\dfrac{\dfrac{\dfrac{}{s, \neg s \vdash \neg s}\,\mathsf{A} \qquad \dfrac{}{s, \neg s \vdash s}\,\mathsf{A}}{s, \neg s \vdash \bot}\,\mathsf{E}_{\to}}{s \vdash \neg\neg s}\,\mathsf{I}_{\to}$$

The labels A, $\mathsf{E}_{\to}$, and $\mathsf{I}_{\to}$ 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 table as if the formulas were propositions.
2. Translate the proof table 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 tables. 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 24.2.2 Give derivation trees for $A \vdash (s \to s)$ and $\neg\neg\bot \vdash \bot$.

Exercise 24.2.3 If you are eager to construct more derivations, Exercise 24.3.3 will provide you with interesting examples.

24.3 Formalization with Indexed Inductive Type Family

It turns out that propositional deduction systems like the one in Figure 24.2 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 24.1 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 24.2: 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 24.2. Note that the types of the constructors follow exactly the patterns of the deduction rules in Figure 24.1.

When we look at the target types of the constructors in Figure 24.2, it becomes clear that the argument s of the type constructor $A \vdash s$ is *not* a parameter since it is instantiated by the constructors for the introduction rules ($\mathsf{I}_\to$, $\mathsf{I}_\wedge$, $\mathsf{I}^1_\vee$, $\mathsf{I}^2_\vee$). Such nonparametric arguments of type constructors are called **indices**. In contrast, the argument A of the type constructor $A \vdash s$ is a parameter since it is not instantiated in the target types of the constructors. More precisely, the argument A is a *nonuniform* parameter of the type constructor $A \vdash s$ since it is instantiated in some argument types of some of the constructors ($\mathsf{I}_\to$, $\mathsf{E}_\wedge$, and $\mathsf{E}_\vee$).

We call inductive type definitions where the type constructor has indices **indexed inductive definitions**. Indexed inductive definitions can also introduce **indexed inductive predicates**. In fact, we alternatively could introduce $\vdash$ as an indexed inductive predicate and this way demote derivations from computational objects to proofs.

The suggestive BNF-style notation we have used so far to write inductive type definitions does not generalize to indexed inductive type definitions. So we will use an explicit format giving the type constructor together with the list of its value constructors. Often, the format used in Figure 24.2 will be convenient.

Fact 24.3.1 (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 24.2.1 and the remarks there after. ■

In §24.9 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 24.3.2 (Cut) $A \vdash s \;\to\; A, s \vdash t \;\to\; A \vdash t$.

Proof We assume $A \vdash s$ and $A, s \vdash t$ and derive $A \vdash t$. By $\mathsf{I}_{\to}$ we have $A \vdash (s \to t)$. Thus $A \vdash t$ by $\mathsf{E}_{\to}$. ■

The cut lemma gives us a function that given a derivation $A \vdash s$ and a derivation $A, s \vdash t$ yields a derivation $A \vdash t$. Informally, the cut lemma says that once we have derived s from A, we can use s like an assumption.

Exercise 24.3.3 Construct derivations as follows:

a) $A \vdash \neg\neg\bot \to \bot$
b) $A \vdash s \to \neg\neg s$
c) $A \vdash (\neg s \to \neg\neg\bot) \to \neg\neg s$
d) $A \vdash (s \to \neg\neg t) \to \neg\neg(s \to t)$
e) $A \vdash \neg\neg(s \to t) \to \neg\neg s \to \neg\neg t$
f) $A \vdash \neg\neg\neg s \to \neg s$
g) $A \vdash \neg s \to \neg\neg\neg s$

Exercise 24.3.4 Establish the following functions:

a) $A \vdash (s_1 \to s_2 \to t) \;\to\; A \vdash s_1 \;\to\; A \vdash s_2 \;\to\; A \vdash t$
b) $\neg\neg s \in A \;\to\; A, s \vdash \bot \;\to\; A \vdash \bot$
c) $A, s, \neg t \vdash \bot \;\to\; A \vdash \neg\neg(s \to t)$

Hint: (c) is routine if you first show $A \vdash (\neg t \to \neg s) \to \neg\neg(s \to t)$.

Exercise 24.3.5 Prove the implicative facts (1)–(6) appearing in Exercise 24.11.6.

24.4 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 24.3. 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 24.3, 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 $\mathsf{E}_\vdash$.

24.5 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.

$$
\begin{aligned}
\mathsf{E}_\vdash :\ & \forall p^{\mathcal{L}(\mathsf{For})\to\mathsf{For}\to\mathbb{T}}.\\
& (\forall As.\ s \in A \to pAs) \to\\
& (\forall As.\ pA\bot \to pAs) \to\\
& (\forall Ast.\ p(s :: A)t \to pA(s \to t)) \to\\
& (\forall Ast.\ pA(s \to t) \to pAs \to pAt) \to\\
& (\forall Ast.\ pAs \to pAt \to pA(s \land t)) \to\\
& (\forall Astu.\ pA(s \land t) \to p(s :: t :: A)u \to pAu) \to\\
& (\forall Ast.\ pAs \to pA(s \lor t)) \to\\
& (\forall Ast.\ pAt \to pA(s \lor t)) \to\\
& (\forall Astu.\ pA(s \lor t) \to p(s :: A)u \to p(t :: A)u \to pAu) \to\\
& \forall As.\ A \vdash s \to pAs
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{E}_\vdash\, pe_1 \dots e_9\, A\, \_\, (\mathsf{A}\, sh) &:= e_1 Ash\\
(\mathsf{E}_\bot\, sd) &:= e_2 As(\mathsf{E}_\vdash \dots A\bot d)\\
(\mathsf{I}_\to\, std) &:= e_3 Ast(\mathsf{E}_\vdash \dots (s :: A)td)\\
(\mathsf{E}_\to\, std_1d_2) &:= e_4 Ast(\mathsf{E}_\vdash \dots A(s \to t)d_1)(\mathsf{E}_\vdash \dots Asd_2)\\
(\mathsf{I}_\land\, std_1d_2) &:= e_5 Ast(\mathsf{E}_\vdash \dots Asd_1)(\mathsf{E}_\vdash \dots Atd_2)\\
(\mathsf{E}_\land\, stud_1d_2) &:= e_6 Astu(\mathsf{E}_\vdash \dots A(s \land t)d_1)(\mathsf{E}_\vdash \dots (s :: t :: A)ud_2)\\
(\mathsf{I}^1_\lor\, std) &:= e_7 Ast(\mathsf{E}_\vdash \dots Asd)\\
(\mathsf{I}^2_\lor\, std) &:= e_8 Ast(\mathsf{E}_\vdash \dots Atd)\\
(\mathsf{E}_\lor\, stud_1d_2d_3) &:= e_9 Astu(\mathsf{E}_\vdash \dots A(s \lor t)d_1)\\
&\qquad\qquad (\mathsf{E}_\vdash \dots (s :: A)ud_2)\\
&\qquad\qquad (\mathsf{E}_\vdash \dots (t :: A)ud_3)
\end{aligned}
$$

Figure 24.3: Eliminator for $A \vdash s$

Fact 24.5.1 (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 24.5.2 (Implication) $A \vdash (s \to t) \quad \Leftrightarrow \quad A, s \vdash t$.

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 24.5.3 (Reversion) $A \vdash s \Leftrightarrow \ \vdash A \cdot s$.

Proof By induction on A with s quantified using the implication lemma. ∎

A formula is **ground** if it contains no variable. We assume a recursively defined predicate $\mathsf{ground}\, s$ for groundness.

Fact 24.5.4 (Ground Prover) $\forall s.\ \mathsf{ground}\, s \to (\vdash s) + (\vdash \neg s)$.

Proof By induction on s using weakening. ∎

Design principles

The intuitionistic ND system satisfies the following design principles:

1. Modularity: Every logical constant is accommodated with introduction and elimination rules where the constant appears once and no other constant appears.
2. Weakening: If s can be derived for A, s can be derived for every B containing all formulas in A.
3. Agreement: Implication $\to$ agrees with the external turnstile $\vdash$.

Exercise 24.5.5 Prove $\ \forall s.\ \mathsf{ground}\, s \to \ \vdash (s \vee \neg s)$.

Exercise 24.5.6 Prove $\ \forall A s.\ \mathsf{ground}\, s \to A, s \vdash t \ \to \ A, \neg s \vdash t \ \to \ A \vdash t$.

Exercise 24.5.7 Prove the deduction laws for conjunctions and disjunctions:

a) $A \vdash (s \wedge t) \quad \Leftrightarrow \quad A \vdash s \ \times \ A \vdash t$

b) $A \vdash (s \vee t) \quad \Leftrightarrow \quad \forall u.\ \ A, s \vdash u \ \to \ A, t \vdash u \ \to \ A \vdash u$

Exercise 24.5.8 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 24.5.9 (Order-preserving reversion)
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 $A \vdash s \Leftrightarrow \vdash A \cdot s$.
Hint: Prove the generalization $\forall B.\ B +\!\!\!+ A \vdash s \Leftrightarrow B \vdash A \cdot s$ by induction on A.

24.6 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 24.6.1 (Double Negation) $A \dot\vdash (\neg\neg s \to s)$.

Proof Straightforward using the contradiction rule. ■

Fact 24.6.2 (Cut) $A \dot\vdash s \;\to\; A, s \dot\vdash t \;\to\; A \dot\vdash t$.

Proof Same as for the intuitionistic system. ■

Fact 24.6.3 (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 24.6.4 (Explosion) $A \dot\vdash \bot \to A \dot\vdash s$.

Proof By contradiction and weakening. ■

Fact 24.6.5 (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 24.6.6 (Implication) $A, s \dot\vdash t \Leftrightarrow A \dot\vdash (s \to t)$.

Proof Same proof as for the intuitionistic system. ■

Fact 24.6.7 (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 24.6.8 (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 24.6.9 Show $(A \vdash s \to t \to u) \iff (A \vdash t \to s \to u)$.

Exercise 24.6.10 Show $\dot\vdash s \vee \neg s$ and $\dot\vdash ((s \to t) \to s) \to s$.

Exercise 24.6.11 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 24.6.12 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.

Exercise 24.6.13 (Double negation rule)
A classical ND system can also be obtained with a double negation rule

$$\frac{}{A \dot\vdash \neg\neg s \to s}$$

in place of the contradiction rule.

a) Show that the modified system can derive the contradiction rule.

b) Show that the unmodified system can derive the double negation rule.

c) Show that the two systems are equivalent (that is, derive the same pairs (A, s)).

d) Note that weakening is not required for the equivalence proof.

Modularity with Peirce's rule

The contradiction rule of the classical system violates the modularity principle since it involves 2 logical constants ($\bot$ and $\to$). This design flaw can be fixed by replacing the contradiction rule with the elimination rule for $\bot$ (as in the intuitionistic system) and a rule formulating Peirce's law (§17.1):

$$\mathsf{E}_\bot \; \frac{A \dot\vdash \bot}{A \dot\vdash s} \qquad\qquad \text{Peirce} \; \frac{A,\, s \to t \dot\vdash s}{A \dot\vdash s}$$

Exercise 24.6.14 (Peirce's rule)

a) Show that the system with the contradiction rule can derive Peirce's rule: $\forall Ast.\ (A, s \to t \dot\vdash s) \to (A \dot\vdash s)$.

b) Assume Peirce's rule ($\forall Ast.\ (A, s \to t \dot\vdash s) \to (A \dot\vdash s)$) and derive the contradiction rule ($\forall As.\ (A, \neg s \dot\vdash \bot) \to (A \dot\vdash s)$) using explosion.

c) Model the system with Peirce's rule as a separate inductive type familiy and show that the system with Peirce's rule and $\mathsf{E}_\bot$ can derive the same pairs (A, s) as the system with the contradiction rule.

24.7 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 24.7.1 $A \dot{\vdash} s \;\rightarrow\; A \vdash \neg\neg s$.

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 24.3.3). ■

Theorem 24.7.2 (Glivenko) $A \dot{\vdash} s \;\Leftrightarrow\; A \vdash \neg\neg s$.

Proof Direction $\Rightarrow$ follows with Lemma 24.7.1. Direction $\Leftarrow$ follows with translation (24.6.5) and double negation (24.6.1). ■

Corollary 24.7.3 (Agreement on negated formulas) $A \dot{\vdash} \neg s \;\Leftrightarrow\; A \vdash \neg s$.

Corollary 24.7.4 (Refutation agreement)
Intuitionistic and classical refutation agree: $A \vdash \bot \Leftrightarrow A \dot{\vdash} \bot$.

Proof Glivenko's theorem and the bottom law 24.3.1. ■

Corollary 24.7.5 (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 24.7.4. ■

Exercise 24.7.6 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 $A \dot{\vdash} s \;\Leftrightarrow\; A \vdash s$.

24.8 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

$$\mathsf{H_{MP}}\ \frac{\mathcal{H}(s \to t) \qquad \mathcal{H}(s)}{\mathcal{H}(t)} \qquad\qquad \mathsf{H_K}\ \frac{}{\mathcal{H}(s \to t \to s)}$$

$$\mathsf{H_S}\ \frac{}{\mathcal{H}((s \to t \to u) \to (s \to t) \to s \to u)} \qquad\qquad \mathsf{H_\perp}\ \frac{}{\mathcal{H}(\perp \to s)}$$

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 $\mathsf{H_\perp}$ 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 Gottlob Frege before they were popularized by David Hilbert and coworkers.

$$\mathsf{H}^{\Vdash}_{\mathsf{A}}\ \frac{s \in A}{A \Vdash s} \qquad \mathsf{H}^{\Vdash}_{\mathsf{MP}}\ \frac{A \Vdash s \to t \qquad A \Vdash s}{A \Vdash t} \qquad \mathsf{H}^{\Vdash}_{\mathsf{K}}\ \frac{}{A \Vdash s \to t \to s}$$

$$\mathsf{H}^{\Vdash}_{\mathsf{S}}\ \frac{}{A \Vdash (s \to t \to u) \to (s \to t) \to s \to u} \qquad \mathsf{H}^{\Vdash}_{\bot}\ \frac{}{A \Vdash \bot \to s}$$

Figure 24.4: 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 24.8.1 (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 24.4 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 24.8.2 (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 24.1). The simulation functions are obvious for all rules of the ND system but for $\mathsf{I}_\to$.

Fact 24.8.3 (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 $\mathsf{H}_{\mathsf{A}}^{\Vdash}$ and $\mathsf{H}_{\mathsf{MP}}^{\Vdash}$. Function (3) can be obtained with $\mathsf{H}_{\bot}^{\Vdash}$ and $\mathsf{H}_{\mathsf{MP}}^{\Vdash}$. ■

The translation function for $\mathsf{I}_{\to}$ needs several auxiliary functions.

Fact 24.8.4 (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 $\mathsf{H}_{\mathsf{K}}^{\Vdash}$ and $\mathsf{H}_{\mathsf{MP}}^{\Vdash}$. (2) follows with $\mathsf{H}_{\mathsf{S}}^{\Vdash}$ and $\mathsf{H}_{\mathsf{MP}}^{\Vdash}$. ■

Fact 24.8.5 (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 $\mathsf{H}_{\mathsf{K}}^{\Vdash}$ 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 24.8.6 (Simulation function for $\mathsf{I}_{\to}$) $\forall Ast.\ (A, s \Vdash t) \to (A \Vdash s \to t)$.

Proof By induction on the derivation $A, s \Vdash t$ (the context argument of $\Vdash$ is a uniform parameter).

- $\mathsf{H}_{\mathsf{A}}^{\Vdash}$. If $s = t$, the claim follows with Fact 24.8.5. If $t \in A$, the claim follows with $\mathsf{H}_{\mathsf{A}}^{\Vdash}$ and the operational version of K (Fact 24.8.4(1)). The case distinction is possible since equality of formulas is decidable.
- $\mathsf{H}_{\mathsf{MP}}^{\Vdash}$. Follows with the operational version of S (Fact 24.8.4(2)) and the inductive hypotheses.
- $\mathsf{H}_{\mathsf{K}}^{\Vdash}$, $\mathsf{H}_{\mathsf{S}}^{\Vdash}$, $\mathsf{H}_{\bot}^{\Vdash}$. The axiomatic cases follow with the operational version of K (Fact 24.8.4(1)) and $\mathsf{H}_{\mathsf{K}}^{\Vdash}$, $\mathsf{H}_{\mathsf{S}}^{\Vdash}$, $\mathsf{H}_{\bot}^{\Vdash}$, rspectively. ■

Fact 24.8.7 (Completeness for ND) $(A \vdash s) \to (A \Vdash s)$.

Proof By induction on the derivation of $A \vdash s$ using Facts 24.8.3 and 24.8.6. ■

Theorem 24.8.8 (Agreement) $\mathcal{H}(s) \Leftrightarrow \vdash s$.

Proof Follows with Facts 24.8.1, 24.8.7, and 24.8.2. ■

Exercise 24.8.9 Show $(A \Vdash s) \Leftrightarrow (A \vdash s)$.

Exercise 24.8.10 Extend the development of this section to formulas with conjunctions and disjunctions. Add the axiomatic rules shown at the beginning of § 24.8.

Exercise 24.8.11 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 $\neg\neg s \to s$.

24.9 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:

$$\mathsf{imp}\, ab \;:=\; \textsc{if}\ a \le b\ \textsc{then}\ 2\ \textsc{else}\ b$$

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 $a \le b$. 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:

$$\begin{aligned}
&\mathcal{E} : (\mathsf{N} \to \mathsf{V}) \to \mathsf{For} \to \mathsf{V}\\
&\mathcal{E}\alpha x := \alpha x\\
&\mathcal{E}\alpha\bot := 0\\
&\mathcal{E}\alpha(s \to t) := \textsc{if}\ \mathcal{E}\alpha s \le \mathcal{E}\alpha t\ \textsc{then}\ 2\ \textsc{else}\ \mathcal{E}\alpha t\\
&\mathcal{E}\alpha(s \wedge t) := \textsc{if}\ \mathcal{E}\alpha s \le \mathcal{E}\alpha t\ \textsc{then}\ \mathcal{E}\alpha s\ \textsc{else}\ \mathcal{E}\alpha t\\
&\mathcal{E}\alpha(s \vee t) := \textsc{if}\ \mathcal{E}\alpha s \le \mathcal{E}\alpha t\ \textsc{then}\ \mathcal{E}\alpha t\ \textsc{else}\ \mathcal{E}\alpha s
\end{aligned}$$

[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 §24.8 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 §24.8) 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 24.9.1 (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 24.9.2 (Soundness) $\vdash s \to \mathcal{E}\alpha s = 2$.

Proof Fact 24.9.1 and Theorem 24.8.8. ■

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 24.9.3 (Independence) x, $\neg\neg x \to x$ and $x \vee \neg x$ are independent in $\vdash$.

Proof Follows with Corollary 24.9.2 and the assignment $\lambda\_.1$. ■

Corollary 24.9.4 (Consistency) $\nvdash \bot$ and $\nVdash \bot$.

Proof Intuitionistic consistency follows with Corollary 24.9.2 and the assignment $\lambda\_.1$. Classic consistency follows with equiconsistency (Corollary 24.7.5). ■

Exercise 24.9.5 Show that x, $\neg x$, and $(x \to y) \to x) \to x$ are independent in $\vdash$.

Exercise 24.9.6 Show $\neg\forall s.\ ((\vdash (\neg\neg s \to s)) \to \bot)$.

Exercise 24.9.7 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 24.9.8 Disprove $\dot\vdash x$ and $\dot\vdash \neg x$.

Exercise 24.9.9 Disprove $\dot\vdash (s \vee t) \Leftrightarrow\ \dot\vdash s \vee A \dot\vdash t$.

Exercise 24.9.10 (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 24.9.11 (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*.

24.10 Boolean Evaluation

We define **boolean evaluation** of formulas following familiar ideas:

$$\mathcal{E} : (\mathsf{N} \to \mathsf{B}) \to \mathsf{For} \to \mathsf{B}$$

$$\begin{aligned}
\mathcal{E}\alpha x &:= \alpha x \\
\mathcal{E}\alpha\bot &:= \mathsf{false} \\
\mathcal{E}\alpha(s \to t) &:= \textsc{if}\ \mathcal{E}\alpha s\ \textsc{then}\ \mathcal{E}\alpha t\ \textsc{else}\ \mathsf{true} \\
\mathcal{E}\alpha(s \wedge t) &:= \textsc{if}\ \mathcal{E}\alpha s\ \textsc{then}\ \mathcal{E}\alpha t\ \textsc{else}\ \mathsf{false} \\
\mathcal{E}\alpha(s \vee t) &:= \textsc{if}\ \mathcal{E}\alpha s\ \textsc{then}\ \mathsf{true}\ \textsc{else}\ \mathcal{E}\alpha t
\end{aligned}$$

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 $\mathsf{false} < \mathsf{true}$.

We define

$$\begin{aligned}
\mathsf{sat}\,\alpha\,s &:= (\mathcal{E}\alpha s = \mathsf{true}) && \textbf{α satisfies s} \\
\mathsf{sat}\,\alpha\,A &:= \forall s \in A.\ \mathsf{sat}\,\alpha s && \textbf{α satisfies A} \\
\mathsf{sat}\,A &:= \Sigma\alpha.\ \mathsf{sat}\,\alpha\,A && \textbf{A is satisfiable}
\end{aligned}$$

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.

24.11 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 24.5. 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 24.5: Boolean decomposition table

the possibly negated subformulas on the right. Formally we have the equivalences

$$\begin{array}{rcl}
\mathsf{sat}\,\alpha\,(\neg\bot) & \longleftrightarrow & \top \\
\mathsf{sat}\,\alpha\,(s \wedge t) & \longleftrightarrow & \mathsf{sat}\,\alpha\,s \;\wedge\; \mathsf{sat}\,\alpha\,t \\
\mathsf{sat}\,\alpha\,(\neg(s \wedge t)) & \longleftrightarrow & \mathsf{sat}\,\alpha\,(\neg s) \;\vee\; \mathsf{sat}\,\alpha\,(\neg t) \\
\mathsf{sat}\,\alpha\,(s \vee t) & \longleftrightarrow & \mathsf{sat}\,\alpha\,s \;\vee\; \mathsf{sat}\,\alpha\,t \\
\mathsf{sat}\,\alpha\,(\neg(s \vee t)) & \longleftrightarrow & \mathsf{sat}\,\alpha\,(\neg s) \;\wedge\; \mathsf{sat}\,\alpha\,(\neg t) \\
\mathsf{sat}\,\alpha\,(s \to t) & \longleftrightarrow & \mathsf{sat}\,\alpha\,(\neg s) \;\vee\; \mathsf{sat}\,\alpha\,t \\
\mathsf{sat}\,\alpha\,(\neg(s \to t)) & \longleftrightarrow & \mathsf{sat}\,\alpha\,s \;\wedge\; \mathsf{sat}\,\alpha\,(\neg t)
\end{array}$$

for all boolean assignment α. The equivalences follow with the de Morgan laws

$$\begin{array}{rcl}
\mathcal{E}\alpha(\neg(s \wedge t)) & = & \mathcal{E}\alpha(\neg s \vee \neg t) \\
\mathcal{E}\alpha(\neg(s \vee t)) & = & \mathcal{E}\alpha(\neg s \wedge \neg t)
\end{array}$$

and the implication and double negation laws:

$$\begin{array}{rcl}
\mathcal{E}\alpha(s \to t) & = & \mathcal{E}\alpha(\neg s \vee t) \\
\mathcal{E}\alpha(\neg\neg s) & = & \mathcal{E}\alpha(s)
\end{array}$$

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 24.5). 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 24.6 and 24.7. 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 24.11.1 (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 24.5. 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 24.11.2 (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. ∎

$$\frac{\text{solved}\ A}{\sigma(A)} \qquad \frac{\sigma(A +\!\!+ [s])}{\sigma(s :: A)} \qquad \frac{\sigma(A)}{\sigma(\neg\bot :: A)}$$

$$\frac{\sigma(s :: t :: A)}{\sigma(s \wedge t :: A)} \qquad \frac{\sigma(\neg s :: A)}{\sigma(\neg(s \wedge t) :: A)} \qquad \frac{\sigma(\neg t :: A)}{\sigma(\neg(s \wedge t) :: A)}$$

$$\frac{\sigma(s :: A)}{\sigma(s \vee t :: A)} \qquad \frac{\sigma(t :: A)}{\sigma(s \vee t :: A)} \qquad \frac{\sigma(\neg s :: \neg t :: A)}{\sigma(\neg(s \vee t) :: A)}$$

$$\frac{\sigma(\neg s :: A)}{\sigma(s \to t :: A)} \qquad \frac{\sigma(t :: A)}{\sigma(s \to t :: A)} \qquad \frac{\sigma(s :: \neg t :: A)}{\sigma(\neg(s \to t) :: A)}$$

Figure 24.6: Corefutation system $\sigma(A)$

$$\frac{\text{clashed}\ A}{\rho(A)} \qquad \frac{\rho(A +\!\!+ [s])}{\rho(s :: A)} \qquad \frac{\rho(A)}{\rho(\neg\bot :: A)}$$

$$\frac{\rho(s :: t :: A)}{\rho(s \wedge t :: A)} \qquad \frac{\rho(\neg s :: A) \qquad \rho(\neg t :: A)}{\rho(\neg(s \wedge t) :: A)}$$

$$\frac{\rho(s :: A) \qquad \rho(t :: A)}{\rho(s \vee t :: A)} \qquad \frac{\rho(\neg s :: \neg t :: A)}{\rho(\neg(s \vee t) :: A)}$$

$$\frac{\rho(\neg s :: A) \qquad \rho(t :: A)}{\rho(s \to t :: A)} \qquad \frac{\rho(s :: \neg t :: A)}{\rho(\neg(s \to t) :: A)}$$

Figure 24.7: Refutation system $\rho(A)$

Fact 24.11.3 (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 24.11.4 (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 24.5.2). it suffices to show $\dot{\vdash} \neg\neg s \to \neg t \to \neg(s \to t)$, which is routine. ∎

Exercise 24.11.5 Define the derivation systems σ and ρ as inductive type families and say whether the arguments are parameters or an indices.

Exercise 24.11.6 Verify the proof of ND soundness (Lemma 24.11.4) in detail. The proof is modular in that there is a separate proof obligation for every rule of the refutation systems (Figure 24.7). 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 24.5.2) and the derivability of the ND judgments from Exercise 24.5.8.

24.12 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 24.12.1 (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 24.5) 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 24.12.2 (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 24.12.3 (Basic certifying solver)
$\forall A.\ \sigma(A) + \rho(A).$

Proof By size induction 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 24.12.4 (ND solver) $\forall A.\ \mathsf{sat}\,A + (A \dot{\vdash} \bot)$.

Proof Lemma 24.12.3 and soundness theorems. ∎

Exercise 24.12.5 (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 24.12.6 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.

24.13 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 24.13.1 (ND soundness) $(A \dot{\vdash} s) \to (A \dot{\vDash} s)$.

Proof By induction on the derivation $A \dot{\vdash} s$. ∎

Fact 24.13.2 (ND completeness) $(A \dot{\vDash} s) \to (A \dot{\vdash} s)$.

Proof We assume $A \dot{\vDash} s$. Using the ND solver (Theorem 24.12.4), 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 24.13.3 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 24.13.4 (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 $\mathcal{E}\alpha s = \mathsf{true}$. We now assume $\mathcal{E}\alpha s = \mathsf{false}$ and obtain a contradiction from our assumptions since $\mathcal{E}\alpha(\neg s) = \mathsf{true}$. ■

Fact 24.13.5 (ND decidability) $\forall As.\ \mathcal{D}(A \dot\vdash s)$.

Proof With the ND solver (Theorem 24.12.4) 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 24.13.1) and Fact 24.13.4. ■

Exercise 24.13.6 Note that the results in this section did not use results from the previous two sections except for the ND solver (Theorem 24.12.4). Prove the following facts using the results from this section and possibly the ND solver.

a) $\forall As.\ \mathcal{D}(A \dot\vDash s)$

b) $\forall A.\ \mathcal{D}(\mathsf{sat}\,A)$

c) $\forall A.\ \mathsf{sat}\,A \Leftrightarrow ((A \dot\vdash \bot) \to \bot)$

Exercise 24.13.7 Give a consistency proof for classical ND that does not make use of intuitionistic ND.

Exercise 24.13.8 Show that x and $\neg x$ are independent in $\dot\vdash$.

Exercise 24.13.9 Show that $\neg\neg\neg x$ is independent in $\dot\vdash$.

Exercise 24.13.10 Show $(\forall st.\ \dot\vdash(s \vee t) \to (\dot\vdash s) \vee (\dot\vdash t)) \to \bot$.

24.14 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 [28] to know more.

Figure 24.8 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

$$\frac{\bot \in A}{\gamma(A)} \qquad\qquad \frac{s \in A \qquad \neg s \in A}{\gamma(A)}$$

$$\frac{(s \wedge t) \in A \qquad \gamma(s :: t :: A)}{\gamma(A)} \qquad\qquad \frac{\neg(s \wedge t) \in A \qquad \gamma(\neg s :: A) \qquad \gamma(\neg t :: A)}{\gamma(A)}$$

$$\frac{(s \vee t) \in A \qquad \gamma(s :: A) \qquad \gamma(t :: A)}{\gamma(A)} \qquad\qquad \frac{\neg(s \vee t) \in A \qquad \gamma(\neg s :: \neg t :: A)}{\gamma(A)}$$

$$\frac{(s \to t) \in A \qquad \gamma(\neg s :: A) \qquad \gamma(t :: A)}{\gamma(A)} \qquad\qquad \frac{\neg(s \to t) \in A \qquad \gamma(s :: \neg t :: A)}{\gamma(A)}$$

Figure 24.8: 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 24.14.1 (Boolean soundness)
$\gamma(A) \to \exists s \in A.\ \mathcal{E}\alpha s = \mathsf{false}$.

Proof By induction on the derivation $\gamma(A)$. Similar to the proof of Fact 24.11.3. ■

Fact 24.14.2 (Weakening)
$\gamma(A) \to A \subseteq B \to \gamma(B)$.

Proof By induction on $\gamma(A)$ with B quantified. ■

Fact 24.14.3 (Completeness)
$\rho(A) \to \gamma(A)$.

Proof Straightforward using weakening. ■

Fact 24.14.4 (Agreement)
$\rho(A) \Leftrightarrow \gamma(A)$.

Proof Completeness (Fact 24.14.3), certifying boolean solver for ρ (Theorem 24.12.3), and boolean soundness (Fact 24.14.1). ■

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 24.14.5 Refute the negations of the following formulas with the cumulative refutation system writing a table 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 24.14.6 (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 $\perp$ if it satisfies all *atomic formulas* (x and $\neg x$) in A.
Hint: Prove

$$\forall s.\ (s \in A \to \mathcal{E}\alpha s = \mathsf{true}) \wedge (\neg s \in A \to \mathcal{E}\alpha(\neg s) = \mathsf{true})$$

by induction on s.

24.15 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:

$$\begin{aligned}
\theta\cdot x &:= \theta x\\
\theta\cdot\bot &:= \bot\\
\theta\cdot(s \to t) &:= \theta\cdot s \to \theta\cdot t\\
\theta\cdot(s \wedge t) &:= \theta\cdot s \wedge \theta\cdot t\\
\theta\cdot(s \vee t) &:= \theta\cdot s \vee \theta\cdot t\\[1ex]
\theta\cdot[] &:= []\\
\theta\cdot(s :: A) &:= \theta\cdot s :: \theta\cdot A
\end{aligned}$$

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 24.15.1 $s \in A \to \theta s \in \theta A$.

Proof By induction on A. ■

Fact 24.15.2 (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 24.15.1 for the assumption rule. ■

Exercise 24.15.3 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.

24.16 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 24.9. 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:* $A \Vdash (s \land t) \;\longleftrightarrow\; A \Vdash s \;\land\; A \Vdash t$.
9. *Disjunction:* $A \Vdash (s \lor t) \;\longleftrightarrow\; \forall u.\ A, s \Vdash u \;\to\; A, t \Vdash u \;\to\; A \Vdash u$.

Figure 24.9: Requirements for entailment relations

Fact 24.16.1 Intuitionistic provability ($A \vdash s$) and classical provability ($A \dot\vdash s$) are entailment relations.

Proof Follows with the results shown so far. ∎

Fact 24.16.2 Boolean entailment $A \dot\vDash s$ is an entailment relation.

Proof Follows with Fact 24.16.1 since boolean entailment agrees with classical ND (Fact 24.13.3). ∎

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 24.16.3 (Modus Ponens) $A \Vdash (s \to t) \;\to\; A \Vdash s \;\to\; A \Vdash t$.

Proof By implication and cut. ∎

Fact 24.16.4 (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 24.16.5 $\Vdash s \;\to\; \Vdash \neg s \;\to\; \bot$.

Proof Let $\Vdash s$ and $\Vdash \neg s$. By Fact 24.16.3 we have $\Vdash \bot$. By consistency and explosion we obtain a contradiction. ∎

Fact 24.16.6 (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 24.16.7 (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 24.16.3 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 24.16.8 $\Vdash s \to \dot{\vDash} s$.

Proof Let $\Vdash s$. We assume $\mathcal{E}\alpha s = \mathsf{false}$ 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 24.16.5. ■

Fact 24.16.9 (Greatest entailment relation)
Boolean entailment is a greatest entailment relation: $A \Vdash s \to A \dot{\vDash} s$.

Proof Follows with reversion (Facts 24.16.6 and 24.16.2) and Lemma 24.16.8. ■

Theorem 24.16.10 (Sandwich) Every entailment relation $\Vdash$ satisfies $\vdash \subseteq \Vdash \subseteq \dot{\vdash}$.

Proof Facts 24.16.4, 24.16.9, and 24.13.3. ■

Exercise 24.16.11 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 24.16.12 Tebbi's lemma provides for a particularly elegant proof of Lemma 24.16.8. Verify that Lemma 24.16.8 can also be obtained from the facts (1) $\vdash \hat{\alpha}s \vee \vdash \neg\hat{\alpha}s$ and (2) $\dot{\vDash} \hat{\alpha}s \to \mathcal{E}\alpha s = \mathsf{true}$ using Facts 24.16.4 and 24.16.5.

24.17 Notes

The study of natural deduction originated in the 1930's with the work of Gerhard Gentzen [14, 15] and Stanisław Jaśkowski [19]. The standard text on natural deduction and proof theory is Troelstra and Schwichtenberg [28].

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.

25 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).

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

$$\begin{array}{ccc} \mathsf{true}\ \&\ b := b & \mathsf{true}\ |\ b := \mathsf{true} & !\,\mathsf{true} := \mathsf{false} \\ \mathsf{false}\ \&\ b := \mathsf{false} & \mathsf{false}\ |\ b := b & !\,\mathsf{false} := \mathsf{true} \end{array}$$

With these definitions, boolean identities like

$$a\ \&\ b = b\ \&\ a \qquad\qquad a\ |\ b = b\ |\ a \qquad\qquad !\,!\,b = b$$

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*

$$!(a\ \&\ b) = \,!a\ |\ !b \qquad\qquad !(a\ |\ b) = \,!a\ \&\ !b \qquad\qquad !\,!\,a = a$$

and conjunctions can be pushed below disjunctions with the *distribution law*

$$a\ \&\ (b\ |\ c) = (a\ \&\ b)\ |\ (a\ \&\ b)$$

Besides the defining equations, we will also make use of the *negation law*

$$b \wedge !b = \mathsf{false}$$

to eliminate conjunctions.

There are the **reflection laws**

$$\begin{aligned} a \mathbin{\&} b = \mathsf{true} &\longleftrightarrow a = \mathsf{true} \wedge b = \mathsf{true} \\ a \mid b = \mathsf{true} &\longleftrightarrow a = \mathsf{true} \vee b = \mathsf{true} \\ !a = \mathsf{true} &\longleftrightarrow \neg(a = \mathsf{true}) \end{aligned}$$

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 true.

25.2 Boolean Formulas

Our main interest will be in boolean formulas, which are syntactic representations of boolean terms. We will consider the boolean **formulas**

$$s, t, u : \mathsf{For} ::= x \mid \bot \mid s \to t \mid s \wedge t \mid s \vee t \qquad (x : \mathsf{N})$$

realized with an inductive data type For representing each syntactic form with a value constructor. **Variables** x are represented as numbers. 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

$$\top := \bot \to \bot \qquad\qquad \neg s := s \to \bot$$

Given an **assignment** $\alpha : \mathsf{N} \to \mathsf{B}$, we can evaluate every formula to a boolean value. We formalize evaluation of formulas with the **evaluation function** shown in Figure 25.1. 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

$$\alpha \mathsf{sat} s := \mathcal{E}\alpha s = \mathsf{true}$$

$$\begin{aligned}
\mathcal{E}\alpha x &:= \alpha x \\
\mathcal{E}\alpha\bot &:= \mathsf{false} \\
\mathcal{E}\alpha(s \to t) &:= \,!\mathcal{E}\alpha s \mid \mathcal{E}\alpha t \\
\mathcal{E}\alpha(s \wedge t) &:= \mathcal{E}\alpha s \,\&\, \mathcal{E}\alpha t \\
\mathcal{E}\alpha(s \vee t) &:= \mathcal{E}\alpha s \mid \mathcal{E}\alpha t
\end{aligned}$$

Figure 25.1: 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 $\mathcal{E}\alpha s = \mathsf{true}$. Reasoning with boolean equations will be the main workhorse in our proofs.

Exercise 25.2.1 Prove that $s \to t$ and $\neg s \vee t$ are equivalent.

Exercise 25.2.2 Convince yourself that the predicate $\alpha \mathsf{sat} s$ is decidable.

Exercise 25.2.3 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 25.2.4 (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 25.2.5 (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 25.2.6 (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 25.2.7 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 §25.1.

25.3 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:

$$S, T \ :\ \mathsf{SFor} \ ::= \ s^+ \mid s^- \qquad\qquad \textbf{signed formula}$$
$$C, D \ :\ \mathsf{Cla} \ := \ \mathcal{L}(\mathsf{SFor}) \qquad\qquad \textbf{clause}$$

Clauses represent conjunctions. We define evaluation of signed formulas and clauses as follows:

$$\mathcal{E}\alpha(s^+) := \mathcal{E}\alpha s \qquad\qquad \mathcal{E}\alpha\,[] := \mathsf{true}$$
$$\mathcal{E}\alpha(s^-) := \,!\mathcal{E}\alpha s \qquad\qquad \mathcal{E}\alpha(S :: C) := \mathcal{E}\alpha S \mathbin{\&} \mathcal{E}\alpha C$$

Note that the empty clause represents the boolean true. We also consider lists of clauses

$$\Delta \ :\ \mathcal{L}(\mathsf{Cla})$$

and interpret them disjunctively:

$$\mathcal{E}\alpha\,[] := \mathsf{false}$$
$$\mathcal{E}\alpha\,(C :: \Delta) := \mathcal{E}\alpha C \mid \mathcal{E}\alpha\Delta$$

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 $\mathsf{sat}\ C$. Since formulas, signed formulas, clauses, and lists of clauses all come with the notion of satisfying assignments, we can speak about **equivalence** between these objects although they belong to different types. For instance, s, s^+, $[s^+]$, and $[[s^+]]$, are all equivalent since they are satisfied by the same assignments.

A **solved clause** is a clause consisting of signed variables (i.e., x^+ and x^-) such that no variable appears positively and negatively. Note that a solved clause C is satisfied by every assignment that maps the positive variables in C to true and the negative variables in C to false.

Fact 25.3.1 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 25.3.2 A clausal DNF is satisfiable if and only if it is nonempty.

Exercise 25.3.3 Prove $\mathcal{E}\alpha(C + D) = \mathcal{E}\alpha C \mathbin{\&} \mathcal{E}\alpha D$ and $\mathcal{E}\alpha(\Delta + \Delta') = \mathcal{E}\alpha\Delta \mid \mathcal{E}\alpha\Delta'$.

Exercise 25.3.4 Write a function that maps lists of clauses to equivalent formulas.

Exercise 25.3.5 Our formal proof of Fact 25.3.1 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. $\mathsf{solved}\ C \to S \in C \to \exists x.\ (S = x^+ \wedge x^- \notin C) \vee (S = x^- \wedge x^+ \notin C)$.

The formal development captures solved clauses with an inductive predicate. This is convenient for most purposes but doesn't provide for a convenient proof of Fact 25.3.1. Can you do better?

25.4 DNF Solver

We would like to construct a function computing clausal DNFs for formulas. Formally, we specify the function with the informative type

$$\forall s \Sigma \Delta.\ \ \mathsf{DNF}\,\Delta \wedge s \equiv \Delta$$

where

$$\begin{aligned} s \equiv \Delta \ &:=\ \forall \alpha.\ \alpha \mathsf{sat} s \longleftrightarrow \alpha \mathsf{sat} \Delta \\ \mathsf{DNF}\,\Delta \ &:=\ \forall C \in \Delta.\ \mathsf{solved}\, C \end{aligned}$$

To define the function, we will generalize the type to

$$\forall CD.\ \mathsf{solved}\, C \to \Sigma \Delta.\ \ \mathsf{DNF}\,\Delta \wedge C + D \equiv \Delta$$

$$\begin{array}{rcl}
\mathsf{dnf}\ C\ [] & = & [C] \\
\mathsf{dnf}\ C\ (x^+ :: D) & = & \text{IF } \ulcorner x^- \in C \urcorner \text{ THEN } []\ \text{ELSE } \mathsf{dnf}\ (x^+ :: C)\ D \\
\mathsf{dnf}\ C\ (x^- :: D) & = & \text{IF } \ulcorner x^+ \in C \urcorner \text{ THEN } []\ \text{ELSE } \mathsf{dnf}\ (x^- :: C)\ D \\
\mathsf{dnf}\ C\ (\bot^+ :: D) & = & [] \\
\mathsf{dnf}\ C\ (\bot^- :: D) & = & \mathsf{dnf}\ C\ D \\
\mathsf{dnf}\ C\ ((s \to t)^+ :: D) & = & \mathsf{dnf}\ C\ (s^- :: D) +\!\!\!+ \mathsf{dnf}\ C\ (t^+ :: D) \\
\mathsf{dnf}\ C\ ((s \to t)^- :: D) & = & \mathsf{dnf}\ C\ (s^+ :: t^- :: D) \\
\mathsf{dnf}\ C\ ((s \wedge t)^+ :: D) & = & \mathsf{dnf}\ C\ (s^+ :: t^+ :: D) \\
\mathsf{dnf}\ C\ ((s \wedge t)^- :: D) & = & \mathsf{dnf}\ C\ (s^- :: D) +\!\!\!+ \mathsf{dnf}\ C\ (t^- :: D) \\
\mathsf{dnf}\ C\ ((s \vee t)^+ :: D) & = & \mathsf{dnf}\ C\ (s^+ :: D) +\!\!\!+ \mathsf{dnf}\ C\ (t^+ :: D) \\
\mathsf{dnf}\ C\ ((s \vee t)^- :: D) & = & \mathsf{dnf}\ C\ (s^- :: t^- :: D)
\end{array}$$

Figure 25.2: 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 25.2. We refer to the first argument C of the procedure as **accumulator**, and to the second argument as **agenda**. The agenda holds the signed formulas still to be processed, and the accumulator collects signed variables taken from the agenda. The procedure processes the formulas on the agenda one by one decreasing the size of the agenda with every recursion step. We define the **size** of clauses and formulas as follows:

$$\begin{array}{rclcrcl}
\sigma[] & := & 0 & \qquad & \sigma x & := & 1 \\
\sigma(s^+ :: C) & := & \sigma s + \sigma C & & \sigma\bot & := & 1 \\
\sigma(s^- :: C) & := & \sigma s + \sigma C & & \sigma(s \circ t) & := & 1 + \sigma s + \sigma t
\end{array}$$

Note that the equations specifying the procedure in Figure 25.2 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 §25.1.

Lemma 25.4.1 $\forall CD.\ \mathsf{solved}\, C \to \Sigma\Delta.\ \ \mathsf{DNF}\,\Delta \wedge C +\!\!\!+ D \equiv \Delta$.

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 25.4.2 (DNF solver) $\forall C\, \Sigma\Delta.\ \mathsf{DNF}\,\Delta \wedge C \equiv \Delta$.

Proof Immediate from Lemma 25.4.1. ∎

Corollary 25.4.3 $\forall s\, \Sigma\Delta.\ \mathsf{DNF}\,\Delta \wedge s \equiv \Delta$.

Corollary 25.4.4 There is a solver $\forall C.\ (\Sigma\alpha.\ \alpha \mathsf{sat} C) + \neg \mathsf{sat}\ C$.

Corollary 25.4.5 There is a solver $\forall s.\ (\Sigma\alpha.\ \alpha \mathsf{sat} s) + \neg \mathsf{sat}\ s$.

Corollary 25.4.6 Satisfiability of clauses and formulas is decidable.

Exercise 25.4.7 Convince yourself that the predicate $S \in C$ is decidable.

Exercise 25.4.8 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 25.4.9 Recall the definition of valid formulas from Exercise 25.2.6. Prove the following:

a) Validity of formulas is decidable.

b) A formula is satisfiable if and only if its negation is not valid.

c) $\forall s.\ \mathsf{val}\ s + (\Sigma\alpha.\ \mathcal{E}\alpha s = \mathsf{false})$.

Exercise 25.4.10 If you are already familiar with well-founded recursion in computational type theory (Chapter 30), define a function $\mathsf{Cla} \to \mathsf{Cla} \to \mathcal{L}(\mathsf{Cla})$ satisfying the equations specifying the procedure dnf in Figure 25.2.

25.5 DNF Recursion

From the equations for the DNF procedure (Figure 25.2) and the construction of the basic DNF solver (Lemma 25.4.1) one can abstract out the recursion scheme shown in Figure 25.3. We refer to this recursion scheme as **DNF recursion**. DNF recursion has one clause for every equation of the DNF procedure in Figure 25.2 where the recursive calls appear as inductive hypotheses. DNF recursion simplifies the proof of Lemma 25.4.1. However, DNF recursion can also be used for other constructions (our main example is a completeness lemma (25.6.5) for a tableau system) given that it is formulated with an abstract type function p. Note that DNF recursion encapsulates the use of size induction 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 25.3: DNF recursion scheme

Lemma 25.5.1 (DNF recursion)
The DNF recursion scheme shown in Figure 25.3 is inhabited.

Proof By size induction 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 25.5.2 Use DNF recursion to construct a certifying boolean solver for clauses: $\forall C.\ (\Sigma\alpha.\ \alpha \mathsf{sat} C) + (\neg \mathsf{sat}(C))$.

$$\frac{\mathsf{tab}(S :: C \mathbin{+\!\!+} D)}{\mathsf{tab}(C \mathbin{+\!\!+} S :: D)} \qquad \frac{}{\mathsf{tab}(x^+ :: x^- :: C)} \qquad \frac{}{\mathsf{tab}(\bot^+ :: C)}$$

$$\frac{\mathsf{tab}(s^- :: C) \qquad \mathsf{tab}(t^+ :: C)}{\mathsf{tab}((s \to t)^+ :: C)} \qquad \frac{\mathsf{tab}(s^+ :: t^- :: C)}{\mathsf{tab}((s \to t)^- :: C)}$$

$$\frac{\mathsf{tab}(s^+ :: t^+ :: C)}{\mathsf{tab}((s \wedge t)^+ :: C)} \qquad \frac{\mathsf{tab}(s^- :: C) \qquad \mathsf{tab}(t^- :: C)}{\mathsf{tab}((s \wedge t)^- :: C)}$$

$$\frac{\mathsf{tab}(s^+ :: C) \qquad \mathsf{tab}(t^+ :: C)}{\mathsf{tab}((s \vee t)^+ :: C)} \qquad \frac{\mathsf{tab}(s^- :: t^- :: C)}{\mathsf{tab}((s \vee t)^- :: C)}$$

Figure 25.4: Inductive type family tab : Cla → $\mathbb{T}$

25.6 Tableau Refutations

Figure 25.4 defines an indexed inductive type family tab : Cla → $\mathbb{T}$ 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 25.4 as a simplification of the equations specifying the DNF procedure in Figure 25.2. 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 25.6.1 (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 25.6.2 (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 25.6.3 (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 25.6.4 (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 25.6.5 (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 25.6.6
A clause is tableau refutable if and only if it is unsatisfiable:
$\mathsf{tab}(C) \Leftrightarrow \neg\mathsf{sat}(C)$.

Proof Follows with Fact 25.6.1 and Lemma 25.6.5. ■

Corollary 25.6.7 $\forall C.\ \mathsf{tab}(C) + (\mathsf{tab}(C) \to \bot)$.

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 25.6.8 Prove $\mathsf{tab}(C + S :: D + T :: E) \longleftrightarrow \mathsf{tab}(C + T :: D + S :: E)$.

Exercise 25.6.9 Give an inductive type family deriving exactly the satisfiable clauses. Start with an inductive family deriving exactly the solved clauses.

25.7 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 24).

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 25.5: Rules for abstract refutation systems $\rho : \mathcal{L}(\mathsf{For}) \to \mathbb{P}$

Fact 25.7.1 $\mathcal{E}\alpha\hat{C} = \mathcal{E}\alpha C$, $\mathcal{E}\alpha A^+ = \mathcal{E}\alpha A$, and $\widehat{A^+} = A$.

Fact 25.7.2 (Decidability) Satisfiability of unsigned clauses is decidable.

Proof Follows with Corollary 25.4.6 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 25.5. 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 25.7.3 Let ρ be a refutation system. Then $\mathsf{tab}\ C \to \rho\hat{C}$.

Proof Straightforward by induction on $\mathsf{tab}\ C$. ■

Fact 25.7.4 (Completeness)
Every refutation system derives all unsatisfiable unsigned clauses.

Proof Follows with Theorem 25.6.6 and Lemma 25.7.3. ■

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 25.7.5 A sound refutation system is decidable and derives exactly the unsatisfiable unsigned clauses.

Proof Facts 25.7.4 and 25.7.2. ■

Theorem 25.7.6 The minimal refutation system inductively defined with the rules for abstract refutation systems derives exactly the unsatisfiable unsigned clauses.

Proof Follows with Fact 25.7.4 and a soundness lemma similar to Fact 25.6.1. ∎

Exercise 25.7.7 (Certifying Solver) Construct a function $\forall A.\ (\Sigma\alpha.\ \alpha\mathsf{sat}A) + \mathsf{tab}\,A$.

Exercise 25.7.8 Show that boolean entailment

$$A \mathrel{\dot{\vDash}} s \;:=\; \forall\alpha.\ \alpha\mathsf{sat}A \;\to\; \alpha\mathsf{sat}s$$

is decidable.

Exercise 25.7.9 Let $A \mathrel{\dot{\vdash}} s$ be the inductive type family for classical natural deduction. Prove that $A \mathrel{\dot{\vdash}} s$ is decidable and agrees with boolean entailment. Hint: Exploit refutation completeness and show that $A \mathrel{\dot{\vdash}} \bot$ is a refutation system.

26 Regular Expression Matching

We consider regular expressions describing list of numbers. The expressions are formed with constructors for singleton lists, concatenation, star concatenation, and union, among others. Using derivatives, we show that regular expression matching is decidable.

26.1 Basics

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

$$s \subseteq s^* \tag{26.1}$$

(only constructor applications and rewriting with $A \mathbin{+\!\!+} [] = A$ are needed). More challenging is the inclusion

$$s^* \cdot s^* \subseteq s^* \tag{26.2}$$

We need an inversion function

$$A \vdash s \cdot t \to \Sigma A_1 A_2.\ (A = A_1 \mathbin{+\!\!+} A_2) \times (A_1 \vdash s) \times (A_2 \vdash t) \tag{26.3}$$

and a lemma

$$A \vdash s^* \ \to\ B \vdash s^* \ \to\ A \mathbin{+\!\!+} B \vdash s^* \tag{26.4}$$

The inversion function can be obtained as an instance of a more general **inversion operator**

$$\begin{array}{ll} \forall As.\ A \vdash s \ \to & \text{MATCH } s \\ & [\, x \Rightarrow A = [x] \\ & \mid \mathbf{0} \Rightarrow \bot \\ & \mid \mathbf{1} \Rightarrow A = [] \\ & \mid u + v \Rightarrow (A \vdash u) + (A \vdash v) \\ & \mid u \cdot v \Rightarrow \Sigma A_1 A_2.\ (A = A_1 \mathbin{+\!\!+} A_2) \times (A_1 \vdash u) \times (A_2 \vdash v) \\ & \mid u^* \Rightarrow (A = []) + \Sigma A_1 A_2.\ (A = A_1 \mathbin{+\!\!+} A_2) \times (A_1 \vdash u) \times (A_2 \vdash u^*) \\ &] \end{array}$$

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 (26.4). 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 $A = A_1 \mathbin{+\!\!+} A_2$, $A_1 \vdash s$, and $A_2 \vdash s^*$. Since $A_2 \vdash s^*$ is obtained by a subderivation, the inductive hypothesis gives us $A_2 \mathbin{+\!\!+} B \vdash s^*$. Hence $A_1 \mathbin{+\!\!+} A_2 \mathbin{+\!\!+} B \vdash s^*$ 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:

$$\forall As.\ A \vdash s \ \to\ \text{MATCH } s \ [\, s^* \Rightarrow B \vdash s^* \to A \mathbin{+\!\!+} B \vdash s^* \mid \_ \Rightarrow \top \,]$$

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 (26.2). A straightforward consequence is

$$s^* \cdot s^* \equiv s^*$$

A less obvious consequence is the equivalence

$$(s^*)^* \equiv s^* \tag{26.5}$$

saying that the star operation is idempotent. Given (26.1), it suffices to show

$$A \vdash (s^*)^* \to A \vdash s^* \tag{26.6}$$

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 (26.4).

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 26.1.1 (Certifying solver)
Define a certifying solver $\forall s.\ (\Sigma A.\ A \vdash s) + (\forall A.\ A \vdash s \to \bot)$.

Exercise 26.1.2 (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 $A \mathrel{\dot{\vdash}} s$ adopting the more restrictive rule and show that it is intertranslatable with $A \vdash s$: $\ \forall As.\ A \mathrel{\dot{\vdash}} s \Leftrightarrow A \vdash s$.

Exercise 26.1.3 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 26.1.4 (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:

$$\begin{aligned}
\mathcal{R}\,x\,A &:= (A = [x]) \\
\mathcal{R}\,\mathbf{0}\,A &:= \bot \\
\mathcal{R}\,\mathbf{1}\,A &:= (A = []) \\
\mathcal{R}\,(s+t)\,A &:= \mathcal{R}sA + \mathcal{R}tA \\
\mathcal{R}\,(s\cdot t)\,A &:= \Sigma A_1A_2.\ (A = A_1 \mathbin{+\!\!+} A_2) \times \mathcal{R}sA_1 \times \mathcal{R}tA_2 \\
\mathcal{R}\,(s^*)\,A &:= \Sigma n.\ \mathcal{P}\,(\mathcal{R}s)nA \\[1ex]
\mathcal{P}\,\varphi\,0\,A &:= (A = []) \\
\mathcal{P}\,\varphi\,(\mathsf{S}n)\,A &:= \Sigma A_1A_2.\ (A = A_1 \mathbin{+\!\!+} A_2) \times \varphi A_1 \times \mathcal{P}\,\varphi\,n\,A
\end{aligned}$$

a) Prove $\mathcal{R}\,sA \Leftrightarrow A \vdash s$.

b) We have represented languages as type functions $\mathcal{L}(\mathsf{N}) \to \mathbb{T}$. A representation as predicates $\mathcal{L}(\mathsf{N}) \to \mathbb{P}$ would be more faithful to the literature. Rewrite the definitions of $\vdash$ and $\mathcal{R}$ accordingly and show their equivalence.

26.2 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

$$\forall xAs.\ \ x :: A \vdash s \Leftrightarrow A \vdash Dxs$$

In words we may say that a string $x :: A$ satisfies a regular expression s if and only if A satisfies the **derivative** Dxs. If we have a decider $\forall s.\ \mathcal{D}([] \vdash s)$ and in addition a derivation function, we have a decider for $A \vdash s$.

Fact 26.2.1 $\forall s.\ \mathcal{D}([] \vdash s)$.

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 26.2.2 $\forall As.\ \mathcal{D}(A \vdash s)$ provided we have a derivation function.

Proof By recursion on A using Fact 26.2.1 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 26.2.3 (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 26.2.4 (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 26.2.3. ■

Corollary 26.2.5 $\forall As.\ \mathcal{D}(A \vdash s).$

Proof Follows with Fact 26.2.2 and Theorem 26.2.4. ■

27 Abstract Reduction Systems

Warning: This chapter is under construction.

27.1 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 27.1.1 (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:

$$\begin{aligned}
\forall x^{\mathsf{N}}.\ \mathsf{len}\,x\,x\,(\mathsf{P}_1\,x)\ &=\ 0\\
\forall xx'y^{\mathsf{N}}\,\forall r^{Rxx'}\,\forall a^{\mathsf{path}\,x'y}.\ \mathsf{len}\,x\,y\,(\mathsf{P}_2\,x\,x'\,y\,r\,a)\ &=\ \mathsf{S}(\mathsf{len}\,x'\,y\,a)
\end{aligned}$$

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:

$$\begin{aligned} \mathsf{app}\,zx\,\_\,(\mathsf{P_1}\,\_) &:= \lambda b.b & &: \ \mathsf{path}\,xz \to \mathsf{path}\,xz \\ \mathsf{app}\,zx\,\_\,(\mathsf{P_2}\,\_\,x'yra) &:= \lambda b.\,\mathsf{P_2}\,xx'zr(\mathsf{app}\,zx'y\,ab) & &: \ \mathsf{path}\,yz \to \mathsf{path}\,xz \end{aligned}$$

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 $\mathsf{P_2}$ gives us the proof obligation

$$\mathsf{S}(\mathsf{len}\,(\mathsf{app}\,ab)) = \mathsf{S}(\mathsf{len}\,a + \mathsf{len}\,b)$$

which follows by the inductive hypothesis, Formally, the induction can be validated with the universal eliminator for path:

$$\begin{aligned} E: \ &\forall p^{\forall xy.\ \mathsf{path}\,xy \to \mathbb{T}}. \\ &(\forall x.\ pxx(\mathsf{P_1}\,x)) \to \\ &(\forall xyz\,\forall r^{Rxy}\,\forall a^{\mathsf{path}\,yz}.\ pxz(\mathsf{P_2}\,xyzra)) \to \\ &\forall xya.\ pxya \end{aligned}$$

$$\begin{aligned} E\,pe_1e_2\,x\,\_\,,(\mathsf{P_1}\,\_) &:= e_1x \\ E\,pe_1e_2\,x\,\_\,(\mathsf{P_2}\,\_\,x'y\,r\,a) &:= e_2\,xx'y\,r(E\,pe_1e_2\,x'y\,a) \end{aligned}$$

Not that the type function p takes the nonuniform parameter, the index, and the discriminating argument as arguments. The general rule to remember here is that all nonuniform parameters and all indices appear as arguments of the target 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 target 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 27.1.2 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 27.1.3 Define the step function asserted by Fact 27.1.1 with a term.

Exercise 27.1.4 (Index eliminator) Define an index eliminator for path:

$$\begin{aligned}
&\forall p^{X \to X \to \mathbb{T}}.\\
&(\forall x.\, pxx) \to\\
&(\forall xx'y.\, Rxx' \to px'y \to pxy) \to\\
&(\forall xy.\, \mathsf{path}\, xy \to pxy)
\end{aligned}$$

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 27.1.5 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 27.1.6 (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 27.1.4). Direction $\Leftarrow$ follows with $\forall k.\, \mathsf{path}_R\, x(k + x)$, which follows by induction on k with x quantified.

27.2 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 27.1.1.

Fact 27.2.1 (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 27.1.4). 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 27.2.1 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 27.2.2 R^* and $(R^*)^*$ agree.

Proof We have $R^* \subseteq (R^*)^*$ by Fact 27.1.1. For the other direction $(R^*)^* \subseteq R^*$ we use star recursion (Fact 27.2.1). 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 27.2.3 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 27.2.2 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 27.2.4 (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

Foundational Studies

28 Axiom CT and Semidecidability

All functions definable in CTT without assumptions are computable. Nevertheless, there are noncomputational interpretations of CTT where the function type $\mathsf{N} \to \mathsf{N}$ contains uncomputable functions. Moreover, there are degenerate interpretations of CTT where all predicates $\mathsf{N} \to \mathbb{P}$ are decidable. Consequently, CTT can only prove undecidability if we have an assumption excluding degenerate interpretations. In this chapter, we study such an assumption called Axiom CT. Axiom CT is consistent with excluded middle and extensionality and provides for undecidability proofs within CTT.

We will base Axiom CT on the diophantine characterization of recursively enumerable sets, which has a straightforward formalization in CTT. Axiom CT will say that for every function $f^{\mathsf{N}\to\mathsf{N}\to\mathsf{B}}$ there is a diophantine expression describing the predicate $\lambda n.\, \exists k.\, fnk = \mathsf{true}$.

We will give a self-contained development of type-theoretic computability not assuming knowledge of set-theoretic computability theory. The main notions in this enterprise are semidecidable predicates and promising functions. While semidecidable predicates model recursively enumerable sets, promising functions model computable set-theoretic functions as step-indexed type-theoretic functions. Assuming Axiom CT, we will construct undecidable semidecidable predicates and promising functions not having total extensions.

Remarkably, our primary development does not use excluded middle. If we assume excluded middle, we can characterize decidable predicates as semidecidable predicates that are cosemidecidable. As it turns out, this characterization is equivalent to a prominent instance of excluded middle known as Markov's principle.

We will make essential use of the extra-expressivity computational type theory has over set theory:

- Functions in CTT are computational. In contrast, set theory has no native notion of computability. Set-theoretic functions are merely notational sugar for functional relations.
- Computational availability (Σ) can be used besides propositional existence ($\exists$).
- Complex computational constructions can be carried out with certifying functions combining specification and verification using the Skolem translation.

28.1 Tests and Basic Predicates

Basic predicates are an abstract type-theoretic version of r.e. sets from set-theoretic computability theory. R.e. sets are sets of numbers that can be algorithmically enumerated. R.e. sets are also known recursively enumerable sets.

We define **complement** and **equivalence** of predicates $p^{X\to\mathbb{P}}$ and $q^{X\to\mathbb{P}}$:

$$\begin{array}{rcll} \overline{p} & := & \lambda x.\neg px & \qquad\qquad \textbf{complement} \\ p \equiv q & := & \forall x.\, px \longleftrightarrow qx & \qquad\qquad \textbf{equivalence} \end{array}$$

We have $p \equiv q \to \overline{p} \equiv \overline{q}$. We define basic predicates as follows:

$$\begin{array}{rcll} \mathsf{T} & := & \mathsf{N} \to \mathsf{B} & \qquad\qquad \textbf{unary tests} \\ \mathsf{T}_2 & := & \mathsf{N} \to \mathsf{N} \to \mathsf{B} & \qquad\qquad \textbf{binary tests} \\ \mathsf{K}\, f^{\mathsf{T}} & := & \exists k.\, fk = \mathsf{true} & \qquad\qquad \textbf{satisfiability} \\ \mathsf{D}\, f^{\mathsf{T}_2} & := & \lambda n.\, \mathsf{K}(fn) & \qquad\qquad \textbf{domain} \\ \mathsf{basic}\; p^{\mathsf{N}\to\mathbb{P}} & := & \exists f^{\mathsf{T}_2}.\, p \equiv \mathsf{D}\, f & \qquad\qquad \textbf{basic predicates} \end{array}$$

We say that basic predicates are the domains of binary tests, and that basic predicates are generated by binary tests.

Fact 28.1.1 Decidable predicates on numbers are basic: $\forall p^{\mathsf{N}\to\mathbb{P}}.\ \mathsf{dec}\, p \to \mathsf{basic}\, p$.

Proof p is equivalent to the domain of the test $\lambda nk.$ IF pn THEN true ELSE false. ∎

Computational intuition tells us that K and $\overline{\mathsf{K}}$ are computationally undecidable. For K the situation is somewhat better than for $\overline{\mathsf{K}}$ since every n such that $fn = \mathsf{true}$ is a certificate for $\mathsf{K}\, f$. For $\overline{\mathsf{K}}$ no such certificate system exists.

Given a test f, we can perform a linear search $f0, f1, f2, \ldots$ that halts with the first n such that $fn = \mathsf{true}$. We refer to this n is the least witness of f. Using an EWO, we can compute the least witness of a satisfiable test and use it as a certificate for $\mathsf{sat}\, f$.

Fact 28.1.2 (Witness)
For a satisfiable test one can compute a satisfying number: $\mathsf{sat}\, f \to \Sigma n.\, fn = \mathsf{true}$.

Proof Follows with an EWO for N (Fact 14.2.2) since boolean equality is decidable. ∎

Exercise 28.1.3 Prove the following:

a) $\neg\, \mathsf{K}\, f \longleftrightarrow \forall k.\, fk = \mathsf{false}$

b) $\mathsf{D}\,(\lambda n.f) \equiv \mathsf{D}\,(\lambda n.g) \longleftrightarrow (\mathsf{K}\, f \longleftrightarrow \mathsf{K}\, g)$

c) $\mathsf{D}\,(\lambda n.f) \equiv \mathsf{D}\,(\lambda nk.\mathsf{false}) \longleftrightarrow \neg\, \mathsf{K}\, f$

Exercise 28.1.4 Construct certifying functions as follows:

a) $\forall P^{\mathbb{P}}.\ \mathcal{D}P \to \Sigma f.\, P \longleftrightarrow \mathsf{K}\, f$

b) $\forall p^{\mathsf{N}\to\mathbb{P}}.\ \mathsf{dec}\, p \to \Sigma f.\, \mathsf{ex}\, p \longleftrightarrow \mathsf{K}\, f$

Exercise 28.1.5 (Conjunction and disjunction of tests)
Construct certifying functions for unary tests as follows:

a) $\forall f g.\, \Sigma h.\, \mathsf{K}\, h \longleftrightarrow \mathsf{K}\, f \wedge \mathsf{K}\, g$

b) $\forall f g.\, \Sigma h.\, \mathsf{K}\, h \longleftrightarrow \mathsf{K}\, f \vee \mathsf{K}\, g$

28.2 UT and Undecidability

Before we formulate Axiom CT, we look at a consequence of Axiom CT that suffices for almost all results in this chapter:

$$\mathsf{UT} \ := \ \Sigma U^{\mathsf{N}\to\mathsf{T}_2}\ \forall f^{\mathsf{T}_2}\ \exists c.\ \ \mathsf{D}\, f \equiv \mathsf{D}\,(Uc)$$

We refer to the function U provided by UT as universal test, and to the number c as the code for the test f. We see U as a function enumerating the domains of binary tests via binary tests. That computational objects are computationally enumerable is a prominent insight in computability theory.

With UT we will show a number of undecidability results, including the existence of an undecidable basic set and the undecidability of the satisfaction predicate K for unary tests.

We now come to our first undecidability result. The key idea of the proof is to show that the complement of the domain of the diagonal test $\lambda n.Unn$ of a universal test U is not basic.

Fact 28.2.1 (Undecidability)
$\mathsf{UT} \to \Sigma f^{\mathsf{T}_2}.\ \neg\, \mathsf{basic}\, \overline{\mathsf{D}\, f}$.

Proof We show that the complement of the domain of the diagonal binary test $\lambda n.Unn$ is not basic. Suppose $\mathsf{D}\, f \equiv (\lambda n. \neg \mathsf{K}(Unn))$ for some binary test f. By UT we have $\mathsf{D}\, f \equiv \mathsf{D}\,(Uc)$ for some c. Hence $\neg\mathsf{K}(Ucc) \longleftrightarrow \mathsf{D}\,(Uc)c = \mathsf{K}(Ucc)$, which is contradictory. ■

Fact 28.2.2 (Undecidability)

1. $\mathsf{UT} \to \Sigma p.\ \ \mathsf{basic}\, p \wedge \neg\, \mathsf{basic}\, \overline{p}$
2. $\mathsf{UT} \to \Sigma p.\ \ \mathsf{basic}\, p \wedge \neg\, \mathsf{dec}\, p$

Proof (1) follows from Fact 28.2.1 with $p := \mathsf{D}\, f$. (2) follows from (1) since complements of decidable predicates are decidable and hence basic. ■

Fact 28.2.3 (Undecidability)

1. $\mathsf{UT} \to \neg\, \mathsf{dec}\, \overline{\mathsf{K}}$
2. $\mathsf{UT} \to \neg\, \mathsf{dec}\, \mathsf{K}$

Proof (2) follows from (1) since complements of decidable predicates are decidable. For (1), we assume a decider $d : \forall f.\ \neg \mathsf{K}\, f$ and derive a contradiction. We consider the binary test $fn :=$ IF $d(Unn)$ THEN $\lambda k.\mathsf{true}$ ELSE $\lambda k.\mathsf{false}$ with $\mathsf{D}\, f \equiv \mathsf{D}\,(Uc)$. We have $\mathsf{K}\,(fc) \longleftrightarrow \mathsf{K}\,(Ucc)$. We do a case analysis on $d(Ucc)$. If $\neg\mathsf{K}\,(Ucc)$, $fc = \lambda k.\mathsf{true}$ and we have a contradiction. If $\neg\neg\mathsf{K}\,(fc)$, $fc = \lambda k.\mathsf{false}$ and we also have a contradiction. ■

Note that in the above proof the diagonal test plays a key role again. Later (Fact 28.8.5) we will see a different proof building on Fact 28.2.1 rather than UT.

Fact 28.2.4 (Undecidability)
Domain membership and domain satisfiability are undecidable for binary tests:

1. $\mathsf{UT} \to \forall n.\ \neg\mathsf{dec}\,(\lambda f^{\mathsf{T}_2}.\, \mathsf{D}\, fn)$
2. $\mathsf{UT} \to \neg\mathsf{dec}\,(\lambda f^{\mathsf{T}_2}.\, \mathsf{ex}\,(\mathsf{D}\, f))$

Proof Straightforward consequences of the undecidability of K (Fact 28.2.3 (2)). ■

Exercise 28.2.5 Assuming UT, prove that there is a basic predicate that cannot be expressed as a unary test: $\mathsf{UT} \to \Sigma p.\ \mathsf{basic}\, p \wedge (\neg\exists f^{\mathsf{T}}\, \forall n.\ npn \longleftrightarrow fn = \mathsf{true})$.

Exercise 28.2.6 Assuming UT, prove that there is a binary test f such that for every binary test g whose domain is disjoint from the domain of f there exists a number n such that neither fn nor gn is satisfiable.

28.3 Diophantine Expressions

Axiom CT will say that every basic predicate can be described with a diophantine expression. CT implies that all basic predicates are generated by countably many arithmetic expressions.

In what follows arithmetic pairing will be essential.

Fact 28.3.1 (Arithmetic pairing) There are functions

$$\pi : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \qquad \pi_1 : \mathsf{N} \to \mathsf{N} \qquad \pi_2 : \mathsf{N} \to \mathsf{N}$$

satisfying the equations

$$\pi_1(\pi n_1 n_2) = n_1 \qquad \pi_2(\pi n_1 n_2) = n_2 \qquad \pi(\pi_1 n)(\pi_2 n) = n$$

for all n_1, n_2, and n.

Proof Exercise 7.2.7. ∎

Diophantine expressions are arithmetic expressions obtained with variables, the constants 0 and 1, and addition, subtraction, and multiplication. We obtain **diophantine expressions** with an inductive type:

$$e : \mathsf{exp} ::= x \mid 0 \mid 1 \mid e + e \mid e - e \mid e \cdot e \qquad (x : \mathsf{N})$$

To evaluate an expression, we need values for the variables occurring in the expression. We assemble the values of variables into a single number using arithmetic pairing:

$$\begin{aligned} \mathsf{cons}\, n_1\, n_2 &:= \mathsf{S}(\pi\, n_1 n_2) \\ \mathsf{get}\ 0\ \_ &:= 0 \\ \mathsf{get}\ \mathsf{S}n\ 0 &:= \pi_1 n \\ \mathsf{get}\ \mathsf{S}n\ \mathsf{S}k &:= \mathsf{get}\,(\pi_2 n)\, k \end{aligned}$$

Note that $\mathsf{get}\, nk$ gets the value for variable k from n. If n doesn't provide an explicit value for a variable, get returns 0 as default value.

We evaluate expressions with a function $\mathsf{eva}^{\mathsf{N} \to \mathsf{exp} \to \mathsf{N}}$ defined follows:

$$\begin{aligned} \mathsf{eva}\ n\ x &:= \mathsf{get}\, n\, x \\ \mathsf{eva}\ n\ 0 &:= 0 \\ \mathsf{eva}\ n\ 1 &:= 1 \\ \mathsf{eva}\ n\ (e_1 + e_2) &:= \mathsf{eva}\, n\, e_1 + \mathsf{eva}\, n\, e_2 \\ \mathsf{eva}\ n\ (e_1 - e_2) &:= \mathsf{eva}\, n\, e_1 - \mathsf{eva}\, n\, e_2 \\ \mathsf{eva}\ n\ (e_1 \cdot e_2) &:= \mathsf{eva}\, n\, e_1 \cdot \mathsf{eva}\, n\, e_2 \end{aligned}$$

We now define functions obtaining **diophantine predicates** and **diophantine tests** from diophantine expressions:

$$\begin{aligned} \mathsf{P}e &: \mathsf{N} \to \mathbb{P} := \lambda n.\ \exists k.\ \mathsf{eva}\,(\mathsf{cons}\, n\, k)\, e) = 0 \\ \tau\, e &: \mathsf{T}_2 \qquad := \lambda nk.\ \text{IF}\ \mathsf{eva}\,(\mathsf{cons}\, n\, k)\, e\ \text{THEN}\ \mathsf{true}\ \text{ELSE}\ \mathsf{false} \end{aligned}$$

Fact 28.3.2 (Diophantine predicates)
Diophantine predicates are basic:

1. $\mathsf{pred}\, e \equiv \mathsf{D}\,(\tau e)$
2. $\mathsf{basic}\,(\mathsf{pred}\, e)$

Proof Straightforward. ∎

Diophantine predicates are a type-theoretic representation of recursively enumerable sets. Augmenting diophantine expressions with additional operations, for instance the arithmetic pairing functions π, π_1, and π_2, will not change the class of diophantine predicates. That addition, subtraction, and multiplication suffice to obtain all recursively enumerable sets is a famous result of Yuri Matiyasevich from 1970. The result implies that satisfiability of diophantine equations is undecidable, thus solving Hilbert's tenth problem from 1900. See Matiyasevich [24] for a fascinating first-hand discussion of the diophantine representation of recursively enumerable sets.

Fact 28.3.3 The type exp of expressions is in bijection with N.

Proof By Corollary 23.7.3 and Theorem 23.3.2, it suffices to construct an equality decider and a list enumeration for exp. We leave these routine constructions as exercises. It is also necessary to show that exp is an infinite type, which is straightforward. ∎

Exercise 28.3.4 (Binary universal test) UT models a universal test as a ternary test. With arithmetic pairing, a universal test can be modeled equivalently with a binary test. Prove $\mathsf{UT} \Leftrightarrow \Sigma g\, \forall f\, \exists c.\, \mathsf{D}\, f \equiv \mathsf{D}\,(\lambda n.g(\pi c n))$.

28.4 Axiom CT

We now formulate Axiom CT:

$$\mathsf{CT} \;:=\; \forall f^{\mathsf{T}_2}\, \exists e.\;\; \mathsf{D}\, f \equiv \mathsf{P}\, e$$

Axiom CT is consistent with excluded middle and extensionality. CT excludes degenerate interpretations of CTT where all predicates are interpreted as computationally decidable predicates. We may say that type theory becomes fully computational only once CT is assumed.

We mention that in informal constructive mathematics there is an assumption called Church's thesis saying that every total function is computable. CT as defined here may be seen as an adaption of Church's thesis to CTT and the diophantine representation of recursive enumerability.

Fact 28.4.1 $\mathsf{CT} \to \mathsf{UT}$.

Proof Let $\delta^{\mathsf{N}\to\mathsf{exp}}$ the surjective function provided by Fact 28.3.3. We verify that $U := \lambda c.\, \tau(\delta c)$ is a universal test. We assume f^{T_2} and prove $\exists c.\, \mathsf{D}\, f \equiv \mathsf{D}\,(Uc)$. CT gives us an expression e such that $\mathsf{D}\, f \equiv \mathsf{P}\, e$. Since δ is surjective, we have $\delta c = e$ for some c. It remains to show $\mathsf{D}\, f \equiv \mathsf{D}\,(Uc)$, which holds by the assumptions and $\mathsf{P}\, e \equiv \mathsf{D}\,(\tau e)$ (Fact 28.3.2 (1)). ■

Fact 28.4.2 (Undecidability)

1. $\mathsf{CT} \to \exists e.\, \neg \mathsf{basic}\,(\overline{\mathsf{P}e})$
2. $\mathsf{CT} \to \exists e.\, \neg \mathsf{dec}\,(\mathsf{P}e)$
3. $\mathsf{CT} \to \neg \forall en.\, \mathcal{D}(\mathsf{P}en)$

Proof (1) follows with Facts 28.2.1 and 28.4.1. (2) is a consequence of (1), and (3) is a consequence of (2). ■

Exercise 28.4.3
Prove $\mathsf{CT} \to (\mathsf{basic}\, p \longleftrightarrow \mathsf{diophantine}\, p)$ where $\mathsf{diophantine}\, p := \exists e.\, p \equiv \mathsf{P}e$.

Exercise 28.4.4 Prove the following:

a) $\forall e\, \exists e'\, \forall k.\, \mathsf{eva}\,(\mathsf{cons}\, 0\, k)\, e = \mathsf{eva}\, k\, e'$

b) $\mathsf{CT} \to \forall f^{\mathsf{T}}\, \exists e.\, \mathsf{K}\, f \longleftrightarrow \exists n.\, \mathsf{eva}\, ne = 0$

Exercise 28.4.5 (Undecidability challenges) We formulate proof challenges that will require serious work with diophantine arithmetic.

a) $CT \to \neg\, \mathsf{dec}(\lambda e.\, \exists n.\, \mathsf{P}\, en)$

b) $CT \to \neg\, \mathsf{dec}(\lambda e.\, \exists n.\, \mathsf{eva}\, ne = 0)$

Note that the second challenge is a variant of the undecidability of the satisfiability of diophantine equations (Hilbert's 10th problem). The best starting point we can offer is Fact 28.4.2.

Exercise 28.4.6 (Abstract CT)
We have chosen to obtain the r.e. predicates with diophantine expressions building on Matiyasevich's result. There is also the possibility to obtain the r.e. predicates with expressions describing partial recursive functions or Turing machines. In any case we have a countable type of expressions that yields the r.e. predicates through binary tests. We may capture the generating expressions and Axiom CT with an abstraction described by the predicate

$$\mathsf{ACT}\; A^{\mathbb{T}}\, \tau^{A\to\mathsf{T}_2}\, \delta^{\mathsf{N}\to A} \;:=\; (\forall f\, \exists a.\, \mathsf{D}\, f \equiv \mathsf{D}\,(\tau a)) \;\wedge\; (\forall a\, \exists n.\, \delta n = a)$$

where A is an abstract type of expressions. Prove the following:

a) $\mathsf{CT} \to \mathsf{sig}\,(\mathsf{ACT}\ \mathsf{exp}\ \tau)$

b) $\mathsf{ACT}\ A\tau\delta \ \to\ \forall p.\ \mathsf{basic}\,p \longleftrightarrow \exists a.\ p \equiv \mathsf{D}\,(\tau a)$

c) $\mathsf{ACT}\ A\tau\delta \to \mathsf{UT}$

28.5 Recusant Relations

We may ask whether there are functional and total relations that cannot be represented with functions. Assuming UT, we will show that such relations exist. This confirms our expectation that relations are more expressive than functions since there is no computability constraint for relations.

We define the **characteristic relation** ρp of predicates $p^{X\to\mathbb{P}}$:

$$\rho : \forall X.\ (X \to \mathbb{P}) \to (X \to \mathsf{B} \to \mathbb{P})$$
$$\rho p x b \ := \ \text{IF } b \text{ THEN } px \text{ ELSE } \neg px$$

Fact 28.5.1 Characteristic relations are functional. Moreover, assuming XM, characteristic relations are total.

Next we show that the characteristic predicate ρp can be represented as a function if and only if p is decidable.

Fact 28.5.2 $\mathsf{dec}\,p \Leftrightarrow \Sigma f\ \forall x.\ \rho p x(fx)$.

Proof Let d be a decider for p. Then $fx := \text{IF } dx \text{ THEN } \mathsf{true} \text{ ELSE } \mathsf{false}$ represents ρp. Conversely, let f represent ρp. Then f is a boolean decider for p. ∎

Assuming UT and XM, we can now construct a functional and total relation $R^{\mathsf{N}\to\mathsf{B}\to\mathbb{P}}$ that cannot be represented with a function. We speak of a *recusant* relation.

Fact 28.5.3 (Recusant relation)
$\mathsf{UT} \to \mathsf{XM} \to \Sigma R^{\mathsf{N}\to\mathsf{B}\to\mathbb{P}}.\ \mathsf{functional}\,R \wedge \mathsf{total}\,R \wedge \neg\exists f\ \forall x.\ Rx(fx)$.

Proof Fact 28.2.2 gives us an undecidable predicate $p^{\mathsf{N}\to\mathbb{P}}$. By Fact 28.5.2 we know that ρp cannot be represented with a function. Moreover, Fact 28.5.1 tells us that ρp is functional and total. ∎

28.6 Inconsistent Strengthening of UT

One must be extremely careful with axioms one adds to a foundational sytem. A good example is

$$\mathsf{UT}_\Sigma \ := \ \Sigma U^{\mathsf{N}\to\mathsf{T}_2}\ \forall f^{\mathsf{T}_2}\ \Sigma c.\ \mathsf{D}\,f \equiv \mathsf{D}\,(Uc)$$

which strengthens UT by replacing propositional existence of c with computational availability of c. We will show that UT_Σ is inconsistent with function extensionality.[1] In fact, extensionality for tests

$$\mathsf{TE} \;:=\; \forall f g^{\mathsf{T}}.\ (\forall n.\ fn = gn) \to f = g$$

suffices for the inconsistency.

Lemma 28.6.1 $\mathsf{UT}_\Sigma \to \mathsf{TE} \to \mathsf{dec}\ \overline{\mathsf{K}}$.

Proof UT_Σ gives us functions $U^{\mathsf{N}\to\mathsf{T}_2}$ and $\gamma^{\mathsf{T}\to\mathsf{N}}$ such that $\forall f.\ \mathsf{K}\,f \longleftrightarrow \mathsf{K}\,(U(\gamma f)\,0)$. We now assume f and construct a decision $\mathcal{D}(\neg\mathsf{K}\,f)$. We consider the codes γf and $\gamma(\lambda k.\mathsf{false})$.
If $\gamma f = \gamma(\lambda k.\mathsf{false})$, we assume $\mathsf{K}\,f$ and derive a contradiction. We have $\mathsf{K}\,f \longleftrightarrow \mathsf{K}\,(\lambda k.\mathsf{false})$, a contradiction.
If $\gamma f \neq \gamma(\lambda k.\mathsf{false})$, we assume $\neg\mathsf{K}\,f$ and prove $\gamma f = \gamma(\lambda k.\mathsf{false})$. By the extensionality assumption it suffice to show $\forall k.\ fk = \mathsf{false}$, which follows from $\neg\mathsf{K}\,f$. ■

Fact 28.6.2 (Inconsistency) $\mathsf{UT}_\Sigma \to \mathsf{TE} \to \bot$.

Proof Follows with Lemma 28.6.1 and Fact 28.2.3. ■

Exercise 28.6.3 Prove $\mathsf{UT} \to \mathsf{UT}_\Sigma$.

28.7 Post Hierarchy

The Post hierarchy is obtained with a preorder $p \preceq q$ on unary predicates such that p is decidable if q is decidable. We shall use the Post hierarchy to define the semidecidable predicates as the predicates $p \preceq \mathsf{K}$. Semidecidable predicates are a generalization of basic predicates to general argument types.

Given predicates $p^{X\to\mathbb{P}}$ and $q^{Y\to\mathbb{P}}$, a **reduction of p to q** is a function $f^{X\to Y}$ such that $\forall x.\ p(x) \longleftrightarrow q(fx)$. Given a reduction of p to q, we can obtain a decider for p from a decider of q. We define a predicate

$$\begin{aligned}
\mathsf{red} \;&:\; \forall XY.\ (X \to \mathbb{P}) \to (Y \to \mathbb{P}) \to (X \to Y) \to \mathbb{P} \\
\mathsf{red}_{XY}\,pqf \;&:=\; \forall x.\ p(x) \longleftrightarrow q(fx)
\end{aligned}$$

expressing that **p reduces to q with f**. We now define **reduction types**:

$$p \preceq q \;:=\; \mathsf{sig}\,(\mathsf{red}\,pq)$$

[1]The result appears in Troelstra and Van Dalen [29] and Forster [10].

Reduction types $p \preceq q$ describe a preorder on predicates called **reducibility** that is compatible with predicate complement, subsumes predicate equivalence, and transports predicate decidability from right to left. We read $p \preceq q$ as p is **below** q or as q is **above** p. We will use two standard notations for the reduction preorder:

$$\begin{aligned} p \npreceq q &:= \neg(p \preceq q) \\ p \prec q &:= (p \preceq q) \times (p \npreceq q) \end{aligned}$$

Informally, we may see a preorder as a hierarchy. With this metaphor in mind, we refer to reducibility as the Post hierarchy to remember Emil Post who came up with computational reductions (called many-one reductions in computability theory) and studied their hierarchy in the 1940's.

Fact 28.7.1 We have functions as follows:

1. $p \preceq p$
2. $p \preceq q \to q \preceq r \to q \preceq r$
3. $p \preceq q \to \overline{p} \preceq \overline{q}$
4. $p \equiv q \to p \preceq q$
5. $p \preceq q \to \mathsf{dec}\, q \to \mathsf{dec}\, p$

We also define **interreduction types**:

$$p \approx q := (p \preceq q) \times (q \preceq p)$$

Interreduction types $p \approx q$ describe the equivalence relation obtained as the symmetric closure of the reduction preorder $p \preceq q$. We speak of **reduction equivalence**.

Fact 28.7.2 We have functions as follows:

1. $p \approx p$
2. $p \approx q \to q \approx p$
3. $p \approx q \to q \approx r \to q \approx r$
4. $p \approx q \to \overline{p} \approx \overline{q}$
5. $p \equiv q \to p \approx q$
6. $p \approx q \to \ \mathsf{dec}\, p \to \mathsf{dec}\, q$

When we construct concrete reductions, it is often helpful to make use of the Skolem translation (Fact 10.3.2).

Fact 28.7.3 (Skolem for Reductions)
$(\forall x\, \Sigma y.\; px \longleftrightarrow qy) \to p \preceq q$.

Under XM we have $\overline{\overline{p}} \equiv p$ for all predicates p. We analyse the situation on a per-predicate basis using **stability of predicates**:

$$\mathsf{stable}\, p \;:=\; \forall x.\, \neg\neg px \to px$$

Fact 28.7.4 (Stable predicates)

1. $\mathsf{stable}\, p \;\longleftrightarrow\; \overline{\overline{p}} \equiv p$
2. $\mathsf{stable}\, \overline{p}$
3. $\mathsf{dec}\, p \;\to \mathsf{stable}\, p$
4. $p \preceq q \;\to\; \mathsf{stable}\, q \;\to\; \mathsf{stable}\, p$
5. $\mathsf{stable}\, p \to \overline{p} \preceq q \;\to\; p \preceq \overline{q}$
6. $\mathsf{stable}\, q \to p \preceq \overline{q} \;\to\; \overline{p} \preceq q$

The above fact is sharp in that in no statement an implication can be replaced with an equivalence.

Exercise 28.7.5 (Constancy of tests)
Prove that the following predicates on tests are reduction equivalent to $\overline{\mathsf{K}}$:

1. $\lambda g^{\mathsf{T}}.\forall n.\, gn = \mathsf{false}$
2. $\lambda g^{\mathsf{T}}.\forall n.\, gn = \mathsf{true}$
3. $\lambda g^{\mathsf{T}}.\exists b.\, \forall n.\, gn = b$
4. $\lambda f^{\mathsf{T}_2}.\, \exists b\, \forall nk.\, fnk = b$

Exercise 28.7.6 (Domain membership and domain satisfiability)
Prove the following:

a) $\forall n.\, (\lambda f.\, \mathsf{D}\, fn) \approx \mathsf{K}$

b) $(\lambda f.\, \mathsf{ex}\, (\mathsf{D}\, f)) \approx \mathsf{K}$

Hint for (b): Use arithmetic pairing to translate between unary and binary tests.

Exercise 28.7.7 (General Tests) Besides unary and binary tests we may also consider tests with three and more arguments. With arithmetic pairing we can show that satisfiability of tests with $n > 1$ arguments is interreducible with satisfiability of unary tests. What makes the problem interesting is its dependently typed formalization. We define a family $\mathsf{T}^{\mathsf{N}\to\mathbb{T}}$ of *test types*

$$\begin{aligned} \mathsf{T}_0 &:= \mathsf{B} \\ \mathsf{T}_{\mathsf{S}n} &:= \mathsf{N} \to \mathsf{T}_n \end{aligned}$$

and a family $\mathsf{sat}^{\forall n.\, \mathsf{T}_n \to \mathbb{P}}$ of *satisfaction predicates*:

$$\begin{aligned} \mathsf{sat}_0\, f &:= (f = \mathsf{true}) \\ \mathsf{sat}_{\mathsf{S}n}\, f &:= \exists k.\, \mathsf{sat}_n (fk) \end{aligned}$$

Prove the following:

a) $\forall n.\ \mathsf{sat}_{\mathsf{SS}n} \approx \mathsf{sat}_{\mathsf{S}n}$

b) $\forall n.\ \mathsf{sat}_{\mathsf{S}n} \approx \mathsf{K}$

Hint: The proof for (a) is basically the proof for Exercise 28.7.6.

28.8 Semidecidable Predicates

We will define semidecidable predicates as the predicates $p \preceq \mathsf{K}$. We also introduce a decidable predicate $\mathsf{D} \preceq \mathsf{K}$ such that a predicate p is decidable if and only if $p \preceq \mathsf{D}$. We define D as follows:

$$\mathsf{D} \ := \ \lambda b^{\mathsf{B}}.\, b = \mathsf{true}$$

Fact 28.8.1 (Decidable predicates)

1. $\mathsf{dec}\, D$
2. $\mathsf{stable}\, D$
3. $p \preceq \mathsf{D} \Leftrightarrow \mathsf{dec}\, p$
4. $\mathsf{dec}\, p \to \mathsf{dec}\, \overline{p}$
5. $p \preceq \mathsf{D} \to \overline{p} \preceq \mathsf{D}$
6. $\overline{\mathsf{D}} \approx \mathsf{D}$

We call a predicate **semidecidable** if it is below K and refer to reductions of p to K as **semideciders for p**. In other words, a predicate $p^{X\to\mathbb{P}}$ is semidecidable if and only if there is a function $f^{X\to\mathsf{T}}$ such that $\forall x.\ px \longleftrightarrow \mathsf{K}(fx)$. We refer to reductions of p to K as **semideciders for p** and define a notation for **semidecider types**:

$$\mathsf{sdec}_X\, p^{X\to\mathbb{P}} \ := \ (p \preceq \mathsf{K})$$

Finally, we call a predicate **cosemidecidable** if its complement is semidecidable.

Fact 28.8.2 Basic predicates are the semidecidable predicates on numbers: $\forall p^{\mathsf{N}\to\mathbb{P}}.\ \mathsf{basic}\, p \longleftrightarrow \Box\,(\mathsf{sdec}\, p)$.

Recall that $\Box$ is the truncation operator defined in § 10.5.

Fact 28.8.3

1. $\forall f^{\mathsf{T}_2}.\ \mathsf{D} f \preceq \mathsf{K}$
2. $\mathsf{D} \preceq \mathsf{K}$
3. $\mathsf{D} \preceq \overline{\mathsf{K}}$
4. $\mathsf{K} \preceq \mathsf{D} \to \overline{\mathsf{K}} \preceq \mathsf{K}$

Proof (1) is obvious from the definitions.

(2) $\lambda b.\lambda n.b$ reduces D to K.

3 We have $\mathsf{D} \preceq \overline{\mathsf{D}} \preceq \mathsf{D} \preceq \mathsf{K}$. Thus $\overline{\mathsf{D}} \preceq \overline{\mathsf{K}}$ and hence $\mathsf{D} \preceq \overline{\mathsf{K}}$.

(4) We have $\mathsf{K} \preceq \mathsf{D} \preceq \mathsf{K}$. Thus $\overline{\mathsf{K}} \preceq \mathsf{D}$ since D is closed under complements. Now $\overline{\mathsf{K}} \preceq \mathsf{K}$. ■

Fact 28.8.4

1. $\mathsf{sdec}\ \mathsf{K}$
2. $p \preceq q \to \mathsf{sdec}\, q \to \mathsf{sdec}\, p$
3. $\mathsf{dec}\, p \to \mathsf{sdec}\, p$
4. $\mathsf{dec}\, p \to \mathsf{sdec}\, \overline{p}$
5. $\forall f^{\mathsf{T}_2}.\ \mathsf{sdec}\,(\mathsf{D}\, f)$

We already know that K and $\overline{\mathsf{K}}$ are undecidable (Fact 28.2.3). We now show the stronger result that $\overline{\mathsf{K}}$ is not semidecidable building on Fact 28.2.1. The proof demonstrates the elegance of arguing within the Post hierarchy.

Fact 28.8.5 (Undecidabilty)

1. $\mathsf{UT} \to \neg\,\mathsf{sdec}\ \overline{\mathsf{K}}$
2. $\mathsf{UT} \to \neg\,\mathsf{dec}\ \mathsf{K}$

Proof (1) We have $\neg\,\mathsf{sdec}\ \overline{\mathsf{D}\, f}$ for some binary test f by Fact 28.2.1. By contraposition we assume $\mathsf{sdec}\ \overline{\mathsf{K}}$ and prove $\mathsf{sdec}\ \overline{\mathsf{D}\, f}$. It now suffices to prove $\overline{\mathsf{D}\, f} \preceq \overline{\mathsf{K}}$, which follows from $\mathsf{D}\, f \preceq \mathsf{K}$.
(2) We have $\neg\,\mathsf{sdec}\ \overline{\mathsf{K}}$ by (1). By contraposition we assume $\mathsf{dec}\ \mathsf{K}$ and prove $\mathsf{sdec}\ \overline{\mathsf{K}}$. Easy since the assumption yields $\mathsf{dec}\ \overline{\mathsf{K}}$ and thus $\overline{\mathsf{K}} \preceq \mathsf{D} \preceq \mathsf{K}$. ■

Exercise 28.8.6 Argue $\mathsf{D} \prec \mathsf{K}$ and $\overline{\mathsf{K}} \npreceq \mathsf{K}$.

Exercise 28.8.7 (Extensionality law via transport law)
Prove the extensionality law $p \equiv p' \to \mathsf{sdec}\, p \to \mathsf{sdec}\, p'$ using the transport law for sdec and the reduction law $p \equiv p' \to p \preceq p'$.

Exercise 28.8.8 (Constancy of tests)
Prove that constancy of tests $\lambda g^{\mathsf{T}}.\exists b.\ \forall n.\ gn = b$ is not semidecidable.
Hint: Exercise 28.7.5.

Exercise 28.8.9 Prove that domain emptiness of binary tests is not semidecidable: $\neg\,\mathsf{sdec}\,(\lambda f^{\mathsf{T}_2}.\ \neg\mathsf{ex}(\mathsf{D}\, f))$.

28.9 More Semidecidability

One may see semidecidable predicates as arithmetic projections $\lambda x.\exists k.\,p(x,k)$ of decidable predicates $p^{X\times\mathsf{N}\to\mathbb{P}}$. In fact, because of arithmetic pairing, the projection predicate $\lambda x.\exists k.\ p(x,k)$ is semidecidable already if the underlying predicate p is semidecidable. Moreover, the underlying predicate p may have more than one step index. For instance, $\lambda x.\exists k_1 k_2.\,p(x,k_1,k_2)$ is semidecidable if $p^{X\times\mathsf{N}\times\mathsf{N}\to\mathbb{P}}$ is semidecidable.

In the following we assume that π_1 and π_2 are the projections for an arithmetic pairing function π (Exercise 7.2.7).

Fact 28.9.1 (Arithmetic projection)
Given a semidecidable predicate $p^{X\times\mathsf{N}\to\mathbb{P}}$, $\lambda x.\ \exists n.\ p(x,n)$ is semidecidable:
$\forall p^{X\times\mathsf{N}\to\mathbb{P}}.\ \mathsf{sdec}\,p \to \mathsf{sdec}\,(\lambda x.\exists n.\ p(x,n))$.

Proof Let $f^{X\times\mathsf{N}\to\mathsf{T}}$ be a semidecider for $p^{X\times\mathsf{N}\to\mathbb{P}}$. We show that

$$gxn := \text{IF } f(x,\pi_1 n)(\pi_2 n) \text{ THEN } \mathsf{true} \text{ ELSE } \mathsf{false}$$

satisfies $(\exists n.\,p(x,n)) \longleftrightarrow \mathsf{K}\,(gx)$ for all x.
For $\to$, we assume $p(x,n)$ and show $\mathsf{K}(gx)$. From $p(x,n)$ we obtain $\mathsf{K}\,(f(x,n))$ and thus $f(x,n)k$ for some k. Now $gx(\pi nk) = \mathsf{true}$ follows.
For $\leftarrow$, we assume $gxk = \mathsf{true}$ and show $p(x,\pi_1 k)$. By the primary assumption it suffices to show $f(x,\pi_1 k)(\pi_2 k)$, which follows. ∎

Corollary 28.9.2 (Arithmetic double projection)
Given a semidecidable predicate $p^{X\times\mathsf{N}\times\mathsf{N}\to\mathbb{P}}$, $\lambda x.\ \exists nk.\ pxnk$ is semidecidable:
$\forall p^{X\times\mathsf{N}\times\mathsf{N}\to\mathbb{P}}.\ \mathsf{sdec}\,p \to \mathsf{sdec}\,(\lambda x.\ \exists nk.\ p(x,n,k))$.

Proof Apply Fact 28.9.1 twice. Note that $X\times\mathsf{N}\times\mathsf{N} = (X\times\mathsf{N})\times\mathsf{N}$, and that $(x,n,k) = ((x,n),k)$. ∎

It took several iterations until the author arrived at the formalization of arithmetic projection appearing above. There are two essential design decisions: Assume that the underlying predicate is semidecidable (rather than decidable) and use the cartesian representation for predicates with more than one argument.

Using arithmetic pairing, we can obtain an EWO for basic predicates, a fact noticed by Andrej Dudenhefner in 2020.

Fact 28.9.3 (EWO for basic predicates)
$\forall p^{\mathsf{N}\to\mathbb{P}}.\ \mathsf{sdec}\,p \to \mathsf{ex}\,p \to \mathsf{sig}\,p$.

Proof Suppose $\mathsf{sdec}\,pf$ and $\mathsf{ex}\,p$. Then $qn := (f(\pi_1 n)(\pi_2 n) = \mathsf{true})$ is decidable and satisfiable. Hence an EWO for decidable predicates gives us n such that qn. Thus $p(\pi_1 n)$. ∎

Andrej Dudenhefner also discovered in 2020 that semidecidable equality predicates are decidable.

Fact 28.9.4 (Semidecidable Equality)
Equality predicates are decidable if and only if they are semidecidable:
$\mathcal{E}X \;\Leftrightarrow\; \Sigma f^{X\to X\to \mathsf{T}}.\, \forall xy^X.\; x = y \longleftrightarrow \mathsf{K}(fxy)$.

Proof Direction $\rightarrow$ is obvious. For direction $\leftarrow$ suppose $\forall xy^X.\, x = y \longleftrightarrow \mathsf{sat}fxy$ and fix $x, y : X$. We need to construct a decision $\mathcal{D}(x = y)$. With an arithmetic EWO we compute k such that $fxxk = \mathsf{true}$. Now IF $fxyk$ THEN $x = y$ ELSE $x \neq y$ providing the decision. ■

Exercise 28.9.5 (Arithmetic projection) Show the following using Fact 28.9.1:

a) $\forall p^{X\to\mathsf{N}\to\mathbb{P}}.\; (\forall xn.\, \mathcal{D}(pxn)) \to \mathsf{sdec}\,(\lambda x.\, \mathsf{ex}(px))$

b) $\forall p^{X\times\mathsf{N}\times\mathsf{N}\times\mathsf{N}\to\mathbb{P}}.\; \mathsf{sdec}\,p \to \mathsf{sdec}\,(\lambda x.\, \exists k_1k_2k_3.\, p(x, k_1, k_2, k_3))$

Exercise 28.9.6 Construct a function $\mathsf{sdec}\,p \to \forall xy.\, px \to (py + (y \neq x))$.
Hint: The proof is similar to the proof of Fact 28.9.4. Using the witness operator one obtains n such that $fxn = \mathsf{true}$ and then discriminates on fyn. In fact, Fact 28.9.4 is a consequence of the above result. The exercise was contributed by Marc Hermes in March 2021.

28.10 Markov's Principle

Suppose we have a proposition P and two tests f and g characterizing P and $\neg P$: $P \longleftrightarrow \mathsf{K}\,f$ and $\neg P \longleftrightarrow \mathsf{K}\,g$. Then we can perform a linear search

$$f0 \mid g0,\;\; f1 \mid g1,\;\; f2 \mid g2,\;\; \ldots$$

on the disjunctive test $\lambda n.\, fn \mid gn$. If the proposition P is definite (i.e., $P \vee \neg P$), the disjunctive test is satisfiable and the linear search will find the first n such that $fn \mid gn = \mathsf{true}$. From $fn \mid gn = \mathsf{true}$ we can obtain a decision $\mathsf{K}\,f + \mathsf{K}\,g$, from which we can obtain a decision $\mathcal{D}P$. In short, if we have a definite proposition P and tests characterizing P and $\neg P$, we can construct a decision for P.

The definiteness of all propositions P where P and $\neg P$ can be characterized by tests can be obtained from an instance of excluded middle known as **Markov's principle**:

$$\mathsf{MP} \;:=\; \forall f^{\mathsf{N}\to\mathsf{B}}.\; \neg\neg\mathsf{K}\,f \to \mathsf{K}\,f$$

Markov's principle says that satisfiability of tests is stable. In fact, $\mathsf{MP} = \mathsf{stable}\,\mathsf{K}$.

Fact 28.10.1 (Markov equivalenc)
The following propositions are equivalent: $\mathsf{MP} \longleftrightarrow \mathsf{stable}\,\mathsf{K} \longleftrightarrow \overline{\overline{\mathsf{K}}} \equiv \mathsf{K}$.

Fact 28.10.2 (Markov complement law) $\mathsf{MP} \to p \preceq \overline{\mathsf{K}} \to \overline{p} \preceq \mathsf{K}$.

Proof Follows with Facts 28.7.1 and 28.10.1. ∎

We define the disjunction $f|g$ of two tests f and g as the test

$$f|g \;:=\; \lambda n.\,\text{IF } fn \text{ THEN } \mathsf{true} \text{ ELSE } gn$$

Fact 28.10.3 $\mathsf{K}f \vee \mathsf{K}g \;\longleftrightarrow\; \mathsf{K}(f|g) \;\Leftrightarrow\; \mathsf{K}f + \mathsf{K}g$.

Proof Follows with Fact 28.1.2. ∎

Fact 28.10.4 (Markov)
The following types are equivalent:

1. MP
2. $\forall fg^{\mathsf{N}\to\mathsf{B}}.\ (\neg\mathsf{K}f \longleftrightarrow \mathsf{K}g) \to (\mathsf{K}f + \mathsf{K}g)$
3. $\forall P^{\mathbb{P}}\forall fg^{\mathsf{N}\to\mathsf{B}}.\ (P \longleftrightarrow \mathsf{K}f) \to (\neg P \longleftrightarrow \mathsf{K}g) \to \mathcal{D}P$

Proof $1 \to 2$. Let $\neg\mathsf{K}f \longleftrightarrow \mathsf{K}g$. Using Fact 28.10.3 and (1) we assume $\neg\mathsf{sat}f|g$ and derive a contradiction. Using the assumptions, we have $\neg\mathsf{K}f$ and $\neg\neg\mathsf{K}f$.

$2 \to 1$. We assume $\neg\neg\mathsf{K}f$ and prove $\mathsf{K}f$. It suffices to prove $\mathsf{K}f + \mathsf{sat}\lambda n.\mathsf{false}$. By (2) it suffices to prove $\neg\mathsf{K}f \longleftrightarrow \bot$, which follows from the assumption $\neg\neg\mathsf{K}f$.

$2 \to 3$. We assume $P \longleftrightarrow \mathsf{K}f$ and $\neg P \longleftrightarrow \mathsf{K}g$ and prove $\mathcal{D}P$. It suffice to show $\mathsf{K}f + \mathsf{K}g$, which follows with (2).

$3 \to 2$. We assume $\neg\mathsf{K}f \longleftrightarrow \mathsf{K}g$ and prove $\mathsf{K}f + \mathsf{K}g$. We instantiate (3) with $P := \mathsf{K}f$ and obtain $\mathcal{D}\mathsf{K}f$, which yields the claim. ∎

Corollary 28.10.5 (Bi-Testability) $\mathsf{MP} \to (P \longleftrightarrow \mathsf{K}f) \to (\neg P \longleftrightarrow \mathsf{K}g) \to \mathcal{D}P$.

Under MP, a predicate is decidable if and only if both the predicate and its complement are semidecidable.

Fact 28.10.6 (Bisemidecidability) $\forall p^{X\to\mathbb{P}}.\ \mathsf{MP} \to \mathsf{sdec}\,p \to \mathsf{sdec}\,\overline{p} \to \mathsf{dec}\,p$.

Proof Suppose f and g are semideciders for p and $\overline{p}$. We fix x and construct a decision $\mathcal{D}(px)$. The assumptions give us tests fx and gx characterizing px and $\neg px$. Now Corollary 28.10.5 yields a decision $\mathcal{D}(px)$. ∎

The results on Markov's principle and semi-decidability appear in [12].

Exercise 28.10.7 Prove the following equivalences:

a) $\mathsf{MP} \to \mathsf{dec}\, p \Leftrightarrow \mathsf{sdec}\, p \times \mathsf{sdec}\, \overline{p}$

b) $\mathsf{MP} \to \mathsf{dec}\, \mathsf{K} \Leftrightarrow \mathsf{sdec}\, \overline{\mathsf{K}}$

Exercise 28.10.8 Prove $\forall X^{\mathbb{T}}.\ X \to (\forall p^{X \to \mathbb{P}}.\ \mathsf{sdec}\, p \to \mathsf{sdec}\, \overline{p} \to \mathsf{dec}\, p) \to \mathsf{MP}$

Exercise 28.10.9 Prove $\mathsf{MP} \to \neg(\forall n.\ fn = b) \longleftrightarrow (\exists n.\ fn = !b)$.

28.11 Promises

We call functions $f^{\mathsf{N} \to \mathcal{O}X}$ **promises**. We see promises as tests with output. Given a promise f, we can perform a linear search $f0, f1, f2, \ldots$ until we find the first n and x such that $fn = {}^\circ x$. If the search terminates, we refer to n as the **span** and to x as the **value** of f. Formally, we define the predicates

$$\begin{aligned}
\mathsf{del}\ f^{\mathsf{N} \to \mathcal{O}X}\, n &:= (fn \neq \emptyset) \\
\mathsf{span}\ f^{\mathsf{N} \to \mathcal{O}X}\, n &:= \mathsf{least}\,(\mathsf{del}\, f)\, n \\
\delta\ f^{\mathsf{N} \to \mathcal{O}X}\, x\, n &:= \mathsf{span}\, f\, n \wedge fn = {}^\circ x \\
f{\Downarrow} &:= \mathsf{ex}\,(\mathsf{del}\, f) \\
f{\downarrow}x &:= \mathsf{ex}\,(\delta f x)
\end{aligned}$$

The least witness predicate least is from § 13.1.

We assume the implicit typings $f^{\mathsf{N} \to \mathcal{O}X}$ and x^X for the rest of the section.

Fact 28.11.1 $f{\downarrow}x \to f{\Downarrow}$.

Fact 28.11.2 (Uniqueness)
Spans and values of promises are unique:

1. $\mathsf{span}\, fn \to \mathsf{span}\, fn' \to n = n'$
2. $f{\downarrow}x \to f{\downarrow}x' \to x = x'$

Proof Follows with the uniqueness of least (Fact 13.1.1). ∎

Fact 28.11.3 (Decidability)

1. $\mathcal{D}\,(\mathsf{del}\, fn)$
2. $\mathcal{D}\,(\mathsf{span}\, fn)$
3. $\mathcal{E}(X) \to \mathcal{D}\,(\delta f x n)$

Proof Follows with the decidability of least (Fact 13.3.1). ∎

Fact 28.11.4 (Computability)

1. $f\Downarrow \to \Sigma n.\ \mathsf{span}\, fn$
2. $f\Downarrow \to \Sigma x.\ f\downarrow x$

Proof (1) follows with an existential least witness operator (Fact 13.2.5) and the decidability of del (Fact 28.11.3). (2) is a straightforward consequence of (1). ∎

Fact 28.11.5 (Semidecidability)

1. $\mathsf{sdec}\,(\lambda f.f\Downarrow)$
2. $\mathcal{E}X \to \mathsf{sdec}\,(\lambda f.f\downarrow x)$

Proof (1) $\lambda fn.$ IF fn THEN true ELSE false is a reduction $(\lambda f.f\Downarrow) \preceq \mathsf{K}$.
(2) $\lambda fn.$ IF $dfxn$ THEN true ELSE false where d decides δ (Fact 28.11.3) is a reduction $(\lambda f.f\downarrow x) \preceq \mathsf{K}$. ∎

Lemma 28.11.6 (Reductions)

1. $X \to \mathsf{K} \preceq (\lambda f.f\Downarrow)$
2. $(\lambda f.f\Downarrow) \preceq (\lambda f.f\downarrow x)$

Proof (1) $\lambda gn.$ IF gn THEN $°x$ ELSE $\emptyset$ is a reduction $\mathsf{K} \preceq (\lambda f.f\Downarrow)$.
(2) $\lambda fn.$ IF fn THEN $°x$ ELSE $\emptyset$ is a reduction $(\lambda f.f\Downarrow) \preceq (\lambda f.f\downarrow x)$. Direction $f\Downarrow \to rf\downarrow x$ of the correctness proof uses Fact 28.11.4 (2). ∎

Fact 28.11.7 (Interreducibility)
Promise delivery and test satisfiability are interreducible:

1. $X \to (\lambda f.f\Downarrow) \approx \mathsf{K}$
2. $\mathcal{E}X \to (\lambda f.f\downarrow x) \approx \mathsf{K}$

Proof Follows with Facts 28.11.5 and 28.11.6. ∎

Fact 28.11.8 (Pruning) For every promise one can construct a value-equivalent promise that delivers at most once:
$\forall f\ \Sigma f'.\ (\forall x.\ f\downarrow x \longleftrightarrow f'\downarrow x) \wedge (\forall xn.\ f'n = °x \to \delta fxn)$.

Proof Function $f'n :=$ IF dfn THEN fn ELSE $\emptyset$ where d is a decider for $\mathsf{span}\, fn$ (Fact 28.11.3) does it. Correctness can be verified by a case analysis following the definition of f'. ∎

Exercise 28.11.9 Show that the promise delivery predicates $\lambda f.\ f\Downarrow$ and $\lambda f.\ f\downarrow x$ are not cosemidecidable, assuming UT and an inhabited and discrete output type.

28.12 Promising Functions

A **promising function** is a function $f^{X \to (\mathsf{N} \to \mathcal{O}Y)}$ mapping values to promises. We use the notation

$$\mathsf{PF}\,XY \;:=\; X \to \mathsf{N} \to \mathcal{O}Y$$

for the types of promising functions. Given a promising function $f^{\mathsf{PF}\,XY}$, we call the predicate $\lambda x.fx\Downarrow$ the **domain of f** and the predicate $\lambda xy.fx \downarrow y$ the **delivery relation** of f. We call a promising function is **total** if its delivery relation is total.

Fact 28.12.1 (Promising functions)

1. The delivery relation of a promising function is functional.
2. The values of promising functions are computable: $fx\Downarrow \to \Sigma y.\ fx \downarrow y$.

Proof Immediate with Facts 28.11.2 and 28.11.4. ■

Fact 28.12.2 (Total promising functions)
Functions $g^{X \to Y}$ and total promising functions $f^{\mathsf{PF}\,XY}$ are intertranslatable:

1. $\forall g^{X \to Y}\ \Sigma f^{\mathsf{PF}\,XY}.\ \forall x.\ fx \downarrow hx.$
2. $\forall f^{\mathsf{PF}\,XY}.\ (\forall x.\ fx\Downarrow) \to \Sigma h^{X \to Y}.\ \forall x.\ fx \downarrow hx.$

Proof (1) follows with constant promises. (2) follows with the Skolem translation and Fact 28.12.1 (2). ■

Fact 28.12.3 (Composition)
Promising functions are closed under composition:
$\forall f^{\mathsf{PF}\,XY}\ \forall g^{\mathsf{PF}\,YZ}\ \Sigma h^{\mathsf{PF}\,XZ}\ \forall xz.\ \ hx \downarrow z \longleftrightarrow (\exists y.\ fx \downarrow y \wedge gy \downarrow z).$

Proof We combine the spans for $fx \downarrow y$ and $gy \downarrow z$ with arithmetic pairing and make use of the decidability of span:

```
$hxn$ := MATCH $fx(\pi_1 n)$
         [ $\emptyset \Rightarrow \emptyset$
         | $^\circ y \Rightarrow$ IF span $(fx)(\pi_1 n)$
               THEN IF span $(gy)(\pi_2 n)$ THEN $gy(\pi_2 n)$ ELSE $\emptyset$
               ELSE $\emptyset$ ]
```

Correctness of h can be verified with a case analysis following the definition of h and using the uniqueness of span. ■

Promising functions are semideciders with output. They can be seen as type-theoretic representation of computable functions as they appear in set-theoretic computability theory. We can translate between semideciders and promising functions such that the domain is preserved.

Fact 28.12.4 (Translations)

1. $Y \to \forall g^{X \to \mathsf{N} \to \mathsf{B}}.\, \Sigma f^{\mathsf{PF}\,XY}.\; \forall x.\; \mathsf{K}(gx) \longleftrightarrow fx\Downarrow$
2. $\forall f^{\mathsf{PF}\,XY}.\, \Sigma g^{X \to \mathsf{N} \to \mathsf{B}}.\; \forall x.\; \mathsf{K}(gx) \longleftrightarrow fx\Downarrow$

Proof (1) follows with $fxn := \text{IF } gxn \text{ THEN } {}^\circ y \text{ ELSE } \emptyset$ assuming y^Y. (2) follows with $gxn := \text{IF } fxn \text{ THEN } \mathsf{true} \text{ ELSE } \mathsf{false}$. ∎

It follows that a predicate is semidecidable if and only if it agrees with the domain of a promising function.

Fact 28.12.5 (Semidecidable domains)

1. $\forall f^{\mathsf{PF}\,XY}.\, \mathsf{sdec}\,(\lambda x.fx\Downarrow)$
2. $Y \to \forall p^{X \to \mathbb{P}}.\; \mathsf{sdec}\, p \Leftrightarrow \Sigma f^{\mathsf{PF}\,XY}.\; p \equiv \lambda x.fx\Downarrow$

Fact 28.12.6 (Undecidable domain)

$\mathsf{UT} \to Y \to \Sigma f^{\mathsf{PF}\,\mathsf{N}\,Y}.\; \neg\,\mathsf{sdec}\; \overline{\lambda n.\, fn\Downarrow}$.

Proof Follows with Facts 28.2.1 and 28.12.5 (2). ∎

Fact 28.12.7 (Semidecidability of delivery)

$\mathcal{E}Y \to \forall f^{\mathsf{PF}\,XY}.\, \mathsf{sdec}\,(\lambda x.\, fx \downarrow y)$.

Proof Let $f^{\mathsf{PF}\,XY}$. We use the Skolem translation and show

$$\forall x.\, \Sigma g.\; fx \downarrow y \longleftrightarrow \mathsf{K}g$$

The test $gn := \text{IF } d(fx)yn \text{ THEN } \mathsf{true} \text{ ELSE } \mathsf{false}$ where d decides δ (Fact 28.11.3) does it. ∎

28.13 Recusant Partial Deciders

We call a promising function $f^{\mathsf{PF}\,XY}$ **recusant** if it cannot be extended to a total promising function:

$$\mathsf{recusant}\, f^{\mathsf{PF}\,XY} \;:=\; \forall g.\, (\forall xy.\; fx \downarrow y \to gx \downarrow y) \to \exists x.\, \neg gx\Downarrow$$

Note that the definition of recusant states the nontotality of the extending function g existentially as $\exists x.\neg gx\Downarrow$ rather than negatively as $\neg(\forall x. gx\Downarrow)$. While the existential version implies the negative version, it takes excluded middle for the negative version to imply the existential version.

We call promising functions $f^{\mathsf{PF}\,X\,\mathsf{B}}$ **partial deciders**. Assuming UT, we will construct a recusant partial decider. Recusant partial deciders capture the notion of recursively inseparable sets from set-theoretic computability theory. If $f^{\mathsf{PF}\,\mathsf{N}\,\mathsf{B}}$ is a recusant partial decider, the predicates $\lambda x.\, fx \downarrow \mathsf{true}$ and $\lambda x.\, fx \downarrow \mathsf{false}$ represent recursively inseparable sets of numbers. The idea to capture recursive inseparability with partial deciders appears in Kirst and Peters [20].

Fact 28.13.1 (Decidable domain)
Promising functions with decidable domains have total extensions:
$Y \to \forall f^{\mathsf{PF}\,XY}.\ \mathsf{dec}\,(\lambda x.\, fx\Downarrow) \to \Sigma g.\ (\forall x. gx\Downarrow) \wedge (\forall xy.\ fx \downarrow y \to g \downarrow y)$.

Proof Suppose y^Y and $d^{\mathsf{dec}\,(\lambda x.\, fx\Downarrow)}$. Then $gx := \text{IF } dx \text{ THEN } fx \text{ ELSE } \lambda k.y$ is a total extension of f. ■

Corollary 28.13.2 (Undecidable domain)
Recusant functions have undecidable domains: $\mathsf{recusant}\, f \to \neg\, \mathsf{dec}\,(\lambda x. fx\Downarrow)$.

Fact 28.13.3 (Right composition)
$\forall f^{\mathsf{PF}\,XY}\ \forall h^{Y\to Z}\ \Sigma g^{\mathsf{PF}\,XZ}\ \forall xy.\ fx \downarrow y \to gx \downarrow hy$.

Proof $gxn := \text{MATCH } fxk\ [\,{}^\circ y \Rightarrow {}^\circ hy \mid \emptyset \Rightarrow \emptyset\,]$ does it. ■

We define **universal partial deciders** as follows:

$$\mathsf{UPD} \ := \ \Sigma U^{\mathsf{N}\to\mathsf{PF}\,\mathsf{N}\,\mathsf{B}}.\ \forall f^{\mathsf{PF}\,\mathsf{N}\,\mathsf{B}}\ \exists c\ \forall nb.\ fn \downarrow b \longleftrightarrow Ucn \downarrow b$$

Given a universal partial decider U, we show that the diagonal partial decider $\lambda n.Unn$ is recusant.

Fact 28.13.4 (Recusant partial decider)
$\mathsf{UPD} \to \Sigma\, f^{\mathsf{PF}\,\mathsf{N}\,\mathsf{B}}.\ \mathsf{recusant}\, f$.

Proof Let U be a universal partial decider. We show that the diagonal partial decider $\lambda n.Unn$ is recusant. Suppose $f^{\mathsf{PF}\,\mathsf{N}\,\mathsf{B}}$ is an extension of $\lambda n.Unn$. Fact 28.13.3 gives us $g^{\mathsf{PF}\,\mathsf{N}\,\mathsf{B}}$ such that $\forall nb.\ fn \downarrow b \to gn \downarrow !b$. Since U is universal, we have $\forall nb.\ gn \downarrow b \longleftrightarrow Ucn \downarrow b$ for some c. We now assume $fc\Downarrow$ and derive a contradiction. By Fact 28.11.4 (2) we have $fc \downarrow b$. We instantiate the assumptions as follows:

$$\begin{aligned} &Ucc \downarrow !b \to fc \downarrow !b \\ &gc \downarrow !b \longleftrightarrow Ucc \downarrow !b \\ &fc \downarrow b \to gc \downarrow !b \end{aligned}$$

Hence we have both $fc \downarrow b$ and $fc \downarrow !b$. Contradiction with functionality of delivery (Fact 28.11.2). ■

Exercise 28.13.5 We say that a partial decider $f^{\mathsf{PF}\,X\,\mathsf{B}}$ is **sound** for a predicate $p^{X \to \mathbb{P}}$ if $\forall x.\ \text{IF } fx \text{ THEN } px \text{ ELSE } \neg px$. Show that a predicate is decidable if and only if it has a sound and total partial decider.

28.14 Universal Partial Deciders

We now construct a universal partial decider from a universal test. The idea is as follows: Given a partial decider $f^{\mathsf{PF}\,\mathsf{N}\,\mathsf{B}}$, we slice it into two binary tests g_1 and g_2 such that the domains of g_1 and g_2 agree with $\lambda n.fn \downarrow \mathsf{true}$ and $\lambda n.fn \downarrow \mathsf{false}$. From the slices g_1 and g_2 we can rebuild the partial decider f. The universal partial decider will now use the code $\pi c_1 c_2$ for a partial decider whose slices have the codes c_1 and c_2 for the given universal test.

We define a function $\gamma f g$ combining two unary tests into a boolean promise:

$$\begin{aligned}\gamma &: (\mathsf{N} \to \mathsf{B}) \to (\mathsf{N} \to \mathsf{B}) \to (\mathsf{N} \to \mathcal{O}\mathsf{B})\\ \gamma f g &:= \lambda n.\ \text{IF } fn \text{ THEN } {}^\circ\mathsf{true} \text{ ELSE IF } gn \text{ THEN } {}^\circ\mathsf{false} \text{ ELSE } \emptyset\end{aligned}$$

Fact 28.14.1 $\gamma f g \downarrow b \to \text{IF } b \text{ THEN } \mathsf{K} f \text{ ELSE } \mathsf{K} g$.

Proof We assume $\gamma f g n = {}^\circ b$ and prove $\text{IF } b \text{ THEN } \mathsf{K} f \text{ ELSE } \mathsf{K} g$ by case analysis on fn and gn. Straightforward. ■

We define **disjointness** of unary tests as follows: $f \parallel g \ := \ \mathsf{K} f \to \mathsf{K} g \to \bot$.

Fact 28.14.2 (Disjoint test combination)
$f \parallel g \to \gamma f g \downarrow b \ \longleftrightarrow \ (\text{IF } b \text{ THEN } \mathsf{K} f \text{ ELSE } \mathsf{K} g)$.

Proof Direction $\to$ follows with Fact 28.14.1. We prove direction $\leftarrow$ for $b = \mathsf{true}$, the other case is analogous.
We assume $f \parallel g$ and $\mathsf{K} f$ and prove $\gamma f g \downarrow \mathsf{true}$. The assumption gives us $fn = \mathsf{true}$ for some n and thus $\gamma f g \Downarrow$. Now $\gamma f g \downarrow b$ for some b with Fact 28.11.4 (2). This closes the proof since $\gamma f g \downarrow \mathsf{false}$ is contradictory by Fact 28.14.1 and $f \parallel g$. ■

Fact 28.14.3 $\mathsf{UT} \to \mathsf{UPD}$.

Proof Let U be a universal test. We define

$$Vcn \ := \ \gamma\,(U(\pi_1 c)n)\,(U(\pi_2 c)n)$$

and show

$$\forall f^{\mathsf{PF}\,\mathsf{N}\,\mathsf{B}}\ \exists c\ \forall nb.\ fn \downarrow b \longleftrightarrow Vcn \downarrow b$$

We assume f^{PFNB} and slice it into binary tests g_1 and g_2 using Fact 28.12.7:

$$\forall n.\ fn\downarrow \mathsf{true} \longleftrightarrow \mathsf{K}\,(g_1 n)$$
$$\forall n.\ fn\downarrow \mathsf{false} \longleftrightarrow \mathsf{K}\,(g_2 n)$$

We now exploit the universality of U and obtain the codes for g_1 and g_2:

$$\forall n.\ \mathsf{K}\,(g_1 n) \longleftrightarrow \mathsf{K}\,(Uc_1 n)$$
$$\forall n.\ \mathsf{K}\,(g_2 n) \longleftrightarrow \mathsf{K}\,(Uc_2 n)$$

We now show

$$fn\downarrow b \longleftrightarrow V(\pi c_1 c_2)n\downarrow b$$

for given n and B. Using the definition of V, it remains to show

$$fn\downarrow b \longleftrightarrow \gamma(Uc_1 n)(Uc_2 n)\downarrow b$$

We now verify $Uc_1 n \parallel Uc_2 n$ using uniqueness of delivery for fn (Fact 28.11.2) and reduce the claim to

$$fn\downarrow b \longleftrightarrow \textsc{if}\ b\ \textsc{then}\ \mathsf{K}(Uc_1 n)\ \textsc{else}\ \mathsf{K}(Uc_2 n)$$

using Fact 28.14.2. This closes the proof since the equivalence is a straightforward consequence of the assumption for c_1 and c_2. ■

Theorem 28.14.4 (Recusant partial decider) $\mathsf{UT} \to \Sigma f^{\mathsf{PFNB}}.\ \mathsf{recusant}\,f$.

Proof Follows with Facts 28.13.4 and 28.14.3. ■

28.15 Notes

This chapter was written January to April 2024. I'm thankful to Dominik Kirst, whose slides for an Australian summer school got me started, and with whom I had weekly lunches and discussions during the writing. Dominik supplied me with results from the literature that evolved into Facts 28.5.3, 28.6.2, and 28.13.4.

We may say that this chapter investigates results from set-theoretic computability in computational type theory. The type-theoretic development profits from the fact that an explicit model of computation (e.g. Turing machines) can be replaced with the synthetic notion of computability that comes with CTT.

Different variants of Axiom CT (Church's thesis) appear in the literature. Troelstra and Van Dalen [29] discuss CT informally in the context of constructive Mathematics. Forster [10, 11] is the first to study CT in computational type theory. Forster [11] shows for a call-by-value lambda calculus L [13] formalized in CTT the following formulations of CT are equivalent:

1. Every function $\mathsf{N} \to \mathsf{N}$ in CTT is computable in L.
2. Every function $\mathsf{N} \to \mathsf{B}$ in CTT is computable in L.
3. The domain of every binary test in CTT can be obtained as the domain of a function in L.

Forster [11] also covers the diophantine representation of recursively enumerable sets.

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 $C_p e a$ with $e^{x=y}$ changes the type of a from px to py for every admissible type function p. We have

$$C(\mathsf{Q}x)a \;\approx\; a$$

and say that trivial casts $C(\mathsf{Q}x)a$ can be **discharged**. We also have

$$\forall p^{X\to\mathbb{T}}\,\forall e^{x=y}\forall a^{px}.\;C_p e a \;\approx\; \mathcal{J}(\lambda y\_.py)aye$$

which says that the cast eliminator can be expressed with the full eliminator.

Inductive quality as defined here is stronger than the Leibniz equality considered in Chapter 4. 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 $(\mathsf{true} = \mathsf{false}) \to \forall X^{\mathbb{T}}.\ X$ 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 (§4.5).

29.2 Uniqueness of Identity Proofs

We will now show that the following properties of types are equivalent:

$$
\begin{array}{rll}
\mathsf{UIP}(X) := & \forall xy^X\ \forall ee'^{x=y}.\ e = e' & \textit{uniqueness of identity proofs} \\
\mathsf{UIP}'(X) := & \forall x^X\ \forall e^{x=x}.\ e = \mathsf{Q}x & \textit{u. of trivial identiy proofs} \\
\mathsf{K}(X) := & \forall x\ \forall p^{x=x\to\mathbb{P}}.\ p(\mathsf{Q}x) \to \forall e.pe & \textit{Streicher's K} \\
\mathsf{CD}(X) := & \forall p^{X\to\mathbb{T}}\ \forall x\ \forall a^{px}\ \forall e^{x=x}.\ Cea = a & \textit{cast discharge} \\
\mathsf{DPI}(X) := & \forall p^{X\to\mathbb{T}}\ \forall xuv.\ (x,u)_p = (x,v)_p \to u = v & \textit{dependent pair injectivity}
\end{array}
$$

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 [18] 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 [16] (§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 [16]. 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 17.4.12.

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

$$\begin{aligned}
\forall xy\ \forall a^{\mathsf{L}xy}.\ &\textsc{match}\ y\ \textsc{return}\ \mathsf{L}xy \to \mathbb{T} \\
&[\,0 \Rightarrow \lambda a.\ \bot \\
&\mid \mathsf{S}y' \Rightarrow \lambda a^{\mathsf{L}x(\mathsf{S}y')}.\ (\Sigma e^{y'=x}.\ Cea = \mathsf{L}_1x) + (\Sigma a'.\ a = \mathsf{L}_2xy'a') \\
&]\,a
\end{aligned}$$

The formulation of the type follows the pattern we have seen before, except that there is a cast in the branch for L_1:

$$\Sigma e^{y'=x}.\ Cea = \mathsf{L}_1x$$

The cast is necessary since a has the type $\mathsf{L}x(\mathsf{S}y')$ while L_1x 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

$$\begin{aligned}
&\Sigma e^{x=x}.\ Ce(\mathsf{L}_1x) = \mathsf{L}_1x \\
&\Sigma a'.\ \mathsf{L}_2xy'a = \mathsf{L}_2xy'a'
\end{aligned}$$

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:

$$\forall a^{\mathsf{K}xz}\,\forall b^{\mathsf{K}zy}.\;\; \mathsf{K}_2xzyab = \mathsf{K}_2xzya'b' \to (a,b) = (a',b') \tag{29.3}$$

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 Fact 4.6.1 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 the applicative closure law (Fact 4.6.1).

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 [18] plays an important role in homotopy type theory [30], 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. [21]. That basic type theory cannot prove UIP was discovered by Hofmann and Streicher [18] in 1994 based on a so-called groupoid interpretation.

30 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 induction 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.

30.1 Recursion Types

We assume a binary relation $R^{X\to X\to\mathbb{P}}$ and pronounce the Ryx as **y below x**. We define the **recursion types** for R as follows:

$$\mathcal{A}_R(x : X) : \mathbb{P} \ ::= \ \mathsf{C}\,(\forall y.\ Ryx \to \mathcal{A}_R y)$$

and call the elements of recursion types **recursion certificates**. Note that recursion types are computational propositions. A recursion certificate of type $\mathcal{A}_R(x)$ justifies all recursions starting from x and descending on the relation R. That a recursion on a certificate of type $\mathcal{A}_R(x)$ terminates is ensured by the built-in termination property of computational type theory. Note that recursion types realize higher-order recursion.

We will harvest the recursion provided by recursion certificates with a **recursion operator**

$$\begin{aligned} W' &:\ \forall p^{X\to\mathbb{T}}.\ (\forall x.\ (\forall y.\ Ryx \to py) \to px) \to \forall x.\ \mathcal{A}_R x \to px \\ W'pFx(\mathsf{C}\varphi) &:= \ Fx(\lambda yr.\, W'pFy(\varphi yr)) \end{aligned}$$

Computationally, W' may be seen as an operator that obtains a function

$$\forall x.\ \mathcal{A}_R x \to px$$

from a **step function**

$$\forall x.\ (\forall y.\ Ryx \to py) \to px$$

The step function describes a function $\forall x.px$ obtained with a **continuation function**

$$\forall y.\ Ryx \to py$$

providing recursion for all y below x. We also speak of **recursion guarded by R**.

We define **well-founded relations** as follows:

$$\mathsf{wf}(R^{X \to X \to \mathbb{P}}) \ := \ \forall x.\ \mathcal{A}_R(x)$$

Note that a proof of a proposition $\mathsf{wf}(R)$ is a function that yields a recursion certificate $\mathcal{A}_R(x)$ for every x of the base type of R. For well-founded relations, we can specialize the recursion operator W' as follows:

$$\begin{aligned} &W : \mathsf{wf}(R) \to \forall p^{X \to \mathbb{T}}.\ (\forall x.\ (\forall y.\ Ryx \to py) \to px) \to \forall x.px \\ &WhpFx \ := \ W'pFx(hx) \end{aligned}$$

We will refer to W' and W as **well-founded recursion operators**. Moreover, we will speak of **well-founded induction** if a proof is obtained with an application of W' or W.

It will become clear that W generalizes the size induction 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 30.1.1 (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 30.1.2 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 30.1.3 Prove $\mathcal{A}_R(x) \to \neg Rxx$.
Hint: Use well-founded induction with W'.

Exercise 30.1.4 Prove $Rxy \to Ryx \to \neg\mathcal{A}_R(x)$.

Exercise 30.1.5 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 30.1.6 Suppose we narrow the propositional discrimination restriction of the underlying type theory such that recursion types are the only propositional types allowing for computational discrimination. We can still express an empty propositional type with computational falsity elimination:

$$V : \mathbb{P} \;::=\; \mathcal{A}_{(\lambda ab^{\top}.\top)}(\mathsf{I})$$

Define a function $V \to \forall X^{\mathbb{T}}.\, X$.

30.2 Well-founded Relations

Fact 30.2.1 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 30.2.2 (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 induction on R with the size function σ.

Fact 30.2.3 (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 30.2.4 (Well-founded size induction)
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 induction operator from §19.2 as a special case of the well-founded size induction operator.

Corollary 30.2.5 (Arithmetic size induction)
$\forall \sigma^{X\to \mathsf{N}}\, \forall p^{X\to \mathbb{T}}.\, (\forall x.\, (\forall x'.\sigma x' < \sigma x \to px') \to px) \to \forall x.\, px.$

Proof Follows with Corollary 30.2.4 and Fact 30.2.1. ∎

There is a story here. We came up with retracts to have an elegant construction of the wellfounded size induction operator appearing in Corollary 30.2.4. Note that conversion plays an important role in type checking the construction. The proof that retracts of well-founded relations are well-founded (Fact 30.2.3) 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 30.2.6 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 30.1.1).

Exercise 30.2.7 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.

30.3 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 induction 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 30.3.1 (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 30.3.2 (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 30.3.1. ■

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 30.3.2. We now prove that all functions satisfying a step function agree if FE is assumed and R is well-founded.

Fact 30.3.3 (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 30.3.4 Note that the proof of Lemma 30.3.1 doubles the quantification of a. Verify that this is justified by the general law $(\forall a.\forall a.pa) \to \forall a.pa$.

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

$$\begin{aligned}
&\text{guard conditions} \\
x \le y \;&\to\; \mathsf{S}x + (y-x) < \mathsf{S}x + \mathsf{S}y \\
x > y \;&\to\; (x-y) + \mathsf{S}y < \mathsf{S}x + \mathsf{S}y
\end{aligned}$$

Figure 30.1: Recursive specification of a gcd function

30.4 Example: GCDs

Our second example for the use of well-founded recursion and the unfolding equation is the construction of a function computing GCDs (§ 20.5). We start with the procedural specification in Figure 30.1. 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 30.1, 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 30.2.1) and retracts (Fact 30.2.3). Each of the three specifying equations in Figure 30.1 can now be obtained as an instance of the unfolding equation (Fact 30.3.2).

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 30.4.1 Construct a function $f^{\mathsf{N}\to\mathsf{N}\to\mathsf{N}}$ satisfying the Ackermann equations (§1.11) using well-founded recursion for the lexical product $<_\mathsf{N} \times <_\mathsf{N}$.

30.5 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 30.5.1 $\mathsf{ext}(F) \to \forall xaa'.\ W'Fxa = W'Fxa'$.

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 30.5.2 (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 30.5.1. ■

Exercise 30.5.3 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 (§30.3).

b) Prove that the step function for GCDs is extensional (§30.4).

c) Prove that the step function for the Ackermann equations is extensional (Exercise 30.4.1).

Exercise 30.5.4 Show that all functions satisfying an extensional step function for a well-founded relation agree.

30.6 Witness Operator

There is an elegant and instructive construction of an existential witness operator for numbers (Fact 14.2.2) using recursion types. We assume a decidable predicate $p^{\mathsf{N}\to\mathbb{P}}$ and define a relation

$$Rxy \ :=\ x = \mathsf{S}y \land \neg py$$

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 30.6.1 $p(x+y) \to \mathcal{A}_R(y)$.

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 30.6.2 $\mathcal{A}_R(x) \to \mathsf{sig}(p)$.

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 30.6.3 (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 30.6.2 it suffices to show $\mathcal{A}_R(0)$. We can now obtain a witness x for p. The claim follows with Lemma 30.6.2. ∎

We may see the construction of an existential witness operator for numbers with linear search types (Fact 14.2.2) as a specialization of the construction shown here where the general recursion types used here are replaced with special purpose linear search types.

Exercise 30.6.4 Prove $\mathcal{A}_R(n) \longleftrightarrow T(n)$.

Exercise 30.6.5 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.

30.7 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 30.1 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 §30.5.

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 induction. 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.

30.8 Padding and Simplification

Given a certificate $a : \mathcal{A}_R(x)$, we can obtain a computationally equal certificate $b : \mathcal{A}_R(x)$ that exhibits any number of applications of the constructor for certificates:

$$\begin{aligned} a &\approx Cx(\lambda yr.\,a') \\ a &\approx Cx(\lambda yr.Cy(\lambda y'r'.\,a'')) \end{aligned}$$

We formulate the idea with two functions

$$\begin{aligned} &D:\ \forall x.\,\mathcal{A}_R(x) \to \forall y.\,Ryx \to \mathcal{A}_R(y) \\ &Dxa \ := \ \text{MATCH}\ a\ [\,C\varphi \Rightarrow \varphi\,] \end{aligned}$$

$$\begin{aligned} &P:\ \mathsf{N} \to \forall x.\,\mathcal{A}_R(x) \to \mathcal{A}_R(x) \\ &\quad P\,0xa \ := \ a \\ &P\,(\mathsf{S}n)a \ := \ Cx(\lambda yr.\,Pny(Dxayr)) \end{aligned}$$

and refer to P as **padding function**. We have, for instance,

$$\begin{aligned} P(1+n)xa &\approx Cx(\lambda y_1 r_1.\, Pny_1(Dxay_1r_1)) \\ P(2+n)xa &\approx Cx(\lambda y_1 r_1.\, Cy_1(\lambda y_2 r_2.\, Pny_2(Dy_1(Dxay_1r_1)y_2r_2))) \end{aligned}$$

The construction appears tricky and fragile on paper. When carried out with a proof assistant, the construction is fairly straightforward: Type checking helps with the definitions of D and P, and simplification automatically obtains the right hand sides of the two examples from the left hand sides.

When we simplify a term $P(k+n)xa$ where k is a concrete number and n, x, and a are variables, we obtain a term that needs at least $2k$ additional variables to be written down. Thus the example tells us that simplification may have to introduce an unbounded number of fresh variables.

The possibility for padding functions seems to be a unique feature of higher-order recursion.

Exercise 30.8.1 Write a padding function for linear search types (§ 14.1).

30.9 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 30.9.1 (Characterization by well-founded induction)
$\forall R^{X\to X\to\mathbb{P}}.\ \ \mathsf{wf}(R) \longleftrightarrow \forall p^{X\to\mathbb{P}}.\, (\forall x.\, (\forall y.\, Ryx \to py) \to px) \to \forall x.\, px.$

Proof Direction $\to$ follows with W. For the other direction, we instantiate p with $\mathcal{A}_R$. It remains to show $\forall x.\, (\forall y.\, Ryx \to \mathcal{A}_R y) \to \mathcal{A}_R x$, which is an instance of the type of the constructor for $\mathcal{A}_R$. ■

The characterization of well-foundedness with the principle of well-founded induction is very interesting since no inductive types and only a predicate $p^{X\to\mathbb{P}}$ is used. Thus the computational aspects of well-founded recursion are invisible. They are added by the presence of the inductive predicate $\mathcal{A}_R$ admitting computational elimination.

Next we establish a positive characterization of the non-well-founded elements of a relation. We define **progressive predicates** and **progressive elements** for a relation $R^{X \to X \to \mathbb{P}}$ as follows:

$$\begin{aligned} \mathsf{pro}_R(p^{X \to \mathbb{P}}) &:= \forall x.\ px \to \exists y.\ py \land Ryx \\ \mathsf{pro}_R(x^X) &:= \exists p.\ px \land \mathsf{pro}_R(p) \end{aligned}$$

Intuitively, progressive elements for a relation R are elements that have an infinite descent in R. Progressive predicates are defined such that every witness has an infinite descent in R. Progressive predicates generalize the frequently used notion of infinite descending chains.

Fact 30.9.2 (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 30.9.3 (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 30.9.4 (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 30.9.2.

For direction $\leftarrow$ we assume $\neg\exists x.\ \mathsf{pro}_R(x)$ and prove $\mathcal{A}_R(x)$. By Fact 30.9.3 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 30.9.5 $\mathsf{XM} \to (\mathsf{pro}_R(p) \longleftrightarrow \neg\exists x.\ \mathrm{min}_{R,p}(x))$.

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 30.9.6 $\mathsf{XM} \rightarrow (\neg(\exists x.\ \mathsf{pro}_R(x)) \longleftrightarrow \forall p.\ (\exists x. px) \rightarrow \exists x.\ \mathrm{min}_{R,p}(x))$.

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 30.9.5 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 30.9.5 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 30.9.7 (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 30.9.4 and 30.9.6. ∎

The above proofs gives us ample opportunity to contemplate about the role of XM in proofs. An interesting example is Fact 30.9.3, where XM is used to show that an element is either well-founded or progressive.

30.10 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 30.10.1 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 30.10.2 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^+.

30.11 Notes

The inductive definition of the well-founded points of a relation appears in Aczel [1] in a set-theoretic setting. Nordström [26] 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.

31 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 propositional discrimination restriction in the presence of the impredicative universe of propositions results in inconsistency.

31.1 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

$$\mathsf{T}\,\bot\,(\lambda a.\,\textsc{match}\;a\;[\,])$$

describes an **atomic tree** not having subtrees. Given two trees t_1 and t_2, the term

$$\mathsf{T}\,\mathsf{B}\,(\lambda b.\,\textsc{match}\;b\;[\,\mathsf{true} \Rightarrow t_1 \mid \mathsf{false} \Rightarrow t_2\,])$$

describes a tree having exactly t_1 and t_2 as subtrees where the boolean values are used as labels. The term

$$\mathsf{T}\,\mathsf{N}\,(\lambda\_.\,\mathsf{T}\bot\,(\lambda h.\,\textsc{match}\;h\;[\,])$$

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 §5.3. 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 31.1.1 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.

31.2 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 propositional discrimination 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 for numbers (§ 14.1)). It seems that the values of computational propositions are always wellfounded.

31.3 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 propositional discrimination restriction prevents us from defining an analogous subtree predicate for propositional Aczel trees (since the target 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 31.3.1 (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 31.3.2 (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 31.3.3 (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 31.3.4 $\neg\exists R^{\mathcal{T}_\mathsf{p}\to\mathcal{T}_\mathsf{p}\to\mathbb{P}}.\ \forall sXf.\ Rs(\mathsf{T_p}\,Xf) \longleftrightarrow \exists x.\ fx = s.$

Proof Let $R^{\mathcal{T}_\mathsf{p}\to\mathcal{T}_\mathsf{p}\to\mathbb{P}}$ 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 $u_\mathsf{p} := \mathsf{T_p}\,\mathcal{T}_\mathsf{p}\,(\lambda s.s)$ satisfies Ruu, it suffices to prove $\forall s.\neg Rss$. We can do this by induction on s (using $\mathsf{E}_{\mathcal{T}_\mathsf{p}}$) following the proof for computational Aczel trees (Fact 31.3.2). ■

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 propositional discrimination restriction) and for computational Aczel trees we cannot define a universal tree (because of the dominance restriction).

Exercise 31.3.5 Suppose you are allowed exactly one violation of the propositional discrimination restriction. Give a proof of falsity.

31.4 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 31.4.1 (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 31.3.4 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 propositional discrimination restriction since the proposition $\exists x. fx = s$ is encoded with E into a proof of the proposition A. ■

Exercise 31.4.2 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 31.4.3 Show $\forall P^{\mathbb{P}}.\ P \neq \mathbb{P}.$

31.5 Excluded Middle Implies Proof Irrelevance

With Coquand's theorem we can show that the law of excluded middle implies proof irrelevance (see §5.2 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 31.5.2).

Theorem 31.5.1 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 31.5.2) 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 31.5.2 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 31.5.1.

31.6 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 31.6.1 Every type embeds into itself: $\forall X^{\mathbb{T}} : \mathcal{E}XX$.

Fact 31.6.2 $\forall XY^{\mathbb{T}} : \neg\mathcal{E}XY \to X \neq Y$.

Fact 31.6.3 $\mathbb{P}$ embeds into no proposition: $\forall P^{\mathbb{P}}.\ \neg\mathcal{E}\mathbb{P}P$.

Proof Follows with Coquand's theorem 31.4.1. ∎

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 31.6.4 (Irreflexivity) $\forall s^{\mathcal{T}}.\, s \notin s$.

Proof Analogous to the proof of Fact 31.3.2. ∎

Recall that we have to construct a contradiction. We embark on a little detour before we construct a universal tree. By Fact 31.6.1 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 31.6.1 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 31.6.5 (Hierachy) $\forall X^U.\ \neg\mathcal{E}UX$.

Exercise 31.6.6 Show $\forall X^U.\ X \neq U$ for all universes U.

Exercise 31.6.7 Let $i \neq j$. Show $\mathbb{T}_i \neq \mathbb{T}_j$.

Exercise 31.6.8 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 V

Appendices

Appendix: Typing Rules

We give the typing rules for graded universes $\mathbb{T}_0, \mathbb{T}_1, \ldots$ and $\mathbb{P} := \mathbb{T}_0$. In this formulation $\mathbb{P} = \mathbb{T}_0$ requires one extra rule distinguishing it from the other graded universes. The extra rule says that $\mathbb{P} = \mathbb{T}_0$ admits impredicative function types.

$$\frac{}{\vdash \mathbb{T}_i : \mathbb{T}_{i+1}} \qquad \frac{\vdash u : \mathbb{T}_i \qquad x : u \vdash v : \mathbb{T}_i}{\vdash \forall x : u.v \;:\; \mathbb{T}_i} \qquad \frac{\vdash u : \mathbb{T}_i \qquad x : u \vdash v : \mathbb{T}_0}{\vdash \forall x^u.v \;:\; \mathbb{T}_0}$$

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

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

$$\frac{\vdash s \;:\; \forall x_1 : t_1. \cdots \forall x_n : t_n.\ \mathbb{T}_i}{\vdash s \;:\; \forall x_1 : t_1. \cdots \forall x_n : t_n.\ \mathbb{T}_{i+1}} \; n \geq 0$$

- In practice, one writes just $\mathbb{P}$ and $\mathbb{T}$ and checks that all occurrences of $\mathbb{T}$ can be assigned universe levels $i \geq 1$. The particular universe level assignment does not matter, provided it type checks.
- The subtyping rule (appearing last) realizes $\mathbb{P} \subseteq \mathbb{T}_1 \subseteq \mathbb{T}_2 \subseteq \cdots$ with $n = 0$.
- We have $\nvdash \mathbb{T}_i : \mathbb{T}_i$.
- v^x_t is capture-free substitution.
- Computational equality $s \approx t$ is defined with reduction and α- and η-equivalence.
- Variable typings ($x : u$ before $\vdash$) are introduced by the rules for $\forall$ and λ and the patterns of defining equations.
- Simple function types $u \to v$ are notation for dependent function types $\forall x : u.v$ where x does not occur in v.
- Functions whose type ends with the universe $\mathbb{P}$ are called *predicates*.
- Functions whose type ends with a universe $\mathbb{T}_i$ with $i > 0$ are called *type functions*.

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 $n \ge 0$ parameters and $m \ge 0$ indices.
- **Index condition**: The **index variables** $y_1, \dots, y_m$ must be distinct and must not occur in $s_1, \dots, s_n$.
- **Propositional discrimination 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 target 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 propositional dis-

crimination 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

$$\begin{aligned}\mathcal{L} &: \mathbb{T} \to \mathbb{T}\\ \mathsf{cons} &: \forall X.\, X \to \mathcal{L}(X) \to \mathcal{L}(X)\\ \mathsf{E} &: \forall X. \forall p^{\mathcal{L}(X)\to\mathbb{T}}.\ \ \ldots\ \ \to \forall A.\, pA\\ \mathsf{E}\,Xp \cdots (\mathsf{cons}\,xA) &:= \ e\,xA(E \cdots A)\end{aligned}$$

$\mathcal{L}(X)$ has uniform parameter X.

Linear search types

$$\begin{aligned}T &: (\mathsf{N} \to \mathbb{P}) \to \mathsf{N} \to \mathbb{T}\\ \mathsf{C} &: \forall qn.\, (\neg qn \to Tq(\mathsf{S}n)) \to Tqn\\ \mathsf{E} &: \forall q. \forall p^{\mathsf{N}\to\mathbb{T}}.\ \ \ldots\ \ \to \forall n.\, Tqn \to pn\\ \mathsf{E}\,qp \cdots n\,(\mathsf{C}\varphi) &:= \ e\,n(\lambda a.E \cdots (\mathsf{S}n)(\varphi a))\end{aligned}$$

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 $n \ge 2$ 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 $n \ge 2$ 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 $n \ge 2$ 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: 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 $A := \textsc{fix}\, f\, x.\, \lambda y.\, \textsc{match}\ x\ [\, 0 \Rightarrow y \mid \mathsf{S}x \Rightarrow \mathsf{S}(fxy)\,]$.

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) $(\textsc{let}\ f = A\,1\ \textsc{in}\ f) = \mathsf{S}$

   (iv) $A = \lambda xy.Axy$

   (v) $A = \textsc{fix}\, f\, x.\, \textsc{match}\ x\ [\, 0 \Rightarrow \lambda y.y \mid \mathsf{S}x \Rightarrow \lambda y.\, \mathsf{S}(fxy)\,]$

Exercise 2.3 Prove the following propositions (tables, 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 table, proof term, coq script). Type the type arguments of the matching functions with $\mathbb{T}$ (rather than $\mathbb{P}$) if this is possible (propositional discrimination 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 table 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 table 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

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

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) $\mathsf{eq}\,x y \to \mathsf{eq}\,y x$

b) $\mathsf{eq}\,x y \to \mathsf{eq}\,y z \to \mathsf{eq}\,x z$

c) $\mathsf{eq}\,x y \to \mathsf{eq}\,(f x)\,(f y)$

d) $\neg\mathsf{eq}\,\top\,\bot$

e) $\neg\mathsf{eq}\,\mathsf{true}\,\mathsf{false}$

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 table 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 $eqb\,x\,y = \mathsf{true} \longleftrightarrow x = y$ using the eliminator for numbers. Use this result to show $\forall xy^{\mathsf{N}}.\ x = y \lor x \neq y$.

Exercise 4.6 (Boolean pigeonhole principle)

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

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 $\top$: $\forall xy^{\top}.\ x = y$.

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 $\mathsf{M}_\exists$ 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 $\lambda X^{\mathbb{T}}.\ \forall f g^{X \to X}\ \forall x y^{X}.\ f x = y \lor g y = x$.

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

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) $fxy = \mathsf{true} \ \longleftrightarrow\ x = y$

c) $fxy = (x - y) + (y - x)$

d) $fxy = \mathsf{true} \longleftrightarrow\ (x - y) + (y - x) \neq 0$

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) $\forall X^{\mathbb{T}}\, f^{X \to \mathsf{B}}\, x^{X}.\ \mathcal{D}(fx = \mathsf{true})$.

d) $\forall X^{\mathbb{T}}.\ \mathcal{D}(X) \Leftrightarrow \Sigma b^{\mathsf{B}}.\ X \Leftrightarrow b = \mathsf{true}$.

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 $\mathcal{B}\ (X + Y)\ (\mathsf{sig}\,(\lambda b^{\mathsf{B}}.\ \textsc{if}\ b\ \textsc{then}\ X\ \textsc{else}\ Y))$.

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 $f(\mathsf{L\,I}) = \mathsf{true}$ and $f(\mathsf{R\,I}) = \mathsf{false}$. Prove $\neg\,\mathsf{PI}$. Why can't you define f inductively?

Exercise 7.6 (Set extensionality)

We define *set extensionality* as $\mathsf{SE} := \forall X^{\mathbb{T}}\, \forall pq^{X \to \mathbb{P}}.\ (\forall x.\ px \longleftrightarrow qx) \to p = q$. Prove the following:

a) $\mathsf{FE} \to \mathsf{PE} \to \mathsf{SE}$.

b) $\mathsf{SE} \to \mathsf{PE}$.

c) $\mathsf{SE} \to p - (q \cup r) = (p - q) \cap (p - r)$.

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 $(x \le y) := (x - y = 0)$. 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) $x \le y \to x + (y - x) = y$

g) $(x \le y) + (y < x)$

h) $\neg(y \le x) \to x < y$

i) $x \le y \longleftrightarrow \exists z.\, x + z = y$

j) $x \le x + y$

k) $x \le \mathsf{S}x$

l) $x + y \le x \to y = 0$

m) $x \le 0 \to x = 0$

n) $x \le x$

o) $x \le y \to y \le z \to x \le z$

p) $x \le y \to y \le x \to x = y$

q) $x \le y < z \to x < z$

r) $\neg(x < 0)$

s) $\neg(x + y < x)$

t) $\neg(x < x)$

u) $x \le y \to x \le y + z$

v) $x \le y \to x \le \mathsf{S}y$

w) $x < y \to x \le y$

x) $\neg(x < y) \to \neg(y < x) \to x = y$

y) $x \le y \le \mathsf{S}x \to x = y \lor y = \mathsf{S}x$

z) $x + y \le x + z \to y \le z$

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 $\forall xy.\,(x \le y) + (y < x)$.

b) Prove a complete induction lemma.

c) Prove $\forall xy.\Sigma ab.\, x = a \cdot \mathsf{S}y + b \;\wedge\; b \le y$ using complete induction and repeated subtraction.

d) Formulate the procedural specification

$$\begin{aligned} &f : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ &f\,x\,y \;:=\; \text{IF } \ulcorner x \le y \urcorner \text{ THEN } x \text{ ELSE } f\,(x - \mathsf{S}y\;y)\;y \end{aligned}$$

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 $\forall xy.\, fxy \le y$.

   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

$$(\forall xy.\ \gamma xy(fxy)) \quad\longleftrightarrow\quad \forall xy.\ fxy = \text{IF } \ulcorner x \le y \urcorner \text{ THEN } 0 \text{ ELSE } \mathsf{S}(f(x - \mathsf{S}y)y)$$

d) Consider the function

$$\begin{aligned} &f : \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\ &\quad f0yb \ := \ 0 \\ &f(\mathsf{S}x)yb \ := \ \text{IF } \ulcorner b = y \urcorner \text{ THEN } \mathsf{S}(fxy0) \text{ ELSE } fxy(\mathsf{S}b) \end{aligned}$$

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

$$\delta x y \;:=\; (\mathsf{least}\,p y \wedge y \leq x) \vee (y = x \wedge \mathsf{safe}\,p x)$$

Understand and prove the following:

a) $\forall n x y.\;\; p n \to n \leq x \to \delta x y \to \mathsf{least}\,p y$ *soundness*

b) $\forall x y y'.\;\; \delta x y \to \delta x y' \to y = y'$ *uniqueness*

c) $\forall x\, \Sigma y.\; \delta x y$ *satisfiability*

d) $\forall x.\;\; \delta x (G x)$ *correctness of G*

e) $\forall x.\;\; \delta x (L x 0)$ *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) $x \in f@A \;\longleftrightarrow\; \exists a.\; a \in A \wedge x = f a.$

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: $\mathsf{sum}\,A > \mathsf{len}\,A \to \Sigma x.\; x \in A \wedge x \geq 2$. 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) $\mathsf{nrep}\,A \to A \subseteq B \to \mathsf{len}\,B \leq \mathsf{len}\,A \to \mathsf{nrep}\,B$

e) $\mathsf{nrep}\,A \to A \subseteq B \to \mathsf{len}\,B \leq \mathsf{len}\,A \to B \equiv A$

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 $\mathsf{nrep}\,A \to A \subseteq B \to \mathsf{len}\,A \leq \mathsf{len}\,B$ 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) $pn \to m \leq n \to Tm$

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) $Tn \ \longleftrightarrow\ \exists k.\ k \geq n \wedge pk$

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 tables 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 tables 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 $\forall As.\ (A \vdash s) \to (A \dot\vdash s)$ and $\forall As.\ (A \dot\vdash s) \to (A \vdash \neg\neg s)$ 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 $\forall A.\ (\Sigma \alpha. \forall s \in A.\ \mathcal{E}\alpha s = \mathsf{true}) + \rho A$. Show the following:

a) $\lambda A.A \dot{\vdash} \bot$ is a refutation predicate.

b) $\mathcal{D}(\dot{\vdash} s)$.

c) $\dot{\vdash} s \Leftrightarrow \forall \alpha.\ \mathcal{E}\alpha s = \mathsf{true}$.

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 $\rho(A) \to \exists s.\ s \in A \wedge \mathcal{E}\alpha s = \mathsf{false}$.

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) in the literature. The technical terms are given in the order they appear first in the text.

- Discrimination
- Inductive function
- Target type function
- Propositional discrimination restriction
- Computational falsity elimination
- Index condition and index variables
- Reloading match

Appendix: Author’s Notes

Coq Wishlist

- Have equational function definitions with fixed arity and iterated and nested discrimination. This is realized in Agda already.
- Change type inference so that the universe Set is not derived as default. Set is not needed and smuggling it in as default is confusing.
- Have standard notations for sum and sigma types.
- An inductive predicate definition should say explicitly if it wants to lift the propositional discrimination restriction, and have the assistant check whether this can be granted. Conjunction can then be defined without granting computational discrimination.

Clarify

- Named functions come as constructors, reducible functions, or abstract functions.
  - Constructors have a fixed arity.
  - Reducible functions have a declared arity but may take additional arguments.
  - Reducible functions are defined with equations respecting their arity.
  - Reducible functions may be inductive or plain.
  - Certifying functions and theorems are abstract functions.
  - Abstract functions matter; reducible functions are overrated, often don’t want to know their defining equations.
- Term construction is incremental and type driven.
- Term construction is programmed with scripts.
- Term construction can be top down (backwards), bottom up (forward), and middle out (let).
- Coq’s tactic mode is term construction mode, not limited to propositional types.
- Proof tables to support term construction on paper.
- Propositions as types is foundation of mathematical reasoning, explains formulation, application, and proofs of theorems.

Editorial decisions

- Refer to all kinds of theorems as facts
- Assert certifying functions as facts when no name is given
- Define certifying functions in Coq with Fact/Qed

Maybe

- refinement types; numeral types as refinement type of N could be nice; but do we have applications? target types of certifying functions may be seen as refinement types and occur frequently

Work to do

- summary Part Basics
- chapter Introduction
- Coq development of certifying boolean solver (§ 24.12) needs update

Changelog

2024

- plain definitions explained with inductive function definitions
- η-equivalence and computational equality moved to Chapter 4
- new chapter *Axiom CT and Semidecidability*

2023

- chapters on certifying functions
- abstract syntax
- finite types, countable types, EWOs
- arithmetic recursion and Euclidean division
- vectors recursive and inductive, bijection
  (prompted by Lean's course by values recursion, via Yannick)
- numerals recursive and inductive, bijection
- chapter on indexed inductives with inductive equality, reflexive transitive closure, comparisons, numerals, vectors, PCP and bijections with recursive variants
- inductive GCD relation
- Revised structure, now 4 parts named Basics, More Basics, Case Studies, and Foundational Studies

2021

- MPCTT started
- switch to inductive and plain functions in Chapter 2
- unfolding functions and procedural specifications
- computational falsity elimination
- Andrej Dudenhefner was lead TA

2020

- $x \leq y$ as $x - y = 0$, game changer
- chapters on finite types and data types appeared

2019

- semi-decidability

Done

- introduce injections and bijections early
- introduce finite types based on lists early
- countable types.
- linear arithmetic as abstraction level
- abstract constants; equality and Euclidean division as simply typed examples
- construct certifying deciders in tactic mode and provide with abstract constants
- introduce indexed inductives in Part Basics
- constructor patterns always omit arguments of type constructor
- upgrade regular expressions
- harmonize recursive and inductive numeral types

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. Church's thesis and related axioms in Coq's type theory. In Christel Baier and Jean Goubault-Larrecq, editors, *29th EACSL Annual Conference on Computer Science Logic, CSL 2021, January 25-28, 2021, Ljubljana, Slovenia (Virtual Conference)*, volume 183 of *LIPIcs*, pages 21:1–21:19, 2021.

[11] Yannick Forster. *Computability in Constructive Type Theory*. PhD thesis, Saarland University, 2021.

[12] Yannick Forster, Dominik Kirst, and Gert Smolka. On synthetic undecidability in Coq, with an application to the Entscheidungsproblem. In *8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, New York, NY, USA, Jan 2019. ACM.

[13] Yannick Forster and Gert Smolka. Weak call-by-value lambda calculus as a model of computation in Coq. In *ITP 2017, Brasília, Brazil*, volume 10499 of *LNCS*, pages 189–206. Springer, 2017.

[14] 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.

[15] 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.

[16] Michael Hedberg. A coherence theorem for Martin-Löf's type theory. *Journal of Functional Programming*, 8(4):413–436, 1998.

[17] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators, an Introduction.* Cambridge University Press, 2008.

[18] Martin Hofmann and Thomas Streicher. The groupoid model refutes uniqueness of identity proofs. In *LICS 1994*, pages 208–212, 1994.

[19] 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.

[20] Dominik Kirst and Benjamin Peters. Gödel's theorem without tears - essential incompleteness in synthetic computability. In Bartek Klin and Elaine Pimentel, editors, *31st EACSL Annual Conference on Computer Science Logic, CSL 2023, February 13-16, 2023, Warsaw, Poland*, volume 252 of *LIPIcs*, pages 30:1–30:18, 2023.

[21] 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.

[22] Edmund Landau. *Grundlagen der Analysis: With Complete German-English Vocabulary*, volume 141. American Mathematical Soc., 1965.

[23] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[24] Yuri V. Matiyasevich. Martin Davis and Hilbert's Tenth Problem. In Eugenio G. Omodeo and Alberto Policriti, editors, *Martin Davis on Computability, Computational Logic, and Mathematical Foundations*, volume 10 of *Outstanding Contributions to Logic*, pages 35–54. Springer, 2016.

[25] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. *Mathematical aspects of computer science*, 1, 1967.

[26] Bengt Nordström. Terminating general recursion. *BIT Numerical Mathematics*, 28(3):605–619, Sep 1988.

[27] Raymond M. Smullyan and Melvin Fitting. *Set Theory and the Continuum Hypothesis*. Dover, 2010.

[28] A. S. Troelstra and H. Schwichtenberg. *Basic proof theory*. Cambridge University Press, 2nd edition, 2000.

[29] A.S. Troelstra and D. Van Dalen. *Constructivism in Mathematics*. Vol. 121 of Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1988.

[30] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[31] Louis Warren, Hannes Diener, and Maarten McKubre-Jordens. The drinker paradox and its dual. *CoRR*, abs/1805.06216, 2018.