Constructive Logic

Frank Pfenning
Carnegie Mellon University

Draft of December 28, 2000

Material for the course *Constructive Logic* at Carnegie Mellon University, Fall 2000. Material for this course is available at
`http://www.cs.cmu.edu/~fp/courses/logic/`.

Please send comments to `fp@cs.cmu.edu`

This material is in rough draft form and is likely to contain errors. Furthermore, citations are in no way adequate or complete. Please do not cite or distribute this document.

This work was supported in part by the University Education Council at Carnegie Mellon University and by NSF Grant CCR-9619684.

Contents

Chapter 1

Introduction

According to the Encyclopædia Britannica, logic is the study of propositions and their use in argumentation. From the breadth of this definition it is immediately clear that logic constitutes an important area in the disciplines of philosophy and mathematics. Logical tools and methods also play an essential role in the design, specification, and verification of computer hardware and software. It is these applications of logic in computer science which will be the focus of this course. In order to gain a proper understanding of logic and its relevance to computer science, we will need to draw heavily on the much older logical traditions in philosophy and mathematics. We will discuss some of the relevant history of logic and pointers to further reading throughout these notes. In this introduction, we give only a brief overview of the contents and approach of this class.

The course is divided into four parts:

I. Basic Concepts

II. Constructive Reasoning and Programming

III. Automatic Verification

IV. Properties of Logical Systems

In Part I we establish the basic vocabulary and systematically study propositions and proofs, mostly from a philosophical perspective. The treatment will be rather formal in order to permit an easy transition into computational applications. We will also discuss some properties of the logical systems we develop and strategies for proof search. We aim at a systematic account for the usual forms of logical expression, providing us with a flexible and thorough foundation for the remainder of the course. Exercises in this section will test basic understanding of logical connectives and how to reason with them.

In Part II we focus on constructive reasoning. This means we consider only proofs that describe algorithms. This turns out to be quite natural in the framework we have established in Part I. In fact, it may be somewhat

surprising that many proofs in mathematics today are *not* constructive in this sense. Concretely, we find that for a certain fragment of logic, constructive proofs correspond to functional programs and vice versa. More generally, we can extract functional programs from constructive proofs of their specifications. We often refer to constructive reasoning as *intuitionistic*, while non-constructive reasoning is *classical*. Exercises in this part explore the connections between proofs and programs, and between theorem proving and programming.

In Part III we study fragments of logic for which the question whether a proposition is true of false can be effectively decided by an algorithm. Such fragments can be used to specify some aspects of the behavior of software or hardware and then automatically verify them. A key technique here is model-checking that exhaustively explores the truth of a proposition over a finite state space. Model-checking and related methods are routinely used in industry, for example, to support hardware design by detecting design flaws at an early stage in the development cycle.

In Part IV we look more deeply at properties of logical system of the kind we developed and applied in Parts I–III. Among the questions we consider is the relation between intuitionistic and classical reasoning, and the soundness and completeness of various algorithms for proof search.

There are several related goals for this course. The first is simply that we would like students to gain a good working knowledge of constructive logic and its relation to computation. This includes the translation of informally specified problems to logical language, the ability to recognize correct proofs and construct them. The skills further include writing and inductively proving the correctness of recursive programs.

The second goals concerns the transfer of this knowledge to other kinds of reasoning. We will try to illuminate logic and the underlying philosophical and mathematical principles from various points of view. This is important, since there are many different kinds of logics for reasoning in different domains or about different phenomena[1], but there are relatively few underlying philosophical and mathematical principles. Our second goal is to teach these principles so that students can apply them in different domains where rigorous reasoning is required.

A third goal relates to specific, important applications of logic in the practice of computer science. Examples are the design of type systems for programming languages, specification languages, or verification tools for finite-state systems. While we do not aim at teaching the use of particular systems or languages, students should have the basic knowledge to quickly learn them, based on the materials presented in this class.

These learning goals present different challenges for students from different disciplines. Lectures, recitations, exercises, and the study of these notes are all necessary components for reaching them. These notes do not cover all aspects of the material discussed in lecture, but provide a point of reference for defini-

[1] for example: classical, intuitionistic, modal, second-order, temporal, belief, non-monotonic, linear, relevance, authentication, ...

tions, theorems, and motivating examples. Recitations are intended to answer students' questions and practice problem solving skills that are critical for the homework assignments. Exercises are a combination of written homework to be handed at lecture and theorem proving or programming problems to be submitted electronically using the software written in support of the course. An introduction to this software is included in these notes, a separate manual is available with the on-line course material.

Chapter 2

Propositional Logic

The goal of this chapter is to develop the two principal notions of logic, namely propositions and proofs. There is no universal agreement about the proper foundations for these notions. One approach, which has been particularly successful for applications in computer science, is to understand the meaning of a proposition by understanding its proofs. In the words of Martin-Löf [ML96, Page 27]:

> *The meaning of a proposition is determined by [...] what counts as a verification of it.*

In this chapter we apply Martin-Löf's approach, which follows a rich philosophical tradition, to explain the basic propositional connectives. We will see later that universal and existential quantifiers and types such as natural numbers, lists, or trees naturally fit into the same framework.

2.1 Judgments and Propositions

The cornerstone of Martin-Löf's foundation of logic is a clear separation of the notions of judgment and proposition. A *judgment* is something we may know, that is, an object of knowledge. A judgment is *evident* if we in fact know it.

We make a judgment such as "*it is raining*", because we have evidence for it. In everyday life, such evidence is often immediate: we may look out the window and see that it is raining. In logic, we are concerned with situation where the evidence is indirect: we deduce the judgment by making correct inferences from other evident judgments. In other words: a judgment is evident if we have a proof for it.

The most important judgment form in logic is "A *is true*", where A is a proposition. In order to reason correctly, we therefore need a second judgment form "A *is a proposition*". But there are many others that have been studied extensively. For example, "A *is false*", "A *is true at time* t" (from temporal

logic), "*A is necessarily true*" (from modal logic), "*program M has type* τ" (from programming languages), etc.

Returning to the first two judgments, let us try to explain the meaning of conjunction. We write *A prop* for the judgment "*A is a proposition*" and *A true* for the judgment "*A is true*" (presupposing that *A prop*). Given propositions A and B, we want to form the compound proposition "*A and B*", written more formally as $A \wedge B$. We express this in the following inference rule:

$$\frac{A\ prop \qquad B\ prop}{A \wedge B\ prop}\,\wedge F$$

This rule allows us to conclude that $A \wedge B$ *prop* if we already know that *A prop* and *B prop*. In this inference rule, A and B are *schematic variables*, and $\wedge F$ is the name of the rule (which is short for "conjunction formation"). The general form of an inference rule is

$$\frac{J_1\ \ldots\ J_n}{J}\ name$$

where the judgments $J_1, \ldots, J_n$ are called the *premises*, the judgment J is called the *conclusion*. In general, we will use letters J to stand for judgments, while A, B, and C are reserved for propositions.

Once the rule of conjunction formation ($\wedge F$) has been specified, we know that $A \wedge B$ is a proposition, if A and B are. But we have not yet specified what it *means*, that is, what counts as a verification of $A \wedge B$. This is accomplished by the following inference rule:

$$\frac{A\ true \qquad B\ true}{A \wedge B\ true}\,\wedge I$$

Here the name $\wedge I$ stands for "conjunction introduction", since the conjunction is introduced in the conclusion. We take this as specifying the meaning of $A \wedge B$ completely. So what can be deduce if we know that $A \wedge B$ is true? By the above rule, to have a verification for $A \wedge B$ means to have verifications for A and B. Hence the following two rules are justified:

$$\frac{A \wedge B\ true}{A\ true}\,\wedge E_L \qquad\qquad \frac{A \wedge B\ true}{B\ true}\,\wedge E_R$$

The name $\wedge E_L$ stands for "left conjunction elimination", since the conjunction in the premise has been eliminated in the conclusion. Similarly $\wedge E_R$ stands for "right conjunction elimination".

We will later see what precisely is required in order to guarantee that the formation, introduction, and elimination rules for a connective fit together correctly. For now, we will informally argue the correctness of the elimination rules.

As a second example we consider the proposition "*truth*" written as $\top$.

$$\frac{}{\top\ \mathit{prop}}\,\top F$$

Truth should always be true, which means its introduction rule has no premises.

$$\frac{}{\top\ \mathit{true}}\,\top I$$

Consequently, we have no information if we know $\top$ *true*, so there is no elimination rule.

A conjunction of two propositions is characterized by one introduction rule with two premises, and two corresponding elimination rules. We may think of truth as a conjunction of zero propositions. By analogy it should then have one introduction rule with zero premises, and zero corresponding elimination rules. This is precisely what we wrote out above.

2.2 Hypothetical Judgments

Consider the following derivation, for some arbitrary propositions A, B, and C:

$$\dfrac{\dfrac{A \wedge (B \wedge C)\ \mathit{true}}{B \wedge C\ \mathit{true}}\,\wedge E_R}{B\ \mathit{true}}\,\wedge E_L$$

Have we actually proved anything here? At first glance it seems that cannot be the case: B is an arbitrary proposition; clearly we should not be able to prove that it is true. Upon closer inspection we see that all inferences are correct, but the first judgment $A \wedge (B \wedge C)$ has not been justified. We can extract the following knowledge:

> *From the assumption that $A \wedge (B \wedge C)$ is true, we deduce that B must be true.*

This is an example of a *hypothetical judgment*, and the figure above is an *hypothetical derivation*. In general, we may have more than one assumption, so a hypothetical derivation has the form

$$\begin{array}{c} J_1 \quad \cdots \quad J_n \\ \vdots \\ J \end{array}$$

where the judgments $J_1, \ldots, J_n$ are unproven assumptions, and the judgment J is the conclusion. Note that we can always substitute a proof for any hypothesis J_i to eliminate the assumption. We call this the *substitution principle* for hypotheses.

Many mistakes in reasoning arise because dependencies on some hidden assumptions are ignored. When we need to be explicit, we write $J_1, \ldots, J_n \vdash J$ for the hypothetical judgment which is established by the hypothetical derivation above. We may refer to $J_1, \ldots, J_n$ as the antecedents and J as the succedent of the hypothetical judgment.

One has to keep in mind that hypotheses may be used more than once, or not at all. For example, for arbitrary propositions A and B,

$$\dfrac{\dfrac{A \wedge B\ \mathit{true}}{B\ \mathit{true}}\wedge E_R \qquad \dfrac{A \wedge B\ \mathit{true}}{A\ \mathit{true}}\wedge E_L}{B \wedge A\ \mathit{true}}\wedge I$$

can be seen a hypothetical derivation of $A \wedge B\ \mathit{true} \vdash B \wedge A\ \mathit{true}$.

With hypothetical judgments, we can now explain the meaning of implication "*A implies B*" or "*if A then B*" (more formally: $A \supset B$). First the formation rule:

$$\dfrac{A\ \mathit{prop} \qquad B\ \mathit{prop}}{A \supset B\ \mathit{prop}}\supset F$$

Next, the introduction rule: $A \supset B$ is true, if B is true under the assumption that A is true.

$$\dfrac{\dfrac{}{A\ \mathit{true}}\,u}{\begin{array}{c}\vdots\\ B\ \mathit{true}\end{array}}{\scriptstyle \supset I^u}$$
$$A \supset B\ \mathit{true}$$

The tricky part of this rule is the label u. If we omit this annotation, the rule would read

$$\dfrac{\begin{array}{c}A\ \mathit{true}\\ \vdots\\ B\ \mathit{true}\end{array}}{A \supset B\ \mathit{true}}\supset I$$

which would be incorrect: it looks like a derivation of $A \supset B\ \mathit{true}$ from the hypothesis $A\ \mathit{true}$. But the assumption $A\ \mathit{true}$ is introduced in the process of proving $A \supset B\ \mathit{true}$; the conclusion should not depend on it! Therefore we label uses of the assumption with a new name u, and the corresponding inference which introduced this assumption into the derivation with the same label u.

As a concrete example, consider the following proof of $A \supset (B \supset (A \wedge B))$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A\ \mathit{true}}\,u \qquad \dfrac{}{B\ \mathit{true}}\,w}{A \wedge B\ \mathit{true}}\wedge I}{B \supset (A \wedge B)\ \mathit{true}}{\scriptstyle \supset I^w}}{A \supset (B \supset (A \wedge B))\ \mathit{true}}{\scriptstyle \supset I^u}$$

Note that this derivation is not hypothetical (it does not depend on any assumptions). The assumption $A\ true$ labeled u is discharged in the last inference, and the assumption $B\ true$ labeled w is discharged in the second-to-last inference. It is critical that a discharged hypothesis is no longer available for reasoning, and that all labels introduced in a derivation are distinct.

Finally, we consider what the elimination rule for implication should say. By the only introduction rule, having a proof of $A \supset B\ true$ means that we have a hypothetical proof of $B\ true$ from $A\ true$. By the substitution principle, if we also have a proof of $A\ true$ then we get a proof of $B\ true$.

$$\frac{A \supset B\ true \qquad A\ true}{B\ true}\ \supset E$$

This completes the rule concerning implication.

With the rules so far, we can write out proofs of simple properties concerning conjunction and implication. The first expresses that conjunction is commutative—intuitively, an obvious property.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B\ true}\,u}{B\ true}\wedge E_R \qquad \dfrac{\dfrac{}{A \wedge B\ true}\,u}{A\ true}\wedge E_L}{B \wedge A\ true}\wedge I^u}{(A \wedge B) \supset (B \wedge A)\ true}\supset I$$

When we construct such a derivation, we generally proceed by a combination of bottom-up and top-down reasoning. The next example is a distributivity law, allowing us to move implications over conjunctions. This time, we show the partial proofs in each step. Of course, other sequences of steps in proof constructions are also possible.

$$\begin{array}{c}\vdots\\ (A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ true\end{array}$$

First, we use the implication introduction rule bottom-up.

$$\dfrac{\dfrac{}{A \supset (B \wedge C)\ true}\,u}{\begin{array}{c}\vdots\\ (A \supset B) \wedge (A \supset C)\ true\end{array}}{\supset I^u}$$

Next, we use the conjunction introduction rule bottom-up.

$$\dfrac{\dfrac{\dfrac{}{A \supset (B \wedge C)\ true}\,u \qquad \dfrac{}{A \supset (B \wedge C)\ true}\,u}{\begin{array}{cc} \vdots & \vdots \\ A \supset B\ true & A \supset C\ true \end{array}}{(A \supset B) \wedge (A \supset C)\ true}\,\wedge I}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ true}\,\supset I^u$$

We now pursue the left branch, again using implication introduction bottom-up.

$$\dfrac{\dfrac{\dfrac{}{A \supset (B \wedge C)\ true}\,u \qquad \dfrac{}{A\ true}\,w}{\dfrac{\vdots}{B\ true}}{A \supset B\ true}\,\supset I^w \qquad \dfrac{\dfrac{}{A \supset (B \wedge C)\ true}\,u}{\vdots}{A \supset C\ true}}{(A \supset B) \wedge (A \supset C)\ true}\,\wedge I}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ true}\,\supset I^u$$

Note that the hypothesis $A\ true$ is available only in the left branch, but not in the right one: it is discharged at the inference $\supset I^w$. We now switch to top-down reasoning, taking advantage of implication elimination.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \supset (B \wedge C)\ true}\,u \qquad \dfrac{}{A\ true}\,w}{B \wedge C\ true}\,\supset E}{\dfrac{\vdots}{B\ true}}{A \supset B\ true}\,\supset I^w \qquad \dfrac{\dfrac{}{A \supset (B \wedge C)\ true}\,u}{\vdots}{A \supset C\ true}}{(A \supset B) \wedge (A \supset C)\ true}\,\wedge I}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ true}\,\supset I^u$$

Now we can close the gap in the left-hand side by conjunction elimination.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \quad \dfrac{}{A\ \mathit{true}}\,w}{B \wedge C\ \mathit{true}}\supset E}{B\ \mathit{true}}\wedge E_L}{A \supset B\ \mathit{true}}\supset I^w \qquad \begin{array}{c}\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u\\ \vdots \\ A \supset C\ \mathit{true}\end{array}}{(A \supset B) \wedge (A \supset C)\ \mathit{true}}\wedge I}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ \mathit{true}}\supset I^u$$

The right premise of the conjunction introduction can be filled in analogously. We skip the intermediate steps and only show the final derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \quad \dfrac{}{A\ \mathit{true}}\,w}{B \wedge C\ \mathit{true}}\supset E}{B\ \mathit{true}}\wedge E_L}{A \supset B\ \mathit{true}}\supset I^w \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \quad \dfrac{}{A\ \mathit{true}}\,v}{B \wedge C\ \mathit{true}}\supset E}{C\ \mathit{true}}\wedge E_R}{A \supset C\ \mathit{true}}\supset I^v}{(A \supset B) \wedge (A \supset C)\ \mathit{true}}\wedge I}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ \mathit{true}}\supset I^u$$

2.3 Disjunction and Falsehood

So far we have explained the meaning of conjunction, truth, and implication. The disjunction "*A or B*" (written as $A \vee B$) is more difficult, but does not require any new judgment forms.

$$\dfrac{A\ \mathit{prop} \qquad B\ \mathit{prop}}{A \vee B\ \mathit{prop}}\vee F$$

Disjunction is characterized by two introduction rules: $A \vee B$ is true, if either A or B is true.

$$\dfrac{A\ \mathit{true}}{A \vee B\ \mathit{true}}\vee I_L \qquad\qquad \dfrac{B\ \mathit{true}}{A \vee B\ \mathit{true}}\vee I_R$$

Now it would be incorrect to have an elimination rule such as

$$\dfrac{A \vee B\ \mathit{true}}{A\ \mathit{true}}\vee E_L?$$

because even if we know that $A \vee B$ is true, we do not know whether the disjunct A or the disjunct B is true. Concretely, with such a rule we could derive the

truth of *every* proposition A as follows:

$$\dfrac{\dfrac{\dfrac{}{B\ true}\,u}{B \supset B\ true}\supset I^u \qquad \dfrac{\dfrac{\dfrac{}{B \supset B\ true}\,w}{A \vee (B \supset B)\ true}\vee I_R}{A\ true}\vee E_L?}{A\ true}\supset I^w \quad \supset E$$

Thus we take a different approach. If we know that $A \vee B$ is true, we must consider two cases: A *true* and B *true*. If we can prove a conclusion C *true* in both cases, then C must be true! Written as an inference rule:

$$\dfrac{\begin{array}{ccc} & \dfrac{}{A\ true}\,u & \dfrac{}{B\ true}\,w \\ & \vdots & \vdots \\ A \vee B\ true & C\ true & C\ true \end{array}}{C\ true}\vee E^{u,w}$$

Note that we use once again the mechanism of hypothetical judgments. In the proof of the second premise we may use the assumption A *true* labeled u, in the proof of the third premise we may use the assumption B *true* labeled w. Both are discharged at the disjunction elimination rule.

Let us justify the conclusion of this rule more explicitly. By the first premise we know $A \vee B$ *true*. The premises of the two possible introduction rules are A *true* and B *true*. In case A *true* we conclude C *true* by the substitution principle and the second premise: we substitute the proof of A *true* for any use of the assumption labeled u in the hypothetical derivation. The case for B *true* is symmetric, using the hypothetical derivation in the third premise.

Because of the complex nature of the elimination rule, reasoning with disjunction is more difficult than with implication and conjunction. As a simple example, we prove the commutativity of disjunction.

$$\begin{array}{c} \vdots \\ (A \vee B) \supset (B \vee A)\ true \end{array}$$

We begin with an implication introduction.

$$\dfrac{\begin{array}{c} \dfrac{}{A \vee B\ true}\,u \\ \vdots \\ B \vee A\ true \end{array}}{(A \vee B) \supset (B \vee A)\ true}\supset I^u$$

At this point we cannot use either of the two disjunction introduction rules. The problem is that neither B nor A follow from our assumption $A \vee B$! So first we need to distinguish the two cases via the rule of disjunction elimination.

$$\dfrac{\dfrac{}{A \vee B\ \mathit{true}}\,u \qquad \begin{array}{c}\dfrac{}{A\ \mathit{true}}\,v \\ \vdots \\ B \vee A\ \mathit{true}\end{array} \qquad \begin{array}{c}\dfrac{}{B\ \mathit{true}}\,w \\ \vdots \\ B \vee A\ \mathit{true}\end{array}}{\dfrac{B \vee A\ \mathit{true}}{(A \vee B) \supset (B \vee A)\ \mathit{true}}\,\supset I^u}\,\vee E^{v,w}$$

The assumption labeled u is still available for each of the two proof obligations, but we have omitted it, since it is no longer needed.

Now each gap can be filled in directly by the two disjunction introduction rules.

$$\dfrac{\dfrac{}{A \vee B\ \mathit{true}}\,u \qquad \dfrac{\dfrac{}{A\ \mathit{true}}\,v}{B \vee A\ \mathit{true}}\,\vee I_R \qquad \dfrac{\dfrac{}{B\ \mathit{true}}\,w}{B \vee A\ \mathit{true}}\,\vee I_L}{\dfrac{B \vee A\ \mathit{true}}{(A \vee B) \supset (B \vee A)\ \mathit{true}}\,\supset I^u}\,\vee E^{v,w}$$

This concludes the discussion of disjunction. Falsehood (written as $\bot$, sometimes called absurdity) is a proposition that should have no proof! Therefore there are no introduction rules, although we of course have the standard formation rule.

$$\dfrac{}{\bot\ \mathit{prop}}\,\bot F$$

Since there cannot be a proof of $\bot\ \mathit{true}$, it is sound to conclude the truth of any arbitrary proposition if we know $\bot\ \mathit{true}$. This justifies the elimination rule

$$\dfrac{\bot\ \mathit{true}}{C\ \mathit{true}}\,\bot E$$

We can also think of falsehood as a disjunction between zero alternatives. By analogy with the binary disjunction, we therefore have zero introduction rules, and an elimination rule in which we have to consider zero cases. This is precisely the $\bot E$ rule above.

From this is might seem that falsehood it useless: we can never prove it. This is correct, except that we might reason from contradictory hypotheses! We will see some examples when we discuss negation, since we may think of the proposition "*not* A" (written $\neg A$) as $A \supset \bot$. In other words, $\neg A$ is true precisely if the assumption $A\ \mathit{true}$ is contradictory because we could derive $\bot\ \mathit{true}$.

2.4 Notational Definition

The judgments, propositions, and inference rules we have defined so far collectively form a system of *natural deduction*. It is a minor variant of a system introduced by Gentzen [Gen35]. One of his main motivations was to devise rules that model mathematical reasoning as directly as possible, although clearly in much more detail than in a typical mathematical argument.

We now consider how to define negation. So far, the meaning of any logical connective has been defined by its introduction rules, from which we derived its elimination rules. The definitions for all the connectives are *orthogonal*: the rules for any of the connectives do not depend on any other connectives, only on basic judgmental concepts. Hence the meaning of a compound proposition depends only on the meaning of its constituent propositions. From the point of view of understanding logical connectives this is a critical property: to understand disjunction, for example, we only need to understand its introduction rules and not any other connectives.

A frequently proposed introduction rule for "*not* A" (written $\neg A$) is

$$\dfrac{\dfrac{}{A\ \mathit{true}}\,u \\ \vdots \\ \bot\ \mathit{true}}{\neg A\ \mathit{true}}\,\neg I^u?$$

In words: $\neg A$ is true if the assumption that A is true leads to a contradiction. However, this is not a satisfactory introduction rule, since the premise relies the meaning of $\bot$, violating orthogonality among the connectives. There are several approaches to removing this dependency. One is to introduce a new *judgment*, "A *is false*", and reason explicitly about truth and falsehood. Another employs schematic judgments, which we consider when we introduce universal and existential quantification.

Here we pursue a third alternative: for arbitrary propositions A, we think of $\neg A$ as a syntactic abbreviation for $A \supset \bot$. This is called a *notational definition* and we write

$$\neg A = A \supset \bot.$$

This notational definition is schematic in the proposition A. Implicit here is the formation rule

$$\dfrac{A\ \mathit{prop}}{\neg A\ \mathit{prop}}\,\neg F$$

We allow silent expansion of notational definitions. As an example, we prove

that A and $\neg A$ cannot be true simultaneously.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge \neg A \ \mathit{true}}\,u}{\neg A \ \mathit{true}}\wedge E_R \quad \dfrac{\dfrac{}{A \wedge \neg A \ \mathit{true}}\,u}{A \ \mathit{true}}\wedge E_L}{\bot \ \mathit{true}}\supset E}{\neg(A \wedge \neg A) \ \mathit{true}}\supset I^u$$

We can only understand this derivation if we keep in mind that $\neg A$ stands for $A \supset \bot$, and that $\neg(A \wedge \neg A)$ stands for $(A \wedge \neg A) \supset \bot$.

As a second example, we show the proof that $A \supset \neg\neg A$ is true.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg \ A \ \mathit{true}}\,w \quad \dfrac{}{A \ \mathit{true}}\,u}{\bot \ \mathit{true}}\supset E}{\neg\neg A \ \mathit{true}}\supset I^w}{A \supset \neg\neg A \ \mathit{true}}\supset I^u$$

Next we consider $A \vee \neg A$, the so-called *"law" of excluded middle*. It claims that every proposition is either true or false. This, however, contradicts our definition of disjunction: we may have evidence neither for the truth of A, nor for the falsehood of A. Therefore we cannot expect $A \vee \neg A$ to be true unless we have more information about A.

One has to be careful how to interpret this statement, however. There are many propositions A for which it is indeed the case that we know $A \vee \neg A$. For example, $\top \vee (\neg\top)$ is clearly true because $\top$ *true*. Similarly, $\bot \vee (\neg\bot)$ is true because $\neg\bot$ is true. To make this fully explicit:

$$\dfrac{\dfrac{}{\top \ \mathit{true}}\top I}{\top \vee (\neg\top) \ \mathit{true}}\vee I_L \qquad \dfrac{\dfrac{\dfrac{}{\bot \ \mathit{true}}\,u}{\neg\bot \ \mathit{true}}\supset I^u}{\bot \vee (\neg\bot) \ \mathit{true}}\vee I_R$$

In mathematics and computer science, many basic relations satisfy the law of excluded middle. For example, we will be able to show that for any two numbers k and n, either $k < n$ or $\neg(k < n)$. However, this requires proof, because for more complex A propositions we may not know if A *true* or $\neg A$ *true*. We will return to this issue later in this course.

At present we do not have the tools to show formally that $A \vee \neg A$ should not be true for arbitrary A. A proof attempt with our generic proof strategy (reason from the bottom up with introduction rules and from the top down with elimination rules) fails quickly, no matter which introduction rule for disjunction

we start with.

$$\dfrac{\dfrac{\vdots}{A\ true}}{A \vee \neg A\ true}\vee I_L \qquad \dfrac{\dfrac{\dfrac{}{A\ true}\,u}{\vdots}}{\dfrac{\dfrac{\dfrac{\bot\ true}{\neg A\ true}\supset I^u}{A \vee \neg A\ true}\vee I_R}}$$

We will see that this failure is in fact sufficient evidence to know that $A \vee \neg A$ is not true for arbitrary A.

2.5 Derived Rules of Inference

One popular device for shortening derivations is to introduce *derived rules of inference*. For example,

$$\dfrac{A \supset B\ true \qquad B \supset C\ true}{A \supset C\ true}$$

is a derived rule of inference. Its derivation is the following:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A\ true}\,u \qquad A \supset B\ true}{B\ true}\supset E \qquad B \supset C\ true}{C\ true}\supset E}{A \supset C\ true}\supset I^u$$

Note that this is simply a hypothetical derivation, using the premises of the derived rule as assumptions. In other words, a derived rule of inference is nothing but an evident hypothetical judgment; its justification is a hypothetical derivation.

We can freely use derived rules in proofs, since any occurrence of such a rule can be expanded by replacing it with its justification.

A second example of notational definition is logical equivalence "*A if and only if B*" (written $A \equiv B$). We define

$$(A \equiv B) = (A \supset B) \wedge (B \supset A).$$

That is, two propositions A and B are logically equivalent if A implies B and B implies A. Under this definition, the following become derived rules of inference (see Exercise 2.1). They can also be seen as introduction and elimination rules

for logical equivalence (whence their names).

$$\dfrac{\dfrac{}{A\ true}\,u \qquad \dfrac{}{B\ true}\,w}{\begin{array}{cc} \vdots & \vdots \\ B\ true & A\ true \end{array}}{\equiv}I^{u,w}$$
$$A \equiv B\ true$$

$$\dfrac{A \equiv B\ true \qquad A\ true}{B\ true}{\equiv}E_L \qquad \dfrac{A \equiv B\ true \qquad B\ true}{A\ true}{\equiv}E_R$$

2.6 Logical Equivalences

We now consider several classes of logical equivalences in order to develop some intuitions regarding the truth of propositions. Each equivalence has the form $A \equiv B$, but we consider only the basic connectives and constants ($\wedge$, $\supset$, $\vee$, $\top$, $\bot$) in A and B. Later on we consider negation as a special case. We use some standard conventions that allow us to omit some parentheses while writing propositions. We use the following operator precedences

$$\neg > \wedge > \vee > \supset > \equiv$$

where $\wedge$, $\vee$, and $\supset$ are right associative. For example

$$\neg A \supset A \vee \neg\neg A \supset \bot$$

stands for

$$(\neg A) \supset ((A \vee (\neg(\neg A))) \supset \bot)$$

In ordinary mathematical usage, $A \equiv B \equiv C$ stands for $(A \equiv B) \wedge (B \equiv C)$; in the formal language we do not allow iterated equivalences without explicit parentheses in order to avoid confusion with propositions such as $(A \equiv A) \equiv \top$.

Commutativity. Conjunction and disjunction are clearly commutative, while implication is not.

(C1) $A \wedge B \equiv B \wedge A\ true$

(C2) $A \vee B \equiv B \vee A\ true$

(C3) $A \supset B$ is not commutative

Idempotence. Conjunction and disjunction are idempotent, while self-implication reduces to truth.

(I1) $A \wedge A \equiv A\ true$

(I2) $A \vee A \equiv A\ true$

(I3) $A \supset A \equiv \top\ true$

Interaction Laws. These involve two interacting connectives. In principle, there are left and right interaction laws, but because conjunction and disjunction are commutative, some coincide and are not repeated here.

(L1) $A \wedge (B \wedge C) \equiv (A \wedge B) \wedge C$ *true*

(L2) $A \wedge \top \equiv A$ *true*

(L3) $A \wedge (B \supset C)$ do not interact

(L4) $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$ *true*

(L5) $A \wedge \bot \equiv \bot$ *true*

(L6) $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$ *true*

(L7) $A \vee \top \equiv \top$ *true*

(L8) $A \vee (B \supset C)$ do not interact

(L9) $A \vee (B \vee C) \equiv (A \vee B) \vee C$ *true*

(L10) $A \vee \bot \equiv A$ *true*

(L11) $A \supset (B \wedge C) \equiv (A \supset B) \wedge (A \supset C)$ *true*

(L12) $A \supset \top \equiv \top$ *true*

(L13) $A \supset (B \supset C) \equiv (A \wedge B) \supset C$ *true*

(L14) $A \supset (B \vee C)$ do not interact

(L15) $A \supset \bot$ do not interact

(L16) $(A \wedge B) \supset C \equiv A \supset (B \supset C)$ *true*

(L17) $\top \supset C \equiv C$ *true*

(L18) $(A \supset B) \supset C$ do not interact

(L19) $(A \vee B) \supset C \equiv (A \supset C) \wedge (B \supset C)$ *true*

(L20) $\bot \supset C \equiv \top$ *true*

2.7 Summary of Judgments

Judgments.

A *prop*	A is a proposition
A *true*	Proposition A is true

Propositional Constants and Connectives. The following table summarizes the introduction and elimination rules for the propositional constants ($\top$, $\bot$) and connectives ($\wedge$, $\supset$, $\vee$). We omit the straightforward formation rules.

Introduction Rules	Elimination Rules
$\dfrac{A\ true \qquad B\ true}{A \wedge B\ true}\wedge I$	$\dfrac{A \wedge B\ true}{A\ true}\wedge E_L \qquad \dfrac{A \wedge B\ true}{B\ true}\wedge E_R$
$\dfrac{}{\top\ true}\top I$	*no* $\top E$ *rule*
$\dfrac{\begin{array}{c}\overline{A\ true}\,u \\ \vdots \\ B\ true\end{array}}{A \supset B\ true}\supset I^u$	$\dfrac{A \supset B\ true \qquad A\ true}{B\ true}\supset E$
$\dfrac{A\ true}{A \vee B\ true}\vee I_L \qquad \dfrac{B\ true}{A \vee B\ true}\vee I_R$	$\dfrac{A \vee B\ true \quad \begin{array}{c}\overline{A\ true}\,u\end{array} \quad \begin{array}{c}\overline{B\ true}\,w\end{array}}{\begin{array}{cc}\vdots & \vdots \\ C\ true & C\ true\end{array}}\vee E^{u,w}$
no $\bot I$ *rule*	$\dfrac{\bot\ true}{C\ true}\bot E$

Notational Definitions. We use the following notational definitions.

$\neg A$	$=$	$A \supset \bot$	not A
$A \equiv B$	$=$	$(A \supset B) \wedge (B \supset A)$	A if and only if B

2.8 A Linear Notation for Proofs

The two-dimensional format for rules of inference and deductions is almost universal in the literature on logic. Unfortunately, it is not well-suited for writing actual proofs of complex propositions, because deductions become very unwieldy. Instead with use a linearized format explained below. Furthermore, since logical symbols are not available on a keyboard, we use the following concrete syntax for propositions:

$A \equiv B$	`A <=> B`	A if and only if B
$A \supset B$	`A => B`	A implies B
$A \vee B$	`A | B`	A or B
$A \wedge B$	`A & B`	A and B
$\neg A$	`~ A`	not A

The operators are listed in order of increasing binding strength, and implication (`=>`), disjunction (`|`), and conjunction (`&`) associate to the right, just like the corresponding notation from earlier in this chapter.

The linear format is mostly straightforward. A proof is written as a sequence of judgments separated by semi-colon '`;`'. Later judgements must follow from earlier ones by simple applications of rules of inference. Since it can easily be verified that this is the case, explicit justifications of inferences are omitted. Since the only judgment we are interested in at the moment is the truth of a proposition, the judgment "A *true*" is abbreviated simply as "A".

The only additional notation we need is for hypothetical proofs. A hypothetical proof

$$\begin{array}{c} A\ \mathit{true} \\ \vdots \\ C\ \mathit{true} \end{array}$$

is written as `[A;...;C]`.

In other words, the hypothesis A is immediately preceded by a square bracket ('`[`'), followed by the lines representing the hypothetical proof of C, followed by a closing square bracket ('`]`'). So square brackets are used to delimit the scope of an assumption. If we need more than hypothesis, we nest this construct as we will see in the example below.

As an example, we consider the proof of $(A \supset B) \wedge (B \supset C) \supset (A \supset C)$ *true*. We show each stage in the proof during its natural construction, showing both the mathematical and concrete syntax, except that we omit the judgment "*true*" to keep the size of the derivation manageable. We write '`...`' to indicate that the following line has not yet been justified.

$$\begin{array}{l} \vdots \\ (A \supset B) \wedge (B \supset C) \supset (A \supset C) \end{array} \qquad \begin{array}{l} \texttt{...} \\ \texttt{(A => B) \& (B => C) => (A => C);} \end{array}$$

The first bottom-up step is an implication introduction. In the linear form, we use our notation for hypothetical judgments.

$$\dfrac{\dfrac{}{(A \supset B) \wedge (B \supset C)}\,u \\ \vdots \\ A \supset C}{(A \supset B) \wedge (B \supset C) \supset (A \supset C)}\,\supset I^u \qquad \begin{array}{l} \texttt{[(A => B) \& (B => C);} \\ \texttt{\ \ ...} \\ \texttt{\ \ A => C];} \\ \texttt{(A => B) \& (B => C) => (A => C);} \end{array}$$

Again, we proceed via an implication introduction. In the mathematical notation, the hypotheses are shown next to each other. In the linear notation, the second hypothesis `A` is nested inside the first, also making both of them available to fill the remaining gap in the proof.

$$\dfrac{\dfrac{\dfrac{}{(A \supset B) \wedge (B \supset C)}\,u \quad \dfrac{}{A}\,w}{\vdots}}{\dfrac{\dfrac{C}{A \supset C}\supset I^w}{(A \supset B) \wedge (B \supset C) \supset (A \supset C)}\supset I^u}$$

```
[ (A => B) & (B => C);
  [ A;
    ...
    C ];
  A => C ];
(A => B) & (B => C) => (A => C);
```

Now that the conclusion is atomic and cannot be decomposed further, we reason downwards from the hypotheses. In the linear format, we write the new line `A => B;` immediately below the hypothesis, but we could also have inserted it directly below `A;`. In general, the requirement is that the lines representing the premise of an inference rule must all come before the conclusion. Furthermore, lines cannot be used outside the hypothetical proof in which they appear, because their proof could depend on the hypothesis.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(A \supset B) \wedge (B \supset C)}\,u}{A \supset B}\wedge E_L \quad \dfrac{}{A}\,w}{\vdots}}{\dfrac{\dfrac{C}{A \supset C}\supset I^w}{(A \supset B) \wedge (B \supset C) \supset (A \supset C)}\supset I^u}$$

```
[ (A => B) & (B => C);
  A => B;
  [ A;
    ...
    C ];
  A => C ];
(A => B) & (B => C) => (A => C);
```

Nex we apply another straightforward top-down reasoning step. In this case, there is no choice on where to insert `B;`.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(A \supset B) \wedge (B \supset C)}\,u}{A \supset B}\wedge E_L \quad \dfrac{}{A}\,w}{B}\supset E}{\vdots}}{\dfrac{\dfrac{C}{A \supset C}\supset I^w}{(A \supset B) \wedge (B \supset C) \supset (A \supset C)}\supset I^u}$$

```
[ (A => B) & (B => C);
  A => B;
  [ A;
    B;
    ...
    C ];
  A => C ];
(A => B) & (B => C) => (A => C);
```

For the last two steps, we align the derivations vertically. The are both top-down steps (conjunction elimination followed by implication elimination).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(A \supset B) \wedge (B \supset C)}\,u}{B \supset C}\,\wedge E_R \qquad \dfrac{\dfrac{\dfrac{}{(A \supset B) \wedge (B \supset C)}\,u}{A \supset B}\,\wedge E_L \qquad \dfrac{}{A}\,w}{B}\,\supset E}{\vdots}}{C}}{A \supset C}\,\supset I^w}{(A \supset B) \wedge (B \supset C) \supset (A \supset C)}\,\supset I^u$$

```
[ (A => B) & (B => C);
  A => B;
  B => C;
  [ A;
    B;
    ...
    C ];
  A => C ];
(A => B) & (B => C) => (A => C);
```

In the step above we notice that subproofs may be shared in the linearized format, while in the tree format they appear more than once. In this case it is only the hypothesis $(A \supset B) \wedge (B \supset C)$ which is shared.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(A \supset B) \wedge (B \supset C)}\,u}{B \supset C}\,\wedge E_R \qquad \dfrac{\dfrac{\dfrac{}{(A \supset B) \wedge (B \supset C)}\,u}{A \supset B}\,\wedge E_L \qquad \dfrac{}{A}\,w}{B}\,\supset E}{C}\,\supset E}{A \supset C}\,\supset I^w}{(A \supset B) \wedge (B \supset C) \supset (A \supset C)}\,\supset I^u$$

```
[ (A => B) & (B => C);
  A => B;
  B => C;
  [ A;
    B;
    C ];
  A => C ];
(A => B) & (B => C) => (A => C);
```

In the last step, the linear derivation only changed in that we noticed that `C` already follows from two other lines and is therefore justified.

For other details of concrete syntax and usage of the proof-checking program available for this course, please refer to the on-line documentation available through the course home page.

2.9 Normal Deductions

The strategy we have used so far in proof search is easily summarized: we reason with introduction rules from the bottom up and with elimination rules from the top down, hoping that the two will meet in the middle. This description is somewhat vague in that it is not obvious how to apply it to complex rules such as disjunction elimination which involve formulas other than the principal one whose connective is eliminated.

To make this precise we introduce two new judgments

$A\uparrow$ A has a normal proof

$A\downarrow$ A has a neutral proof

We are primarily interest in normal proofs, which are those that our strategy can find. Neutral proofs represent an auxiliary concept (sometimes called an *extraction proof*) necessary for the definition of normal proofs.

We will define these judgments via rules, trying to capture the following intuitions:

1. A normal proof is either neutral, or proceeds by applying introduction rules to other normal proofs.

2. A neutral proof proceeds by applying elimination rules to hypotheses or other neutral proofs.

By construction, every A which has a normal (or neutral) proof is true. The converse, namely that every true A has a normal proof also holds, but is not at all obvious. We may prove this property later on, at least for a fragment of the logic.

First, a general rule to express that every neutral proof is normal.

$$\frac{A\downarrow}{A\uparrow}\downarrow\uparrow$$

Conjunction. The rules for conjunction are easily annotated.

$$\frac{A\uparrow \qquad B\uparrow}{A\wedge B\uparrow}\wedge I \qquad \frac{A\wedge B\downarrow}{A\downarrow}\wedge E_L \qquad \frac{A\wedge B\downarrow}{B\downarrow}\wedge E_R$$

Truth. Truth only has an introduction rule and therefore no neutral proof constructor.

$$\frac{}{\top\uparrow}\top I$$

Implication. Implication first fixes the idea that hypotheses are neutral, so the introduction rule refers to both normal and neutral deductions.

$$\dfrac{\dfrac{}{A\downarrow}\,u \\ \vdots \\ B\uparrow}{A\supset B\uparrow}\supset I^u \qquad \dfrac{A\supset B\downarrow \qquad A\uparrow}{B\downarrow}\supset E$$

The elimination rule is more difficult to understand. The principal premise (with the connective "$\supset$" we are eliminating) should have a neutral proof. The resulting derivation will once again be neutral, but we can only require the second premise to have a normal proof.

Disjunction. For disjunction, the introduction rules are straightforward. The elimination rule requires again the requires the principal premise to have a neutral proof. An the assumptions introduced in both branches are also neutral. In the end we can conclude that we have a normal proof of the conclusion, if we can find a normal proof in each premise.

$$\dfrac{A\uparrow}{A\vee B\uparrow}\vee I_L \qquad \dfrac{B\uparrow}{A\vee B\uparrow}\vee I_R \qquad \dfrac{A\vee B\downarrow \quad \begin{array}{c}\dfrac{}{A\downarrow}\,u\\ \vdots \\ C\uparrow\end{array} \quad \begin{array}{c}\dfrac{}{B\downarrow}\,w\\ \vdots \\ C\uparrow\end{array}}{C\uparrow}\vee E^{u,w}$$

Falsehood. Falsehood is analogous to the rules for disjunction. But since there are no introduction rules, there are no cases to consider in the elimination rule.

$$\dfrac{\bot\downarrow}{C\uparrow}\bot E$$

All the proofs we have seen so far in these notes are normal: we can easily annotate them with arrows using only the rules above. The following is an

example of a proof which is not normal.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A\ true}\,u \quad \dfrac{}{\neg A\ true}\,w}{A\ \wedge \neg A\ true}\,\wedge I}{A\ true}\,\wedge E_L \quad \dfrac{}{\neg A\ true}\,w}{\bot\ true}\,\supset E}{\neg A \supset \bot\ true}\,\supset I^w}{A \supset \neg A \supset \bot\ true}\,\supset I^u$$

If we follow the process of annotation, we fail at only one place as indicated below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A\downarrow}\,u \quad \dfrac{}{\neg A\downarrow}\,w}{A\ \wedge \neg A\ ?}\,\wedge I}{A\downarrow}\,\wedge E_L}{A\uparrow}\,\downarrow\uparrow \quad \dfrac{}{\neg A\downarrow}\,w}{\bot\downarrow}\,\supset E}{\bot\uparrow}\,\downarrow\uparrow}{\neg A \supset \bot\uparrow}\,\supset I^w}{A \supset \neg A \supset \bot\uparrow}\,\supset I^u$$

The situation that prevents this deduction from being normal is that we introduce a connective (in this case, $A \wedge \neg A$) and then immediately eliminate it. This seems like a detour—why do it at all? In fact, we can just replace this little inference with the hypothesis $A\downarrow$ and obtain a deduction which is now normal.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A\downarrow}\,u}{A\uparrow}\,\downarrow\uparrow \quad \dfrac{}{\neg A\downarrow}\,w}{\bot\downarrow}\,\supset E}{\bot\uparrow}\,\downarrow\uparrow}{\neg A \supset \bot\uparrow}\,\supset I^w}{A \supset \neg A \supset \bot\uparrow}\,\supset I^u$$

It turns out that the only reason a deduction may not be normal is an introduction followed by an elimination, and that we can always simplify such a derivation to (eventually) obtain a normal one. This process of simplification

is directly connected to computation in a programming language. We only need to fix a particular simplification strategy. Under this interpretation, a proof corresponds to a program, simplification of the kind above corresponds to computation, and a normal proof corresponds to a value. It is precisely this correspondence which is the central topic of the next chapter.

We close this chapter with our first easy meta-theorem, that is, a theorem *about* a logical system rather than within it. We show that if a the proposition A has a normal proof then it must be true. In order to verify this, we also need the auxiliary property that if A has a neutral proof, it is true.

Theorem 2.1 (Soundness of Normal Proofs) *For natural deduction with logical constants $\wedge$, $\supset$, $\vee$, $\top$ and $\bot$ we have:*

1. *If $A \uparrow$ then A true, and*
2. *if $A \downarrow$ then A true.*

Proof: We replace every judgment $B \uparrow$ and $B \downarrow$ in the deduction of $A \uparrow$ or $A \downarrow$ by B *true* and B *true*. This leads to correct derivation that A *true* with one exception: the rule

$$\frac{B \downarrow}{B \uparrow} \downarrow\uparrow$$

turns into

$$\frac{B \ \mathit{true}}{B \ \mathit{true}}$$

We can simply delete this "inference" since premise and conclusion are identical. □

2.10 Exercises

Exercise 2.1 *Show the derivations for the rules $\equiv I$, $\equiv E_L$ and $\equiv E_R$ under the definition of $A \equiv B$ as $(A \supset B) \wedge (B \supset A)$.*

Chapter 3

Proofs as Programs

In this chapter we investigate a computational interpretation of constructive proofs and relate it to functional programming. On the propositional fragment of logic this is referred to as the Curry-Howard isomorphism [How80]. From the very outset of the development of constructive logic and mathematics, a central idea has been that proofs ought to represent constructions. The Curry-Howard isomorphism is only a particularly poignant and beautiful realization of this idea. In a highly influential subsequent paper, Martin-Löf [ML80] developed it further into a more expressive calculus called *type theory*.

3.1 Propositions as Types

In order to illustrate the relationship between proofs and programs we introduce a new judgment:

$M : A$ $\quad$ M is a proof term for proposition A

We presuppose that A is a proposition when we write this judgment. We will also interpret $M : A$ as "*M is a program of type A*". These dual interpretations of the same judgment is the core of the Curry-Howard isomorphism. We either think of M as a term that represents the proof of A *true*, or we think of A as the type of the program M. As we discuss each connective, we give both readings of the rules to emphasize the analogy.

We intend that if $M : A$ then A *true*. Conversely, if A *true* then $M : A$. But we want something more: every deduction of $M : A$ should correspond to a deduction of A *true* with an identical structure and vice versa. In other words we annotate the inference rules of natural deduction with proof terms. The property above should then be obvious.

Conjunction. Constructively, we think of a proof of $A \wedge B$ *true* as a pair of proofs: one for A *true* and one for B *true*.

$$\frac{M : A \qquad N : B}{\langle M, N\rangle : A \wedge B}\,\wedge I$$

The elimination rules correspond to the projections from a pair to its first and second elements.

$$\frac{M : A \wedge B}{\mathbf{fst}\, M : A}\wedge E_L \qquad \frac{M : A \wedge B}{\mathbf{snd}\, M : B}\wedge E_R$$

Hence conjunction $A \wedge B$ corresponds to the product type $A \times B$.

Truth. Constructively, we think of a proof of $\top$ *true* as a unit element that carries now information.

$$\frac{}{\langle\rangle : \top}\top I$$

Hence $\top$ corresponds to the unit type **1** with one element. There is no elimination rule and hence no further proof term constructs for truth.

Implication. Constructively, we think of a proof of $A \supset B$ *true* as a function which transforms a proof of A *true* into a proof of B *true*.

In mathematics and many programming languages, we define a function f of a variable x by writing $f(x) = \ldots$ where the right-hand side "..." depends on x. For example, we might write $f(x) = x^2 + x - 1$. In functional programming, we can instead write $f = \lambda x.\ x^2 + x - 1$, that is, we explicitly form a functional object by λ-*abstraction* of a variable (x, in the example).

We now use the notation of λ-abstraction to annotate the rule of implication introduction with proof terms. In the official syntax, we label the abstraction with a proposition (writing $\lambda u{:}A$) in order to specify the domain of a function unambiguously. In practice we will often omit the label to make expressions shorter—usually (but not always!) it can be determined from the context.

$$\begin{array}{c}\overline{u : A}\,u\\ \vdots\\ M : B\end{array}\supset I^u$$

The hypothesis label u acts as a variable, and any use of the hypothesis labeled u in the proof of B corresponds to an occurrence of u in M.

As a concrete example, consider the (trivial) proof of $A \supset A$ *true*:

$$\frac{\overline{A\ \mathit{true}}\,u}{A \supset A\ \mathit{true}}\supset I^u$$

If we annotate the deduction with proof terms, we obtain

$$\frac{\overline{u : A}\,u}{(\lambda u{:}A.\ u) : A \supset A}\supset I^u$$

So our proof corresponds to the identity function id at type A which simply returns its argument. It can be defined with $\mathrm{id}(u) = u$ or $\mathrm{id} = (\lambda u{:}A.\ u)$.

The rule for implication elimination corresponds to function application. Following the convention in functional programming, we write $M\,N$ for the application of the function M to argument N, rather than the more verbose $M(N)$.

$$\frac{M : A \supset B \qquad N : A}{M\,N : B}\supset E$$

What is the meaning of $A \supset B$ as a type? From the discussion above it should be clear that it can be interpreted as a function type $A \to B$. The introduction and elimination rules for implication can also be viewed as formation rules for functional abstraction $\lambda u{:}A.\ M$ and application $M\,N$.

Note that we obtain the usual introduction and elimination rules for implication if we erase the proof terms. This will continue to be true for all rules in the remainder of this section and is immediate evidence for the soundness of the proof term calculus, that is, if $M : A$ then A *true*.

As a second example we consider a proof of $(A \wedge B) \supset (B \wedge A)$ *true*.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B\ \mathit{true}}\,u}{B\ \mathit{true}}\wedge E_R \quad \dfrac{\dfrac{}{A \wedge B\ \mathit{true}}\,u}{A\ \mathit{true}}\wedge E_L}{B \wedge A\ \mathit{true}}\wedge I}{(A \wedge B) \supset (B \wedge A)\ \mathit{true}}\supset I^u$$

When we annotate this derivation with proof terms, we obtain a function which takes a pair $\langle M, N\rangle$ and returns the reverse pair $\langle N, M\rangle$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{u : A \wedge B}\,u}{\mathbf{snd}\,u : B}\wedge E_R \quad \dfrac{\dfrac{}{u : A \wedge B}\,u}{\mathbf{fst}\,u : A}\wedge E_L}{\langle \mathbf{snd}\,u, \mathbf{fst}\,u\rangle : B \wedge A}\wedge I}{(\lambda u.\ \langle \mathbf{snd}\,u, \mathbf{fst}\,u\rangle) : (A \wedge B) \supset (B \wedge A)}\supset I^u$$

Disjunction. Constructively, we think of a proof of $A \vee B$ *true* as either a proof of A *true* or B *true*. Disjunction therefore corresponds to a disjoint sum type $A + B$, and the two introduction rules correspond to the left and right injection into a sum type.

$$\frac{M : A}{\mathbf{inl}^B\,M : A \vee B}\vee I_L \qquad \frac{N : B}{\mathbf{inr}^A\,N : A \vee B}\vee I_R$$

In the official syntax, we have annotated the injections **inl** and **inr** with propositions B and A, again so that a (valid) proof term has an unambiguous type. In

writing actual programs we usually omit this annotation. The elimination rule corresponds to a case construct which discriminates between a left and right injection into a sum types.

$$\frac{\begin{array}{cccc} & \overline{}\,u & & \overline{}\,w \\ & u : A & & w : B \\ & \vdots & & \vdots \\ M : A \vee B & N : C & & O : C \end{array}}{\mathbf{case}\, M\, \mathbf{of}\, \mathbf{inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O : C}\, \vee E^{u,w}$$

Recall that the hypothesis labeled u is available only in the proof of the second premise and the hypothesis labeled w only in the proof of the third premise. This means that the scope of the variable u is N, while the scope of the variable w is O.

Falsehood. There is no introduction rule for falsehood ($\bot$). We can therefore view it as the empty type $\mathbf{0}$. The corresponding elimination rule allows a term of $\bot$ to stand for an expression of any type when wrapped with **abort**. However, there is no computation rule for it, which means during computation of a valid program we will never try to evaluate a term of the form $\mathbf{abort}\, M$.

$$\frac{M : \bot}{\mathbf{abort}^C\, M : C}\, \bot E$$

As before, the annotation C which disambiguates the type of $\mathbf{abort}\, M$ will often be omitted.

This completes our assignment of proof terms to the logical inference rules. Now we can interpret the interaction laws we introduced early as programming exercises. Consider the left-to-right direction of (L11)

(L11a) $\quad (A \supset (B \wedge C)) \supset (A \supset B) \wedge (A \supset C)$ *true*

Interpreted constructively, this assignment can be read as:

> Write a function which, when given a function from A to pairs of type $B \wedge C$, returns two functions: one which maps A to B and one which maps A to C.

This is satisfied by the following function:

$$\lambda u.\ \langle (\lambda w.\ \mathbf{fst}\, (u\, w)), (\lambda v.\ \mathbf{snd}\, (u\, v)) \rangle$$

The following deduction provides the evidence:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{u : A \supset (B \wedge C)}\,u \quad \dfrac{}{w : A}\,w}{u\,w : B \wedge C}\supset E}{\mathbf{fst}\,(u\,w) : B}\wedge E_L}{\lambda w.\ \mathbf{fst}\,(u\,w) : A \supset B}\supset I^w \quad \dfrac{\dfrac{\dfrac{\dfrac{}{u : A \supset (B \wedge C)}\,u \quad \dfrac{}{v : A}\,v}{u\,v : B \wedge C}\supset E}{\mathbf{snd}\,(u\,v) : C}\wedge E_R}{\lambda v.\ \mathbf{snd}\,(u\,v) : A \supset C}\supset I^v}{\langle(\lambda w.\ \mathbf{fst}\,(u\,w)), (\lambda v.\ \mathbf{snd}\,(u\,v))\rangle : (A \supset B) \wedge (A \supset C)}\wedge I}{\lambda u.\ \langle(\lambda w.\ \mathbf{fst}\,(u\,w)), (\lambda v.\ \mathbf{snd}\,(u\,v))\rangle : (A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))}\supset I^u$$

Programs in constructive propositional logic are somewhat uninteresting in that they do not manipulate basic data types such as natural numbers, integers, lists, trees, etc. We introduce such data types in Section 3.5, following the same method we have used in the development of logic.

To close this section we recall the guiding principles behind the assignment of proof terms to deductions.

1. For every deduction of A *true* there is a proof term M and deduction of $M : A$.

2. For every deduction of $M : A$ there is a deduction of A *true*

3. The correspondence between proof terms M and deductions of A *true* is a bijection.

We will prove these in Section 3.4.

3.2 Reduction

In the preceding section, we have introduced the assignment of proof terms to natural deductions. If proofs are programs then we need to explain how proofs are to be executed, and which results may be returned by a computation.

We explain the operational interpretation of proofs in two steps. In the first step we introduce a judgment of *reduction* $M \Longrightarrow M'$, read "M *reduces to* M'". A computation then proceeds by a sequence of reductions $M \Longrightarrow M_1 \Longrightarrow M_2 \ldots$, according to a fixed strategy, until we reach a value which is the result of the computation. In this section we cover reduction; we return to reduction strategies in Section **??**.

As in the development of propositional logic, we discuss each of the connectives separately, taking care to make sure the explanations are independent. This means we can consider various sublanguages and we can later extend our logic or programming language without invalidating the results from this section. Furthermore, it greatly simplifies the analysis of properties of the reduction rules.

In general, we think of the proof terms corresponding to the introduction rules as the *constructors* and the proof terms corresponding to the elimination rules as the *destructors*.

Conjunction. The constructor forms a pair, while the destructors are the left and right projections. The reduction rules prescribe the actions of the projections.

$$\begin{array}{rcl} \mathbf{fst}\,\langle M, N\rangle & \Longrightarrow & M \\ \mathbf{snd}\,\langle M, N\rangle & \Longrightarrow & N \end{array}$$

Truth. The constructor just forms the unit element, $\langle\,\rangle$. Since there is no destructor, there is no reduction rule.

Implication. The constructor forms a function by λ-abstraction, while the destructor applies the function to an argument. In general, the application of a function to an argument is computed by *substitution*. As a simple example from mathematics, consider the following equivalent definitions

$$f(x) = x^2 + x - 1 \qquad f = \lambda x.\ x^2 + x - 1$$

and the computation

$$f(3) = (\lambda x.\ x^2 + x - 1)(3) = [3/x](x^2 + x - 1) = 3^2 + 3 - 1 = 11$$

In the second step, we substitute 3 for occurrences of x in $x^2 + x - 1$, the *body of the λ-expression*. We write $[3/x](x^2 + x - 1) = 3^2 + 3 - 1$.

In general, the notation for the substitution of N for occurrences of u in M is $[N/u]M$. We therefore write the reduction rule as

$$(\lambda u{:}A.\ M)\,N \quad\Longrightarrow\quad [N/u]M$$

We have to be somewhat careful so that substitution behaves correctly. In particular, no variable in N should be bound in M in order to avoid conflict. We can always achieve this by renaming bound variables—an operation which clearly does not change the meaning of a proof term.

Disjunction. The constructors inject into a sum types; the destructor distinguishes cases. We need to use substitution again.

$$\begin{array}{rcl} \mathbf{case}\,\mathbf{inl}^B\,M\,\mathbf{of}\,\mathbf{inl}\,u \Rightarrow N \mid \mathbf{inr}\,w \Rightarrow O & \Longrightarrow & [M/u]N \\ \mathbf{case}\,\mathbf{inr}^A\,M\,\mathbf{of}\,\mathbf{inl}\,u \Rightarrow N \mid \mathbf{inr}\,w \Rightarrow O & \Longrightarrow & [M/w]O \end{array}$$

Falsehood. Since there is no constructor for the empty type there is no reduction rule for falsehood.

This concludes the definition of the reduction judgment. In the next section we will prove some of its properties.

As an example we consider a simple program for the composition of two functions. It takes a pair of two functions, one from A to B and one from B to C and returns their composition which maps A directly to C.

$$\mathsf{comp} \quad : \quad ((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$$

We transform the following implicit definition into our notation step-by-step:

$$\begin{array}{rcl} \mathsf{comp}\,\langle f, g\rangle\,(w) & = & g(f(w)) \\ \mathsf{comp}\,\langle f, g\rangle & = & \lambda w.\ g(f(w)) \\ \mathsf{comp}\,u & = & \lambda w.\ (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)(w)) \\ \mathsf{comp} & = & \lambda u.\ \lambda w.\ (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w) \end{array}$$

The final definition represents a correct proof term, as witnessed by the following deduction.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{u : (A \supset B) \wedge (B \supset C)}\,u}{\mathbf{snd}\,u : B \supset C}\wedge E_R \quad \dfrac{\dfrac{\dfrac{}{u : (A \supset B) \wedge (B \supset C)}\,u}{\mathbf{fst}\,u : A \supset B}\wedge E_L \quad \dfrac{}{w : A}\,w}{(\mathbf{fst}\,u)\,w : B}\supset E}{(\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w) : C}\supset E}{\lambda w.\ (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w) : A \supset C}\supset I^w}{(\lambda u.\ \lambda w.\ (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w)) : ((A \supset B) \wedge (B \supset C)) \supset (A \supset C)}\supset I^u$$

We now verify that the composition of two identity functions reduces again to the identity function. First, we verify the typing of this application.

$$(\lambda u.\ \lambda w.\ (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w))\,\langle(\lambda x.\ x), (\lambda y.\ y)\rangle \quad : \quad A \supset A$$

Now we show a possible sequence of reduction steps. This is by no means uniquely determined.

$$\begin{array}{rl} & (\lambda u.\ \lambda w.\ (\mathbf{snd}\,u)\,((\mathbf{fst}\,u)\,w))\,\langle(\lambda x.\ x), (\lambda y.\ y)\rangle \\ \Longrightarrow & \lambda w.\ (\mathbf{snd}\,\langle(\lambda x.\ x), (\lambda y.\ y)\rangle)\,((\mathbf{fst}\,\langle(\lambda x.\ x), (\lambda y.\ y)\rangle)\,w) \\ \Longrightarrow & \lambda w.\ (\lambda y.\ y)\,((\mathbf{fst}\,\langle(\lambda x.\ x), (\lambda y.\ y)\rangle)\,w) \\ \Longrightarrow & \lambda w.\ (\lambda y.\ y)\,((\lambda x.\ x)\,w) \\ \Longrightarrow & \lambda w.\ (\lambda y.\ y)\,w \\ \Longrightarrow & \lambda w.\ w \end{array}$$

We see that we may need to apply reduction steps to subterms in order to reduce a proof term to a form in which it can no longer be reduced. We postpone a more detailed discussion of this until we discuss the operational semantics in full.

3.3 Summary of Proof Terms

Judgments.

$M : A$	M is a proof term for proposition A
$M \Longrightarrow M'$	M reduces to M'

Proof Term Assignment.

Constructors	Destructors
$\dfrac{M : A \qquad N : B}{\langle M, N\rangle : A \wedge B}\,\wedge I$	$\dfrac{M : A \wedge B}{\mathbf{fst}\, M : A}\,\wedge E_L$ $\dfrac{M : A \wedge B}{\mathbf{snd}\, M : B}\,\wedge E_R$
$\dfrac{}{\langle\,\rangle : \top}\,\top I$	no destructor for $\top$
$\dfrac{\begin{array}{c}\overline{u : A}\,u \\ \vdots \\ M : B\end{array}}{\lambda u{:}A.\ M : A \supset B}\,\supset I^u$	$\dfrac{M : A \supset B \qquad N : A}{M\ N : B}\,\supset E$
$\dfrac{M : A}{\mathbf{inl}^B\, M : A \vee B}\,\vee I_L$ $\dfrac{N : B}{\mathbf{inr}^A\, N : A \vee B}\,\vee I_R$	$\dfrac{M : A \vee B \qquad \begin{array}{c}\overline{u : A}\,u \\ \vdots \\ N : C\end{array} \qquad \begin{array}{c}\overline{w : B}\,w \\ \vdots \\ O : C\end{array}}{\mathbf{case}\, M\, \mathbf{of}\, \mathbf{inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O : C}\,\vee E^{u,w}$
no constructor for $\bot$	$\dfrac{M : \bot}{\mathbf{abort}^C\, M : C}\,\bot E$

Reductions.

$$\begin{array}{rcl}
\mathbf{fst}\,\langle M, N\rangle & \Longrightarrow & M \\
\mathbf{snd}\,\langle M, N\rangle & \Longrightarrow & N \\
\text{no reduction for } \langle\,\rangle & & \\
(\lambda u{:}A.\ M)\,N & \Longrightarrow & [N/u]M \\
\mathbf{case}\,\mathbf{inl}^B\,M\,\mathbf{of}\,\mathbf{inl}\,u \Rightarrow N \mid \mathbf{inr}\,w \Rightarrow O & \Longrightarrow & [M/u]N \\
\mathbf{case}\,\mathbf{inr}^A\,M\,\mathbf{of}\,\mathbf{inl}\,u \Rightarrow N \mid \mathbf{inr}\,w \Rightarrow O & \Longrightarrow & [M/w]O \\
\text{no reduction for } \mathbf{abort} & &
\end{array}$$

Concrete Syntax. The concrete syntax for proof terms used in the mechanical proof checker has some minor differences to the form we presented above.

u	`u`	Variable
$\langle M, N\rangle$	`(M,N)`	Pair
$\mathbf{fst}\,M$	`fst M`	First projection
$\mathbf{snd}\,M$	`snd M`	Second projection
$\langle\,\rangle$	`()`	Unit element
$\lambda u{:}A.\ M$	`fn u => M`	Abstraction
$M\,N$	`M N`	Application
$\mathbf{inl}^B\,M$	`inl M`	Left injection
$\mathbf{inr}^A\,N$	`inr N`	Right injection
$\mathbf{case}\,M$	`case M`	Case analysis
$\quad\mathbf{of}\,\mathbf{inl}\,u \Rightarrow N$	`  of inl u => N`	
$\quad\mid \mathbf{inr}\,w \Rightarrow O$	`   | inr w => O`	
	`end`	
$\mathbf{abort}^C\,M$	`abort M`	Abort

Pairs and unit element are delimited by parentheses '`(`' and '`)`' instead of angle brackets $\langle$ and $\rangle$. The `case` constructs requires an `end` token to mark the end of the a sequence of cases.

Type annotations are generally omitted, but a whole term can explicitly be given a type. The proof checker (which here is also a type checker) infers the missing information. Occasionally, an explicit type ascription `M : A` is necessary as a hint to the type checker.

For rules of operator precedence, the reader is refered to the on-line documentation of the proof checking software available with the course material. Generally, parentheses can be used to disambiguate or override the standard rules.

As an example, we show the proof term implementing function composition.

```
term comp : (A => B) & (B => C) => (A => C) =
fn u => fn x => (snd u) ((fst u) x);
```

We also allow annotated deductions, where each line is annotated with a proof term. This is a direct transcription of deduction for judgments of the form $M : A$. As an example, we show the proof that $A \vee B \supset B \vee A$, first in the pure form.

```
proof orcomm : A | B => B | A =
begin
[ A | B;
  [ A;
    B | A];
  [ B;
    B | A];
  B | A ];
A | B => B | A
end;
```

Now we systematically annotate each line and obtain

```
annotated proof orcomm : A | B => B | A =
begin
[ u : A | B;
  [ v : A;
    inr v : B | A];
  [ w : B;
    inl w : B | A];
  case u
    of inl v => inr v
     | inr w => inl w
   end : B | A ];
fn u => case u
          of inl v => inr v
           | inr w => inl w
         end : A | B => B | A
end;
```

3.4 Properties of Proof Terms

In this section we analyze and verify various properties of proof terms. Rather than concentrate on reasoning within the logical calculi we introduced, we now want to reason about them. The techniques are very similar—they echo the ones we have introduced so far in natural deduction. This should not be surprising. After all, natural deduction was introduced to model mathematical reasoning, and we now engage in some mathematical reasoning about proof terms, propositions, and deductions. We refer to this as *meta-logical reasoning.*

First, we need some more formal definitions for certain operations on proof terms, to be used in our meta-logical analysis. One rather intuitive property of is that variable names should not matter. For example, the identity function at type A can be written as $\lambda u{:}A.\ u$ or $\lambda w{:}A.\ w$ or $\lambda u'{:}A.\ u'$, etc. They all denote the same function and the same proof. We therefore identify terms which differ only in the names of variables (here called u) bound in $\lambda u{:}A.\ M$, $\mathbf{inl}\, u \Rightarrow M$ or $\mathbf{inr}\, u \Rightarrow O$. But there are pitfalls with this convention: variables have to be renamed *consistently* so that every variable refers to the same binder before and after the renaming. For example (omitting type labels for brevity):

$$\begin{array}{rcl} \lambda u.\ u & = & \lambda w.\ w \\ \lambda u.\ \lambda w.\ u & = & \lambda u'.\ \lambda w.\ u' \\ \lambda u.\ \lambda w.\ u & \neq & \lambda u.\ \lambda w.\ w \\ \lambda u.\ \lambda w.\ u & \neq & \lambda w.\ \lambda w.\ w \\ \lambda u.\ \lambda w.\ w & = & \lambda w.\ \lambda w.\ w \end{array}$$

The convention to identify terms which differ only in the naming of their bound variables goes back to the first papers on the λ-calculus by Church and Rosser [CR36], is called the "*variable name convention*" and is pervasive in the literature on programming languages and λ-calculi. The term λ-calculus typically refers to a pure calculus of functions formed with λ-abstraction. Our proof term calculus is called a *typed λ-calculus* because of the presence of propositions (which an be viewed as types).

Following the variable name convention, we may silently rename when convenient. A particular instance where this is helpful is substitution. Consider

$$[u/w](\lambda u.\ w\, u)$$

that is, we substitute u for w in $\lambda u.\ w\, u$. Note that u is a variable visible on the outside, but also bound by λu. By the variable name convention we have

$$[u/w](\lambda u.\ w\, u) = [u/w](\lambda u'.\ w\, u') = \lambda u'.\ u\, u'$$

which is correct. But we cannot substitute without renaming, since

$$[u/w](\lambda u.\ w\, u) \neq \lambda u.\ u\, u$$

In fact, the right hand side below is invalid, while the left-hand side makes perfect sense. We say that u is *captured* by the binder λu. If we assume a hypothesis $u{:}\top \supset A$ then

$$[u/w](\lambda u{:}\top.\ w\, u) : A$$

but

$$\lambda u{:}\top.\ u\, u$$

is not well-typed since the first occurrence of u would have to be of type $\top \supset A$ but instead has type $\top$.

So when we carry out substitution $[M/u]N$ we need to make sure that no variable in M is *captured* by a binder in N, leading to an incorrect result.

Fortunately we can always achieve that by renaming some bound variables in N if necessary. We could now write down a formal definition of substitution, based on the cases for the term we are substituting into. However, we hope that the notion is sufficiently clear that this is not necessary.

Instead we revisit the substitution principle for hypothetical judgments. It states that if we have a hypothetical proof of C *true* from A *true* and we have a proof of A *true*, we can substitute the proof of A *true* for uses of the hypothesis A *true* and obtain a (non-hypothetical) proof of A *true*. In order to state this more precisely in the presence of several hypotheses, we recall that

$$\begin{array}{c} A_1\ \mathit{true} \ldots A_n\ \mathit{true} \\ \vdots \\ C\ \mathit{true} \end{array}$$

can be written as

$$\underbrace{A_1\ \mathit{true}, \ldots, A_n\ \mathit{true}}_{\Delta} \vdash C\ \mathit{true}$$

Generally we abbreviate several hypotheses by Δ. We then have the following properties, evident from the very definition of hypothetical judgments and hypothetical proofs

Weakening: If $\Delta \vdash C\ \mathit{true}$ then $\Delta, \Delta' \vdash C\ \mathit{true}$.

Substitution: If $\Delta, A\ \mathit{true}, \Delta' \vdash C\ \mathit{true}$ and $\Delta \vdash A\ \mathit{true}$ then $\Delta, \Delta' \vdash C\ \mathit{true}$.

As indicated above, weakening is realized by adjoining unused hypotheses, substitutions is realized by substitution of proofs for hypotheses.

For the proof term judgment, $M : A$, we use the same notation and write

$$\begin{array}{c} u_1{:}A_1\ \ldots\ u_n{:}A_n \\ \vdots \\ N : C \end{array}$$

as

$$\underbrace{u_1{:}A_1, \ldots, u_n{:}A_n}_{\Gamma} \vdash N : C$$

We use Γ to refer to collections of hypotheses $u_i{:}A_i$. In the deduction of $N : C$, each u_i stands for an unknown proof term for A_i, simply assumed to exist. If we actually find a proof $M_i{:}A_i$ we can eliminate this assumption, again by substitution. However, this time, the substitution has to perform two operations: we have to substitute M_i for u_i (the unknown proof term variable), and the deduction of $M_i : A_i$ for uses of the hypothesis $u_i{:}A_i$. More precisely, we have the following two properties:

Weakening: If $\Gamma \vdash N : C$ then $\Gamma, \Gamma' \vdash N : C$.

Substitution: If $\Gamma, u{:}A, \Gamma' \vdash N : C$ and $\Gamma \vdash M : A$ then $\Gamma, \Gamma' \vdash [M/u]N : C$.

Now we are in a position to state and prove our second meta-theorem, that is, a theorem about the logic under consideration. The theorem is called *subject reduction* because is concerns the *subject* M of the judgment $M : A$. It states that reduction preserves the type of an object. We make the hypotheses explicit as we have done in the explanations above.

Theorem 3.1 (Subject Reduction)
If $\Gamma \vdash M : A$ *and* $M \Longrightarrow M'$ *then* $\Gamma \vdash M' : A$.

Proof: We consider each case in the definition of $M \Longrightarrow M'$ in turn and show that the property holds. This is simply an instance of *proof by cases*.

Case: $\mathbf{fst}\,\langle M_1, M_2\rangle \Longrightarrow M_1$. By assumption we also know that

$$\Gamma \vdash \mathbf{fst}\,\langle M_1, M_2\rangle : A.$$

We need to show that $\Gamma \vdash M_1 : A$.

Now we inspect all inference rules for the judgment $M : A$ and we see that there is only one way how the judgment above could have been inferred: by $\wedge E_L$ from

$$\Gamma \vdash \langle M_1, M_2\rangle : A \wedge A_2$$

for some A_2. This step is called *inversion*, since we infer the premises from the conclusion of the rule. But we have to be extremely careful to inspect all possibilities for derivations so that we do not forget any cases.

Next, we apply inversion again: the judgment above could only have been inferred by $\wedge I$ from the two premises

$$\Gamma \vdash M_1 : A$$

and

$$\Gamma \vdash M_2 : A_2$$

But the first of these is what we had to prove in this case and we are done.

Case: $\mathbf{snd}\,\langle M_1, M_2\rangle \Longrightarrow M_2$. This is symmetric to the previous case. We write it an abbreviated form.

$\Gamma \vdash \mathbf{snd}\,\langle M_1, M_2\rangle : A$	Assumption
$\Gamma \vdash \langle M_1, M_2\rangle : A_1 \wedge A$ for some A_1	By inversion
$\Gamma \vdash M_1 : A_1$ and	
$\Gamma \vdash M_2 : A$	By inversion

Here the last judgment is what we were trying to prove.

Case: There is no reduction for $\top$ since there is no elimination rule and hence no destructor.

Case: $(\lambda u{:}A_1.\ M_2)\, M_1 \Longrightarrow [M_1/u]M_2$. By assumption we also know that

$$\Gamma \vdash (\lambda u{:}A_1.\ M_2)\, M_1 : A.$$

We need to show that $\Gamma \vdash [M_1/u]M_2 : A$.

Since there is only one inference rule for function application, namely implication elimination ($\supset E$), we can apply inversion and find that

$$\Gamma \vdash (\lambda u{:}A_1.\ M_2) : A_1' \supset A$$

and

$$\Gamma \vdash M_1 : A_1'$$

for some A_1'. Now we repeat inversion on the first of these and conclude that

$$\Gamma, u{:}A_1 \vdash M_2 : A$$

and, moreover, that $A_1 = A_1'$. Hence

$$\Gamma \vdash M_1 : A_1$$

Now we can apply the substitution property to these to judgments to conclude

$$\Gamma \vdash [M_1/u]M_2 : A$$

which is what we needed to show.

Case: $(\mathbf{case}\,\mathbf{inl}^C\, M_1\ \mathbf{of}\ \mathbf{inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O) \Longrightarrow [M_1/u]N$. By assumption we also know that

$$\Gamma \vdash (\mathbf{case}\,\mathbf{inl}^C\, M_1\ \mathbf{of}\ \mathbf{inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O) : A$$

Again we apply inversion and obtain three judgments

$$\begin{array}{l}\Gamma \vdash \mathbf{inl}^C\, M_1 : B' \vee C' \\ \Gamma, u{:}B' \vdash N : A \\ \Gamma, w{:}C' \vdash O : A\end{array}$$

for some B' and C'.

Again by inversion on the first of these, we find

$$\Gamma \vdash M_1 : B'$$

and also $C' = C$. Hence we can apply the substitution property to get

$$\Gamma \vdash [M_1/u]N : A$$

which is what we needed to show.

Case: $(\mathbf{case}\,\mathbf{inr}^B\, M_1\ \mathbf{of}\ \mathbf{inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O) \Longrightarrow [M_1/u]N$. This is symmetric to the previous case and left as an exercise.

Case: There is no introduction rule for $\bot$ and hence no reduction rule.

□

The important techniques introduced in the proof above are *proof by cases* and *inversion*. In a proof by cases we simply consider all possibilities for why a judgment could be evident and show the property we want to establish in each case. Inversion is very similar: from the shape of the judgment we see it could have been inferred only in one possible way, so we know the premises of this rule must also be evident. We see that these are just two slightly different forms of the same kind of reasoning.

If we look back at our early example computation, we saw that the reduction step does not always take place at the top level, but that the redex may be embedded in the term. In order to allow this, we need to introduce some additional ways to establish that $M \Longrightarrow M'$ when the actual reduction takes place *inside* M. This is accomplished by so-called *congruence rules*.

Conjunction. As usual, conjunction is the simplest.

$$\frac{M \Longrightarrow M'}{\langle M, N\rangle \Longrightarrow \langle M', N\rangle} \qquad \frac{N \Longrightarrow N'}{\langle M, N\rangle \Longrightarrow \langle M, N'\rangle}$$

$$\frac{M \Longrightarrow M'}{\mathbf{fst}\, M \Longrightarrow \mathbf{fst}\, M'} \qquad \frac{M \Longrightarrow M'}{\mathbf{snd}\, M \Longrightarrow \mathbf{snd}\, M'}$$

Note that there is one rule for each subterm for each construct in the language of proof terms, just in case the reduction might take place in that subterm.

Truth. There are no rules for truth, since $\langle\rangle$ has no subterms and therefore permits no reduction inside.

Implication. This is similar to conjunction.

$$\frac{M \Longrightarrow M'}{M\, N \Longrightarrow M'\, N} \qquad \frac{N \Longrightarrow N'}{M\, N \Longrightarrow M\, N'}$$

$$\frac{M \Longrightarrow M'}{(\lambda u{:}A.\ M) \Longrightarrow (\lambda u{:}A.\ M')}$$

Disjunction. This requires no new ideas, just more cases.

$$\frac{M \Longrightarrow M'}{\mathbf{inl}^B\, M \Longrightarrow \mathbf{inl}^B\, M'} \qquad \frac{N \Longrightarrow N'}{\mathbf{inr}^A\, N \Longrightarrow \mathbf{inr}^A\, N'}$$

$$\frac{M \Longrightarrow M'}{(\mathbf{case}\, M\ \mathbf{of\ inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O) \Longrightarrow (\mathbf{case}\, M'\ \mathbf{of\ inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O)}$$

$$\frac{N \Longrightarrow N'}{(\mathbf{case}\, M\ \mathbf{of\ inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O) \Longrightarrow (\mathbf{case}\, M\ \mathbf{of\ inl}\, u \Rightarrow N' \mid \mathbf{inr}\, w \Rightarrow O)}$$

$$\frac{O \Longrightarrow O'}{(\mathbf{case}\, M\ \mathbf{of\ inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O) \Longrightarrow (\mathbf{case}\, M\ \mathbf{of\ inl}\, u \Rightarrow N \mid \mathbf{inr}\, w \Rightarrow O')}$$

Falsehood. Finally, there *is* a congruence rule for falsehood, since the proof term constructor has a subterm.

$$\frac{M \Longrightarrow M'}{\mathbf{abort}^C\, M \Longrightarrow \mathbf{abort}^C\, M'}$$

We now extend the theorem to the general case of reduction on subterms. A proof by cases is now no longer sufficient, since the congruence rules have premises, for which we would have to analyze cases again, and again, etc.

Instead we use a technique called *structural induction* on proofs. In structural induction we analyse each inference rule, assuming the desired property for the premises, proving that they hold for the conclusion. If that is the case for all inference rules, the conclusion of each deduction must have the property.

Theorem 3.2 (Subterm Subject Reduction)
If $\Gamma \vdash M : A$ *and* $M \Longrightarrow M'$ *then* $\Gamma \vdash M' : A$ *where* $M \Longrightarrow M'$ *refers to the congruent interpretation of reduction.*

Proof: The cases where the reduction takes place at the top level of the term M, the cases in the proof of Theorem 3.1 still apply. The new cases are all very similar, and we only show one.

Case: The derivation of $M \Longrightarrow M'$ has the form

$$\frac{M_1 \Longrightarrow M_1'}{\langle M_1, M_2\rangle \Longrightarrow \langle M_1', M_2\rangle}$$

We also know that $\Gamma \vdash \langle M_1, M_2\rangle : A$. We need to show that

$$\Gamma \vdash \langle M_1', M_2\rangle : A$$

By inversion,

$$\Gamma \vdash M_1 : A_1$$

and

$$\Gamma \vdash M_2 : A_2$$

and $A = A_1 \wedge A_2$.

Since we are proving the theorem by structural induction and we have a deduction of $\Gamma \vdash M_1 : A_1$ we can now apply the induction hypothesis to $M_1 \Longrightarrow M_1'$. This yields

$$\Gamma \vdash M_1' : A_1$$

and we can construct the deduction

$$\frac{\Gamma \vdash M_1' : A_1 \qquad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \langle M_1', M_2 \rangle : A_1 \wedge A_2} \wedge I$$

which is what we needed to show since $A = A_1 \wedge A_2$.

Cases: All other cases are similar and left as an exercise.

□

The importance of the technique of structural induction cannot be overemphasized in this domain. We will see it time and again, so the reader should make sure the understand each step in the proof above.

3.5 Primitive Recursion

In the preceding sections we have developed an interpretation of propositions as types. This interpretation yields function types (from implication), product types (from conjunction), unit type (from truth), sum types (from disjunction) and the empty type (from falsehood). What is missing for a reasonable programming language are basic data types such as natural numbers, integers, lists, trees, etc. There are several approaches to incorporating such types into our framework. One is to add a general definition mechanism for *recursive types* or *inductive types*. We return to this option later. Another one is to specify each type in a way which is analogous to the definitions of the logical connectives via introduction and elimination rules. This is the option we pursue in this section. A third way is to use the constructs we already have to define data. This was Church's original approach culminating in the so-called *Church numerals*. We will not discuss this idea in these notes.

After spending some time to illustrate the interpretation of propositions as types, we now introduce types as a first-class notion. This is not strictly necessary, but it avoids the question what, for example, **nat** (the type of natural numbers) means as a proposition. Accordingly, we have a new judgment $\tau\ type$ meaning "τ *is a type*". To understand the meaning of a type means to understand what elements it has. We therefore need a second judgment $t \in \tau$ (read:

"*t is an element of type* τ") that is defined by introduction rules with their corresponding elimination rules. As in the case of logical connectives, computation arises from the meeting of elimination and introduction rules. Needless to say, we will continue to use our mechanisms of hypothetical judgments.

Before introducing any actual data types, we look ahead at their use in logic. We will introduce new propositions of the form $\forall x \in \tau.\ A(x)$ (*A is true for every element x of type τ*) and $\exists x \in \tau.\ A(x)$ (*A is true some some element x of type τ*). This will be the step from propositional logic to first-order logic. This logic is called *first-order* because we can quantify (via $\forall$ and $\exists$) only over elements of data types, but not propositions themselves.

We begin our presentation of data types with the natural numbers. The formation rule is trivial: **nat** is a type.

$$\frac{}{\mathbf{nat}\ \mathit{type}}\ \mathbf{nat}F$$

Now we state two of Peano's famous axioms in judgmental form as introduction rules: (1) $\mathbf{0}$ is a natural numbers, and (2) if n is a natural number then its successor, $\mathbf{s}(n)$, is a natural number. We write $\mathbf{s}(n)$ instead of $n+1$, since addition and the number 1 have yet to be defined.

$$\frac{}{\mathbf{0} \in \mathbf{nat}}\ \mathbf{nat}I_0 \qquad\qquad \frac{n \in \mathbf{nat}}{\mathbf{s}(n) \in \mathbf{nat}}\ \mathbf{nat}I_s$$

The elimination rule is a bit more difficult to construct. Assume have a natural number n. Now we cannot directly take its predecessor, for example, because we do not know if n was constructed using $\mathbf{nat}I_0$ or $\mathbf{nat}I_s$. This is similar to the case of disjunction, and our solution is also similar: we distinguish cases. In general, it turns out this is not sufficient, but our first approximation for an elimination rule is

$$\frac{\begin{array}{ccc} & & \dfrac{}{x \in \mathbf{nat}}\,x \\ & & \vdots \\ n \in \mathbf{nat} & t_0 \in \tau & t_s \in \tau \end{array}}{\mathbf{case}\, n\, \mathbf{of}\, \mathbf{0} \Rightarrow t_0 \mid \mathbf{s}(x) \Rightarrow t_s \in \tau}\,x$$

Note that x is introduced in the third premise; its scope is t_s. First, we rewrite this using our more concise notation for hypothetical judgments. For now, Γ contains assumptions of the form $x \in \tau$. Later, we will add logical assumptions of the form $u{:}A$.

$$\frac{\Gamma \vdash n \in \mathbf{nat} \qquad \Gamma \vdash t_0 \in \tau \qquad \Gamma, x \in \mathbf{nat} \vdash t_s \in \tau}{\Gamma \vdash \mathbf{case}\, n\, \mathbf{of}\, \mathbf{0} \Rightarrow t_0 \mid \mathbf{s}(x) \Rightarrow t_s \in \tau}\,x$$

This elimination rule is sound, and under the computational interpretation of terms, type preservation holds. The reductions rules are

$$\begin{array}{rcl} (\mathbf{case}\, \mathbf{0}\, \mathbf{of}\, \mathbf{0} \Rightarrow t_0 \mid \mathbf{s}(x) \Rightarrow t_s) & \Longrightarrow & t_0 \\ (\mathbf{case}\, \mathbf{s}(n)\, \mathbf{of}\, \mathbf{0} \Rightarrow t_0 \mid \mathbf{s}(x) \Rightarrow t_s) & \Longrightarrow & [n/x]t_s \end{array}$$

Clearly, this is the intended reading of the case construct in programs.

In order to use this in writing programs independently of the logic developed earlier, we now introduce function types in a way that is isomorphic to implication.

$$\frac{\tau\ \mathit{type} \qquad \sigma\ \mathit{type}}{\tau \to \sigma\ \mathit{type}} \to F$$

$$\frac{\Gamma, x \in \sigma \vdash t \in \tau}{\Gamma \vdash \lambda x \in \sigma.\, t \in \sigma \to \tau} \to I^x \qquad \frac{\Gamma \vdash s \in \tau \to \sigma \qquad \Gamma \vdash t \in \tau}{\Gamma \vdash s\, t \in \sigma} \to E$$

$$(\lambda x \in \sigma.\, s)\, t \Longrightarrow [t/x]s$$

Now we can write a function for truncated predecessor: the predecessor of $\mathbf{0}$ is defined to be $\mathbf{0}$; otherwise the predecessor of $n+1$ is simply n. We phrase this as a notational definition.

$$\mathit{pred} \quad = \quad \lambda x \in \mathbf{nat}.\, \mathbf{case}\, x\, \mathbf{of}\, \mathbf{0} \Rightarrow \mathbf{0} \mid \mathbf{s}(y) \Rightarrow y$$

Then $\vdash \mathit{pred} \in \mathbf{nat} \to \mathbf{nat}$ and we can formally calculate the predecessor of 2.

$$\begin{array}{rcl} \mathit{pred}(\mathbf{s}(\mathbf{s}(\mathbf{0}))) & = & (\lambda x \in \mathbf{nat}.\, \mathbf{case}\, x\, \mathbf{of}\, \mathbf{0} \Rightarrow \mathbf{0} \mid \mathbf{s}(y) \Rightarrow y)\, (\mathbf{s}(\mathbf{s}(\mathbf{0}))) \\ & \Longrightarrow & \mathbf{case}\, \mathbf{s}(\mathbf{s}(\mathbf{0}))\, \mathbf{of}\, \mathbf{0} \Rightarrow \mathbf{0} \mid \mathbf{s}(y) \Rightarrow y \\ & \Longrightarrow & \mathbf{s}(\mathbf{0}) \end{array}$$

As a next example, we consider a function which doubles its argument. The behavior of the *double* function on an argument can be specified as follows:

$$\begin{array}{rcl} \mathit{double}(\mathbf{0}) & = & \mathbf{0} \\ \mathit{double}(\mathbf{s}(n)) & = & \mathbf{s}(\mathbf{s}(\mathit{double}(n))) \end{array}$$

Unfortunately, there is no way to transcribe this definition into an application of the **case**-construct for natural numbers, since it is *recursive*: the right-hand side contains an occurrence of *double*, the function we are trying to define.

Fortunately, we can generalize the elimination construct for natural numbers to permit this kind of recursion which is called *primitive recursion*. Note that we can define the value of a function on $\mathbf{s}(n)$ only in terms of n and the value of the function on n. We write

$$\frac{\Gamma \vdash t \in \mathbf{nat} \qquad \Gamma \vdash t_0 \in \tau \qquad \Gamma, x \in \mathbf{nat}, f(x) \in \tau \vdash t_s \in \tau}{\Gamma \vdash \mathbf{rec}\, t\, \mathbf{of}\, f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s \in \tau}\, \mathbf{nat} E^{f,x}$$

Here, f may not occur in t_0 and can only occur in the form $f(x)$ in t_s to denote the result of the recursive call. Essentially, $f(x)$ is just the mnemonic name of a new variable for the result of the recursive call. Moreover, x is bound with scope t_s. The reduction rules are now recursive:

$$\begin{array}{rcl} (\mathbf{rec}\, \mathbf{0}\, \mathbf{of}\, f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s) & \Longrightarrow & t_0 \\ (\mathbf{rec}\, \mathbf{s}(n)\, \mathbf{of}\, f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s) & \Longrightarrow & \\ [(\mathbf{rec}\, n\, \mathbf{of}\, f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s)/f(x)]\, [n/x]\, t_s & & \end{array}$$

As an example we revisit the double function and give it as a notational definition.

$$\begin{array}{rcl} \mathit{double} & = & \lambda x \in \mathbf{nat}.\ \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ d(\mathbf{0}) \Rightarrow \mathbf{0} \\ & & \qquad\ \mid d(\mathbf{s}(x')) \Rightarrow \mathbf{s}(\mathbf{s}(d(x'))) \end{array}$$

Now $\mathit{double}\,(\mathbf{s}(\mathbf{0}))$ can be computed as follows

$$\begin{array}{rl} & (\lambda x \in \mathbf{nat}.\ \mathbf{rec}\ x \\ & \qquad \mathbf{of}\ d(\mathbf{0}) \Rightarrow \mathbf{0} \\ & \qquad\ \mid d(\mathbf{s}(x')) \Rightarrow \mathbf{s}(\mathbf{s}(d(x')))) \\ & \ (\mathbf{s}(\mathbf{0})) \\ \Longrightarrow & \mathbf{rec}\ (\mathbf{s}(\mathbf{0})) \\ & \ \mathbf{of}\ d(\mathbf{0}) \Rightarrow \mathbf{0} \\ & \quad \mid d(\mathbf{s}(x')) \Rightarrow \mathbf{s}(\mathbf{s}(d(x')))) \\ \Longrightarrow & \mathbf{s}(\mathbf{s}(\mathbf{rec}\ \mathbf{0} \\ & \qquad \mathbf{of}\ d(\mathbf{0}) \Rightarrow \mathbf{0} \\ & \qquad\ \mid d(\mathbf{s}(x')) \Rightarrow \mathbf{s}(\mathbf{s}(d(x'))))) \\ \Longrightarrow & \mathbf{s}(\mathbf{s}(\mathbf{0})) \end{array}$$

As some other examples, we consider the functions for addition and multiplication. These definitions are by no means uniquely determined. In each case we first give an implicit definition, describing the intended behavior of the function, and then the realization in our language.

$$\begin{array}{rcl} \mathit{plus}\,\mathbf{0}\,y & = & y \\ \mathit{plus}\,(\mathbf{s}(x'))\,y & = & \mathbf{s}(\mathit{plus}\,x'\,y) \end{array}$$

$$\begin{array}{rcl} \mathit{plus} & = & \lambda x \in \mathbf{nat}.\ \lambda y \in \mathbf{nat}.\ \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ p(\mathbf{0}) \Rightarrow y \\ & & \qquad\ \mid p(\mathbf{s}(x')) \Rightarrow \mathbf{s}(p(x')) \end{array}$$

$$\begin{array}{rcl} \mathit{times}\,\mathbf{0}\,y & = & \mathbf{0} \\ \mathit{times}\,(\mathbf{s}(x'))\,y & = & \mathit{plus}\,y\,(\mathit{times}\,x'\,y) \end{array}$$

$$\begin{array}{rcl} \mathit{times} & = & \lambda x \in \mathbf{nat}.\ \lambda y \in \mathbf{nat}.\ \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ t(\mathbf{0}) \Rightarrow \mathbf{0} \\ & & \qquad\ \mid t(\mathbf{s}(x')) \Rightarrow \mathit{plus}\,y\,(t(x')) \end{array}$$

The next example requires pairs in the language. We therefore introduce

pairs which are isomorphic to the proof terms for conjunction from before.

$$\frac{\Gamma \vdash s \in \sigma \qquad \Gamma \vdash t \in \tau}{\Gamma \vdash \langle s, t\rangle \in \sigma \times \tau} \times I$$

$$\frac{\Gamma \vdash t \in \tau \times \sigma}{\Gamma \vdash \mathbf{fst}\, t \in \tau} \times E_L \qquad \frac{\Gamma \vdash t \in \tau \times \sigma}{\Gamma \vdash \mathbf{snd}\, t \in \sigma} \times E_R$$

$$\begin{array}{c} \mathbf{fst}\,\langle t, s\rangle \Longrightarrow t \\ \mathbf{snd}\,\langle t, s\rangle \Longrightarrow s \end{array}$$

Next the function *half*, rounding down if necessary. This is slightly trickier then the examples above, since we would like to count down by *two* as the following specification indicates.

$$\begin{array}{rcl} \mathit{half}\,\mathbf{0} & = & \mathbf{0} \\ \mathit{half}\,(\mathbf{s}(\mathbf{0})) & = & \mathbf{0} \\ \mathit{half}\,(\mathbf{s}(\mathbf{s}(x'))) & = & \mathbf{s}(\mathit{half}(x')) \end{array}$$

The first step is to break this function into two, each of which steps down by one.

$$\begin{array}{rcl} \mathit{half}_1\,\mathbf{0} & = & \mathbf{0} \\ \mathit{half}_1\,(\mathbf{s}(x')) & = & \mathit{half}_2(x') \end{array}$$

$$\begin{array}{rcl} \mathit{half}_2\,\mathbf{0} & = & \mathbf{0} \\ \mathit{half}_2\,(\mathbf{s}(x'')) & = & \mathbf{s}(\mathit{half}_1(x'')) \end{array}$$

Note that half_1 calls half_2 and vice versa. This is an example of so-called *mutual recursion*. This can be modeled by one function half_{12} returning a pair such that $\mathit{half}_{12}(x) = \langle \mathit{half}_1(x), \mathit{half}_2(x)\rangle$.

$$\begin{array}{rcl} \mathit{half}_{12}\,\mathbf{0} & = & \langle \mathbf{0}, \mathbf{0}\rangle \\ \mathit{half}_{12}\,(\mathbf{s}(x)) & = & \langle \mathbf{snd}\,(\mathit{half}_{12}(x)), \mathbf{s}(\mathbf{fst}\,(\mathit{half}_{12}(x)))\rangle \\ \mathit{half}\;x & = & \mathbf{fst}\,(\mathit{half}_{12}\,x) \end{array}$$

In our notation this becomes

$$\begin{array}{rcl} \mathit{half}_{12} & = & \lambda x \in \mathbf{nat}.\ \mathbf{rec}\; x \\ & & \quad \mathbf{of}\; h(\mathbf{0}) \Rightarrow \langle \mathbf{0}, \mathbf{0}\rangle \\ & & \quad \mid h(\mathbf{s}(x')) \Rightarrow \langle \mathbf{snd}\,(h(x)), \mathbf{s}(\mathbf{fst}\,(h(x)))\rangle \\ \mathit{half} & = & \lambda x \in \mathbf{nat}.\ \mathbf{fst}\,(\mathit{half}_{12}\,x) \end{array}$$

As a last example in the section, consider the subtraction function which cuts off at zero.

$$\begin{array}{rcl} \mathit{minus}\,\mathbf{0}\,y & = & \mathbf{0} \\ \mathit{minus}\,(\mathbf{s}(x'))\,\mathbf{0} & = & \mathbf{s}(x') \\ \mathit{minus}\,(\mathbf{s}(x'))\,(\mathbf{s}(y')) & = & \mathit{minus}\;x'\;y' \end{array}$$

To be presented in the schema of primitive recursion, this requires two nested case distinctions: the outermost one on the first argument x, the innermost one

on the second argument y. So the result of the first application of *minus* must be function, which is directly represented in the definition below.

$$\begin{array}{lcl} \mathit{minus} & = & \lambda x \in \mathbf{nat}.\, \mathbf{rec}\ x \\ & & \quad \mathbf{of}\ m(\mathbf{0}) \Rightarrow \lambda y \in \mathbf{nat}.\, 0 \\ & & \quad \mid m(\mathbf{s}(x')) \Rightarrow \lambda y \in \mathbf{nat}.\, \mathbf{rec}\ y \\ & & \qquad\qquad \mathbf{of}\ p(\mathbf{0}) \Rightarrow \mathbf{s}(x') \\ & & \qquad\qquad \mid p(\mathbf{s}(y')) \Rightarrow (m\,(x'))\,y' \end{array}$$

Note that m is correctly applied only to x', while p is not used at all. So the inner recursion could have been written as a **case**-expression instead.

Functions defined by primitive recursion terminate. This is because the behavior of the function on $\mathbf{s}(n)$ is defined in terms of the behavior on n. We can therefore count down to $\mathbf{0}$, in which case no recursive call is allowed. An alternative approach is to take **case** as primitive and allow arbitrary recursion. In such a language it is much easier to program, but not every function terminates. We will see that for our purpose about integrating constructive reasoning and functional programming it is simpler if all functions one can write down are *total*, that is, are defined on all arguments. This is because total functions can be used to provide witnesses for propositions of the form $\forall x \in \mathbf{nat}.\, \exists y \in \mathbf{nat}.\, P(x,y)$ by showing how to compute y from x. Functions that may not return an appropriate y cannot be used in this capacity and are generally much more difficult to reason about.

3.6 Booleans

Another simple example of a data type is provided by the Boolean type with two elements **true** and **false**. This should *not* be confused with the propositions $\top$ and $\bot$. In fact, they correspond to the unit type $\mathbf{1}$ and the empty type $\mathbf{0}$. We recall their definitions first, in analogy with the propositions.

$$\frac{}{\mathbf{1}\ \mathit{type}}\,\mathbf{1}F$$

$$\frac{}{\Gamma \vdash \langle\rangle \in \mathbf{1}}\,\mathbf{1}I \qquad \textit{no } \mathbf{1} \textit{ elimination rule}$$

$$\frac{}{\mathbf{0}\ \mathit{type}}\,\mathbf{0}F$$

$$\textit{no } \mathbf{0} \textit{ introduction rule} \qquad \frac{\Gamma \vdash t \in \mathbf{0}}{\Gamma \vdash \mathbf{abort}^{\tau}\, t \in \tau}\,\mathbf{0}E$$

There are no reduction rules at these types.

The Boolean type, **bool**, is instead defined by two introduction rules.

$$\frac{}{\mathbf{bool}\ \mathit{type}}\,\mathbf{bool}F$$

$$\frac{}{\Gamma \vdash \mathbf{true} \in \mathbf{bool}}\,\mathbf{bool}I_1 \qquad \frac{}{\Gamma \vdash \mathbf{false} \in \mathbf{bool}}\,\mathbf{bool}I_0$$

The elimination rule follows the now familiar pattern: since there are two introduction rules, we have to distinguish two cases for a given Boolean value. This could be written as

$$\mathbf{case}\, t\, \mathbf{of}\, \mathbf{true} \Rightarrow s_1 \mid \mathbf{false} \Rightarrow s_0$$

but we typically express the same program as an $\mathbf{if}\, t\, \mathbf{then}\, s_1\, \mathbf{else}\, s_0$.

$$\frac{\Gamma \vdash t \in \mathbf{bool} \qquad \Gamma \vdash s_1 \in \tau \qquad \Gamma \vdash s_0 \in \tau}{\Gamma \vdash \mathbf{if}\, t\, \mathbf{then}\, s_1\, \mathbf{else}\, s_0 \in \tau}\, \mathbf{bool}E$$

The reduction rules just distinguish the two cases for the subject of the **if**-expression.

$$\begin{array}{rcl} \mathbf{if}\, \mathbf{true}\, \mathbf{then}\, s_1\, \mathbf{else}\, s_0 & \Longrightarrow & s_1 \\ \mathbf{if}\, \mathbf{false}\, \mathbf{then}\, s_1\, \mathbf{else}\, s_0 & \Longrightarrow & s_0 \end{array}$$

Now we can define typical functions on booleans, such as *and*, *or*, and *not*.

$$\begin{array}{rcl} \mathit{and} & = & \lambda x \in \mathbf{bool}.\, \lambda y \in \mathbf{bool}. \\ & & \quad \mathbf{if}\, x\, \mathbf{then}\, y\, \mathbf{else}\, \mathbf{false} \\ \mathit{or} & = & \lambda x \in \mathbf{bool}.\, \lambda y \in \mathbf{bool}. \\ & & \quad \mathbf{if}\, x\, \mathbf{then}\, \mathbf{true}\, \mathbf{else}\, y \\ \mathit{not} & = & \lambda x \in \mathbf{bool}. \\ & & \quad \mathbf{if}\, x\, \mathbf{then}\, \mathbf{false}\, \mathbf{else}\, \mathbf{true} \end{array}$$

3.7 Lists

Another more interesting data type is that of lists. Lists can be created with elements from any type whatsoever, which means that $\tau\, \mathbf{list}$ is a type for any type τ.

$$\frac{\tau\ \mathit{type}}{\tau\, \mathbf{list}\ \mathit{type}}\, \mathbf{list}F$$

Lists are built up from the empty list (**nil**) with the operation :: (pronounced "cons"), written in infix notation.

$$\frac{}{\Gamma \vdash \mathbf{nil}^{\tau}\ \in \tau\, \mathbf{list}}\, \mathbf{list}I_n \qquad \frac{\Gamma \vdash t \in \tau \qquad \Gamma \vdash s \in \tau\, \mathbf{list}}{\Gamma \vdash t :: s \in \tau\, \mathbf{list}}\, \mathbf{list}I_c$$

The elimination rule implements the schema of primitive recursion over lists. It can be specified as follows:

$$\begin{array}{rcl} f\,(\mathbf{nil}) & = & s_n \\ f\,(x :: l) & = & s_c(x, l, f(l)) \end{array}$$

where we have indicated that s_c may mention x, l, and $f(l)$, but no other occurrences of f. Again this guarantees termination.

$$\frac{\Gamma \vdash t \in \tau\, \mathbf{list} \qquad \Gamma \vdash s_n \in \sigma \qquad \Gamma, x \in \tau, l \in \tau\, \mathbf{list}, f(l) \in \sigma \vdash s_c \in \sigma}{\Gamma \vdash \mathbf{rec}\, t\, \mathbf{of}\, f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c \in \sigma}\, \mathbf{list}E$$

We have overloaded the **rec** constructor here—from the type of t we can always tell if it should recurse over natural numbers or lists. The reduction rules are once again recursive, as in the case for natural numbers.

$$\begin{array}{rcl}
(\mathbf{rec}\,\mathbf{nil}\,\mathbf{of}\,f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c) & \Longrightarrow & s_n \\
(\mathbf{rec}\,(h :: t)\,\mathbf{of}\,f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c) & \Longrightarrow & \\
[(\mathbf{rec}\,t\,\mathbf{of}\,f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c)/f(l)]\,[h/x]\,[t/l]\,s_c &&
\end{array}$$

Now we can define typical operations on lists via primitive recursion. A simple example is the *append* function to concatenate two lists.

$$\begin{array}{rcl}
\mathit{append}\,\mathbf{nil}\,k & = & k \\
\mathit{append}\,(x :: l')\,k & = & x :: (\mathit{append}\,l'\,k)
\end{array}$$

In the notation of primitive recursion:

$$\begin{array}{rcl}
\mathit{append} & = & \lambda l \in \tau\,\mathbf{list}.\,\lambda k \in \tau\,\mathbf{list}.\,\mathbf{rec}\ l \\
& & \qquad \mathbf{of}\ a(\mathbf{nil}) \Rightarrow k \\
& & \qquad \mid\ a(x :: l') \Rightarrow x :: (a\,l') \\
\vdash \mathit{append} & \in & \tau\,\mathbf{list} \to \tau\,\mathbf{list} \to \tau\,\mathbf{list}
\end{array}$$

Note that the last judgment is parametric in τ, a situation referred to as *parametric polymorphism*. In means that the judgment is valid for every type τ. We have encountered a similar situation, for example, when we asserted that $(A \wedge B) \supset A$ *true*. This judgment is parametric in A and B, and every instance of it by propositions A and B is evident, according to our derivation.

As a second example, we consider a program to reverse a list. The idea is to take elements out of the input list l and attach them to the front of a second list a one which starts out empty. The first list has been traversed, the second has accumulated the original list in reverse. If we call this function *rev* and the original one *reverse*, it satisfies the following specification.

$$\begin{array}{rcl}
\mathit{rev} & \in & \tau\,\mathbf{list} \to \tau\,\mathbf{list} \to \tau\,\mathbf{list} \\
\mathit{rev}\,\mathbf{nil}\,a & = & a \\
\mathit{rev}\,(x :: l')\,a & = & \mathit{rev}\,l'\,(x :: a) \\
\\
\mathit{reverse} & \in & \tau\,\mathbf{list} \to \tau\,\mathbf{list} \\
\mathit{reverse}\,l & = & \mathit{rev}\,l\,\mathbf{nil}
\end{array}$$

In programs of this kind we refer to a as the *accumulator argument* since it accumulates the final result which is returned in the base case. We can see that except for the additional argument a, the *rev* function is primitive recursive. To make this more explicit we can rewrite the definition of *rev* to the following equivalent form:

$$\begin{array}{rcl}
\mathit{rev}\,\mathbf{nil} & = & \lambda a.\,a \\
\mathit{rev}\,(x :: l) & = & \lambda a.\,\mathit{rev}\,l\,(x :: a)
\end{array}$$

Now the transcription into our notation is direct.

$$\begin{array}{rcl}
\mathit{rev} & = & \lambda l \in \tau\,\mathbf{list}.\,\mathbf{rec}\ l \\
& & \qquad \mathbf{of}\ r(\mathbf{nil}) \Rightarrow \lambda a \in \tau\,\mathbf{list}.\,a \\
& & \qquad \mid\ r(x :: l') \Rightarrow \lambda a \in \tau\,\mathbf{list}.\,r\,(l')\,(x :: a) \\
\mathit{reverse}\,l & = & \mathit{rev}\,l\,\mathbf{nil}
\end{array}$$

Finally a few simple functions which mix data types. The first counts the number of elements in a list.

$$\begin{array}{rcl} \mathit{length} & \in & \tau\,\mathbf{list} \to \mathbf{nat} \\ \mathit{length}\ \mathbf{nil} & = & \mathbf{0} \\ \mathit{length}\ (x :: l') & = & \mathbf{s}(\mathit{length}\ (l')) \end{array}$$

$$\begin{array}{rcl} \mathit{length} & = & \lambda x \in \tau\,\mathbf{list}.\ \mathbf{rec}\ x \\ & & \quad \mathbf{of}\ \mathit{le}(\mathbf{nil}) \Rightarrow \mathbf{0} \\ & & \quad \mid \mathit{le}(x :: l') \Rightarrow \mathbf{s}(\mathit{le}\ (l')) \end{array}$$

The second compares two numbers for equality.

$$\begin{array}{rcl} \mathit{eq} & \in & \mathbf{nat} \to \mathbf{nat} \to \mathbf{bool} \\ \mathit{eq}\ \mathbf{0}\ \mathbf{0} & = & \mathbf{true} \\ \mathit{eq}\ \mathbf{0}\ (\mathbf{s}(y')) & = & \mathbf{false} \\ \mathit{eq}\ (\mathbf{s}(x'))\ \mathbf{0} & = & \mathbf{false} \\ \mathit{eq}\ (\mathbf{s}(x'))\ (\mathbf{s}(y')) & = & \mathit{eq}\ x'\ y' \end{array}$$

As in the example of subtraction, we need to distinguish two levels.

$$\begin{array}{rcl} \mathit{eq} & = & \lambda x \in \mathbf{nat}.\ \mathbf{rec}\ x \\ & & \quad \mathbf{of}\ e(\mathbf{0}) \Rightarrow \lambda y \in \mathbf{nat}.\ \mathbf{rec}\ y \\ & & \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{true} \\ & & \qquad \mid f(\mathbf{s}(y')) \Rightarrow \mathbf{false} \\ & & \quad \mid e(\mathbf{s}(x')) \Rightarrow \lambda y \in \mathbf{nat}.\ \mathbf{rec}\ y \\ & & \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{false} \\ & & \qquad \mid f(\mathbf{s}(y')) \Rightarrow e(x')\ y' \end{array}$$

We will see more examples of primitive recursive programming as we proceed to first order logic and quantification.

3.8 Summary of Data Types

Judgments.

$\tau\ \mathit{type}$	τ is a type
$t \in \tau$	t is a term of type τ

Type Formation.

$$\frac{}{\mathbf{nat}\ \mathit{type}}\,\mathbf{nat}F \qquad \frac{}{\mathbf{bool}\ \mathit{type}}\,\mathbf{bool}F \qquad \frac{\tau\ \mathit{type}}{\tau\,\mathbf{list}\ \mathit{type}}\,\mathbf{list}F$$

Term Formation.

$$\frac{}{\mathbf{0} \in \mathbf{nat}}\,\mathbf{nat}I_0 \qquad \frac{n \in \mathbf{nat}}{\mathbf{s}(n) \in \mathbf{nat}}\,\mathbf{nat}I_s$$

$$\frac{\Gamma \vdash t \in \mathbf{nat} \qquad \Gamma \vdash t_0 \in \tau \qquad \Gamma, x \in \mathbf{nat}, f(x) \in \tau \vdash t_s \in \tau}{\Gamma \vdash \mathbf{rec}\ t\ \mathbf{of}\ f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s \in \tau}\,\mathbf{nat}E$$

$$\frac{}{\Gamma \vdash \mathbf{true} \in \mathbf{bool}}\,\mathbf{bool}I_1 \qquad \frac{}{\Gamma \vdash \mathbf{false} \in \mathbf{bool}}\,\mathbf{bool}I_0$$

$$\frac{\Gamma \vdash t \in \mathbf{bool} \qquad \Gamma \vdash s_1 \in \tau \qquad \Gamma \vdash s_0 \in \tau}{\Gamma \vdash \mathbf{if}\ t\ \mathbf{then}\ s_1\ \mathbf{else}\ s_0 \in \tau}\,\mathbf{bool}E$$

$$\frac{}{\Gamma \vdash \mathbf{nil}^\tau\ \in \tau\,\mathbf{list}}\,\mathbf{list}I_n \qquad \frac{\Gamma \vdash t \in \tau \qquad \Gamma \vdash s \in \tau\,\mathbf{list}}{\Gamma \vdash t :: s \in \tau\,\mathbf{list}}\,\mathbf{list}I_c$$

$$\frac{\Gamma \vdash t \in \tau\,\mathbf{list} \qquad \Gamma \vdash s_n \in \sigma \qquad \Gamma, x \in \tau, l \in \tau\,\mathbf{list}, f(l) \in \sigma\,\mathbf{list} \vdash s_c \in \sigma}{\Gamma \vdash \mathbf{rec}\,t\ \mathbf{of}\ f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c \in \sigma}\,\mathbf{list}E$$

Reductions.

$$\begin{aligned}
(\mathbf{rec}\ \mathbf{0}\ \mathbf{of}\ f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s) &\implies t_0 \\
(\mathbf{rec}\ \mathbf{s}(n)\ \mathbf{of}\ f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s) &\implies \\
[(\mathbf{rec}\ n\ \mathbf{of}\ f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s)/f(x)]\,[n/x]\,t_s & \\
\mathbf{if}\ \mathbf{true}\ \mathbf{then}\ s_1\ \mathbf{else}\ s_0 &\implies s_1 \\
\mathbf{if}\ \mathbf{false}\ \mathbf{then}\ s_1\ \mathbf{else}\ s_0 &\implies s_0 \\
(\mathbf{rec}\,\mathbf{nil}\ \mathbf{of}\ f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c) &\implies s_n \\
(\mathbf{rec}\,(h :: t)\ \mathbf{of}\ f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c) &\implies \\
[(\mathbf{rec}\,t\ \mathbf{of}\ f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c)/f(l)]\,[h/x]\,[t/l]\,s_c &
\end{aligned}$$

3.9 Predicates on Data Types

In the preceding sections we have introduced the concept of a type which is determined by its elements. Examples were natural numbers, Booleans, and lists. In the next chapter we will explicitly quantify over elements of types. For example, we may assert that every natural number is either even or odd. Or we may claim that any two numbers possess a greatest common divisor. In order to formulate such statements we need some basic propositions concerned with data types. In this section we will define such predicates, following our usual methodology of using introduction and elimination rules to define the meaning of propositions.

We begin with $n < m$, the less-than relation between natural numbers. We have the following formation rule:

$$\frac{\Gamma \vdash m \in \mathbf{nat} \quad \Gamma \vdash n \in \mathbf{nat}}{\Gamma \vdash m < n \ \mathit{prop}} <F$$

Note that this formation rule for propositions relies on the judgment $t \in \tau$. Consequently, we have to permit a hypothetical judgment, in case n or m mention variables declared with their type, such as $x \in \mathbf{nat}$. Thus, in general, the question whether A *prop* may now depend on assumptions of the form $x \in \tau$.

This has a consequence for the judgment A *true*. As before, we now must allow assumptions of the form B *true*, but in addition we must permit assumptions of the form $x \in \tau$. We still call the collection of such assumptions a *context* and continue to denote it with Γ.

$$\frac{}{\Gamma \vdash \mathbf{0} < \mathbf{s}(n) \ \mathit{true}} <I_0 \qquad \frac{\Gamma \vdash m < n \ \mathit{true}}{\Gamma \vdash \mathbf{s}(m) < \mathbf{s}(n) \ \mathit{true}} <I_s$$

The second rule exhibits a new phenomenon: the relation '$<$' whose meaning we are trying to define appears in the premise as well as in the conclusion. In effect, we have not really introduced '$<$', since it already occurs. However, such a definition is still justified, since the conclusion defines the meaning of $\mathbf{s}(m) < \cdot$ in terms of $m < \cdot$. We refer to this relation as *inductively defined*. Actually we have already seen a similar phenomenon in the second "introduction" rule for **nat**:

$$\frac{\Gamma \vdash n \in \mathbf{nat}}{\Gamma \vdash \mathbf{s}(n) \in \mathbf{nat}} \mathbf{nat}I_s$$

The type **nat** we are trying to define already occurs in the premise! So it may be better to think of this rule as a formation rule for the successor operation on natural numbers, rather than an introduction rule for natural numbers.

Returning to the less-than relation, we have to derive the elimination rules. What can we conclude from $\Gamma \vdash m < n$ *true*? Since there are two introduction rules, we could try our previous approach and distinguish cases for the proof of that judgment. This, however, is somewhat awkward in this case—we postpone discussion of this option until later. Instead of distinguishing cases for the proof of the judgment, we distinguish cases for m and n. In each case, we analyse how the resulting judgment could be proven and write out the corresponding elimination rule. First, if n is zero, then the judgment can never have a normal proof, since no introduction rule applies. Therefore we are justified in concluding anything, as in the elimination rule for falsehood.

$$\frac{\Gamma \vdash m < \mathbf{0} \ \mathit{true}}{\Gamma \vdash C \ \mathit{true}} <E_0$$

If the $m = \mathbf{0}$ and $n = \mathbf{s}(n')$, then it could be inferred only by the first introduction rule $<I_0$. This yields no information, since there are no premises to this rule. This is just as in the case of the true proposition $\top$.

The last remaining possibility is that both $m = \mathbf{s}(m')$ and $n = \mathbf{s}(n')$. In that case we now that $m' < n'$, because $<I_s$ is the only rule that could have been applied.

$$\frac{\Gamma \vdash \mathbf{s}(m') < \mathbf{s}(n')\ \mathit{true}}{\Gamma \vdash m' < n'\ \mathit{true}}\,{<}E_s$$

We summarize the formation, introduction, and elimination rules.

$$\frac{\Gamma \vdash n \in \mathbf{nat} \quad \Gamma \vdash m \in \mathbf{nat}}{\Gamma \vdash n < m\ \mathit{prop}}\,{<}F$$

$$\frac{}{\Gamma \vdash \mathbf{0} < \mathbf{s}(n)\ \mathit{true}}\,{<}I_0 \qquad \frac{\Gamma \vdash m < n\ \mathit{true}}{\Gamma \vdash \mathbf{s}(m) < \mathbf{s}(n)\ \mathit{true}}\,{<}I_s$$

$$\frac{\Gamma \vdash m < \mathbf{0}\ \mathit{true}}{\Gamma \vdash C\ \mathit{true}}\,{<}E_0$$

$$\textit{no rule for } \mathbf{0} < \mathbf{s}(n') \qquad \frac{\Gamma \vdash \mathbf{s}(m') < \mathbf{s}(n')\ \mathit{true}}{\Gamma \vdash m' < n'\ \mathit{true}}\,{<}E_s$$

Now we can prove some simple relations between natural numbers. For example:

$$\dfrac{\dfrac{}{\cdot \vdash \mathbf{0} < \mathbf{s}(\mathbf{0})\ \mathit{true}}\,{<}I_0}{\cdot \vdash \mathbf{0} < \mathbf{s}(\mathbf{s}(\mathbf{0}))\ \mathit{true}}\,{<}I_s$$

We can also establish some simple parametric properties of natural numbers.

$$\dfrac{\dfrac{\dfrac{}{m \in \mathbf{nat}, m < 0\ \mathit{true} \vdash m < 0\ \mathit{true}}\,u}{m \in \mathbf{nat}, m < 0\ \mathit{true} \vdash \bot\ \mathit{true}}\,{<}E_0}{m \in \mathbf{nat} \vdash \neg(m < 0)\ \mathit{true}}\,{\supset}I^u$$

In the application of the $<E_0$ rule, we chose $C = \bot$ in order to complete the proof of $\neg(m < \mathbf{0})$. Even slightly more complicated properties, such as $m < \mathbf{s}(m)$ require a proof by induction and are therefore postponed until Section 3.10.

We introduce one further relation between natural numbers, namely equality.

We write $m =_N n$. Otherwise we follow the blueprint of the less-than relation.

$$\frac{\Gamma \vdash m \in \mathbf{nat} \qquad \Gamma \vdash n \in \mathbf{nat}}{\Gamma \vdash m =_N n \ \mathit{prop}} =_N F$$

$$\frac{}{\Gamma \vdash \mathbf{0} =_N \mathbf{0} \ \mathit{true}} =_N I_0 \qquad \frac{\Gamma \vdash m =_N n \ \mathit{true}}{\Gamma \vdash \mathbf{s}(m) =_N \mathbf{s}(n) \ \mathit{true}} =_N I_s$$

$$\mathit{no} =_N E_{00} \ \mathit{elimination\ rule} \qquad \frac{\Gamma \vdash \mathbf{0} =_N \mathbf{s}(n) \ \mathit{true}}{\Gamma \vdash C \ \mathit{true}} =_N E_{0s}$$

$$\frac{\Gamma \vdash \mathbf{s}(m) =_N \mathbf{0} \ \mathit{true}}{\Gamma \vdash C \ \mathit{true}} =_N E_{s0} \qquad \frac{\Gamma \vdash \mathbf{s}(m) =_N \mathbf{s}(n) \ \mathit{true}}{\Gamma \vdash m =_N n \ \mathit{true}} =_N E_{ss}$$

Note the difference between the *function*

$$\mathit{eq} \in \mathbf{nat} \to \mathbf{nat} \to \mathbf{bool}$$

and the *proposition*

$$m =_N n$$

The equality function provides a computation on natural numbers, always returning **true** or **false**. The proposition $m =_N n$ requires *proof*. Using induction, we can later verify a relationship between these two notions, namely that $\mathit{eq}\, n\, m$ reduces to **true** if $m =_N n$ is true, and $\mathit{eq}\, n\, m$ reduces to **false** if $\neg(m =_N n)$.

3.10 Induction

Now that we have introduced the basic propositions regarding order and equality, we can consider induction as a reasoning principle. So far, we have considered the following elimination rule for natural numbers:

$$\frac{\Gamma \vdash t \in \mathbf{nat} \qquad \Gamma \vdash t_0 \in \tau \qquad \Gamma, x \in \mathbf{nat}, f(x) \in \tau \vdash t_s \in \tau}{\Gamma \vdash \mathbf{rec}\ t\ \mathbf{of}\ f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s \in \tau}\ \mathbf{nat}E$$

This rule can be applied if we can derive $t \in \mathbf{nat}$ from our assumptions and we are trying to construct a term $s \in \tau$. But how do we use a variable or term $t \in \mathbf{nat}$ if the judgment we are trying to prove has the form $M : A$, that is, if we are trying the prove the truth of a proposition? The answer is induction. This is actually very similar to primitive recursion. The only complication is that the proposition A we are trying to prove may depend on t. We indicate this by writing $A(x)$ to mean the proposition A with one or more occurrences of a variable x. $A(t)$ is our notation for the result of substituting t for x in A. We

could also write $[t/x]A$, but this is more difficult to read. Informally, induction says that in order to prove $A(t)$ *true* for arbitrary t we have to prove $A(\mathbf{0})$ *true* (the base case), and that for every $x \in \mathbf{nat}$, if $A(x)$ *true* then $A(\mathbf{s}(x))$ *true*.

Formally this becomes:

$$\frac{\Gamma \vdash t \in \mathbf{nat} \qquad \Gamma \vdash A(\mathbf{0})\ \mathit{true} \qquad \Gamma, x \in \mathbf{nat}, A(x)\ \mathit{true} \vdash A(\mathbf{s}(x))\ \mathit{true}}{\Gamma \vdash A(t)\ \mathit{true}}\ \mathbf{nat}E'$$

Here, $A(x)$ is called the *induction predicate*. If t is a variable (which is frequently the case) it is called the *induction variable*. With this rule, we can now prove some more interesting properties. As a simple example we show that $m < \mathbf{s}(m)$ *true* for any natural number m. Here we use $\mathcal{D}$ to stand for the derivation of the third premise in order to overcome the typesetting difficulties.

$$\mathcal{D} = \frac{\dfrac{}{m \in \mathbf{nat}, x \in \mathbf{nat}, x < \mathbf{s}(x)\ \mathit{true} \vdash x < \mathbf{s}(x)\ \mathit{true}}}{m \in \mathbf{nat}, x \in \mathbf{nat}, x < \mathbf{s}(x)\ \mathit{true} \vdash \mathbf{s}(x) < \mathbf{s}(\mathbf{s}(x))}\ {<}I_s$$

$$\frac{\dfrac{}{m \in \mathbf{nat} \vdash m \in \mathbf{nat}} \qquad \dfrac{}{m \in \mathbf{nat} \vdash \mathbf{0} < \mathbf{s}(\mathbf{0})}\ {<}I_0 \qquad \mathcal{D}}{m \in \mathbf{nat} \vdash m < \mathbf{s}(m)}\ \mathbf{nat}E'$$

The property $A(x)$ appearing in the induction principle is $A(x) = x < \mathbf{s}(x)$. So the final conclusion is $A(m) = m < \mathbf{s}(m)$. In the second premise we have to prove $A(\mathbf{0}) = \mathbf{0} < \mathbf{s}(\mathbf{0})$ which follows directly by an introduction rule.

Despite the presence of the induction rule, there are other properties we cannot yet prove easily since the logic does not have quantifiers. An example is the decidability of equality: For any natural numbers m and n, either $m =_N n$ or $\neg(m =_N n)$. This is an example of the practical limitations of *quantifier-free induction*, that is, induction where the induction predicate does not contain any quantifiers.

The topic of this chapter is the interpretation of constructive proofs as programs. So what is the computational meaning of induction? It actually corresponds very closely to primitive recursion.

$$\frac{\Gamma \vdash t \in \mathbf{nat} \qquad \Gamma \vdash M : A(\mathbf{0}) \qquad \Gamma, x \in \mathbf{nat}, u(x){:}A(x) \vdash N : A(\mathbf{s}(x))}{\Gamma \vdash \mathbf{ind}\ t\ \mathbf{of}\ u(\mathbf{0}) \Rightarrow M \mid u(\mathbf{s}(x)) \Rightarrow N : A(t)}\ \mathbf{nat}E'$$

Here, $u(x)$ is just the notation for a variable which may occur in N. Note that u cannot occur in M or in N in any other form. The reduction rules are precisely the same as for primitive recursion.

$$\begin{array}{rcl} (\mathbf{ind}\ \mathbf{0}\ \mathbf{of}\ u(\mathbf{0}) \Rightarrow M \mid u(\mathbf{s}(x)) \Rightarrow N) & \Longrightarrow & M \\ (\mathbf{ind}\ \mathbf{s}(n)\ \mathbf{of}\ u(\mathbf{0}) \Rightarrow M \mid u(\mathbf{s}(x)) \Rightarrow N) & \Longrightarrow & \\ [(\mathbf{ind}\ n\ \mathbf{of}\ u(\mathbf{0}) \Rightarrow M \mid u(\mathbf{s}(x)) \Rightarrow N)/u(n)]\ [n/x]N & & \end{array}$$

We see that primitive recursion and induction are almost identical. The only difference is that primitive recursion returns an element of a type, while induction generates a proof of a proposition. Thus one could say that they are related by an extension of the Curry-Howard correspondence. However, not every type τ can be naturally interpreted as a proposition (which proposition, for example, is expressed by **nat**?), so we no longer speak of an isomorphism.

We close this section by the version of the rules for the basic relations between natural numbers that carry proof terms. This annotation of the rules is straightforward.

$$\frac{\Gamma \vdash n \in \mathbf{nat} \quad \Gamma \vdash m \in \mathbf{nat}}{\Gamma \vdash n < m \ prop} <F$$

$$\frac{}{\Gamma \vdash \mathbf{lt}_0 : \mathbf{0} < \mathbf{s}(n)} <I_0 \qquad \frac{\Gamma \vdash M : m < n}{\Gamma \vdash \mathbf{lt}_s(M) : \mathbf{s}(m) < \mathbf{s}(n)} <I_s$$

$$\frac{\Gamma \vdash M : m < \mathbf{0}}{\Gamma \vdash \mathbf{ltE}_0(M) : C} <E_0$$

$$\textit{no rule for } \mathbf{0} < \mathbf{s}(n') \qquad \frac{\Gamma \vdash M : \mathbf{s}(m') < \mathbf{s}(n')}{\Gamma \vdash \mathbf{ltE}_s(M) : m' < n'} <E_s$$

$$\frac{\Gamma \vdash m \in \mathbf{nat} \qquad \Gamma \vdash n \in \mathbf{nat}}{\Gamma \vdash m =_N n \ prop} =_N F$$

$$\frac{}{\Gamma \vdash \mathbf{eq}_0 : \mathbf{0} =_N \mathbf{0}} =_N I_0 \qquad \frac{\Gamma \vdash M : m =_N n}{\Gamma \vdash \mathbf{eq}_s(M) : \mathbf{s}(m) =_N \mathbf{s}(n)} =_N I_s$$

$$\textit{no } =_N E_{00} \textit{ elimination rule} \qquad \frac{\Gamma \vdash M : \mathbf{0} =_N \mathbf{s}(n)}{\Gamma \vdash \mathbf{eqE}_{0s}(M) : C} =_N E_{0s}$$

$$\frac{\Gamma \vdash M : \mathbf{s}(m) =_N \mathbf{0}}{\Gamma \vdash \mathbf{eqE}_{s0}(M) : C} =_N E_{s0} \qquad \frac{\Gamma \vdash M : \mathbf{s}(m) =_N \mathbf{s}(n)}{\Gamma \vdash \mathbf{eqE}_{ss}(M) : m =_N n} =_N E_{ss}$$

Chapter 4

First-Order Logic and Type Theory

In the first chapter we developed the logic of pure propositions without reference to data types such as natural numbers. In the second chapter we explained the computational interpretation of proofs, and, separately, introduced several data types and ways to compute with them using primitive recursion.

In this chapter we will put these together, which allows us to reason about data and programs manipulating data. In other words, we will be able to prove our programs correct with respect to their expected behavior on data. The principal means for this is induction, introduced at the end of the last chapter. There are several ways to employ the machinery we will develop. For example, we can execute proofs directly, using their interpretation as programs. Or we can *extract* functions, ignoring some proof objects that have are irrelevant with respect to the data our programs return. That is, we can contract proofs to programs. Or we can simply write our programs and use the logical machinery we have developed to prove them correct.

In practice, there are situations in which each of them is appropriate. However, we note that in practice we rarely formally prove our programs to be correct. This is because there is no mechanical procedure to establish if a given programs satisfies its specification. Moreover, we often have to deal with input or output, with mutable state or concurrency, or with complex systems where the specification itself could be as difficult to develop as the implementation. Instead, we typically convince ourselves that central parts of our program and the critical algorithms are correct. Even if proofs are never formalized, this chapter will help you in reasoning about programs and their correctness.

There is another way in which the material of this chapter is directly relevant to computing practice. In the absence of practical methods for verifying full correctness, we can be less ambitious by limiting ourselves to program properties that can indeed be mechanically verified. The most pervasive application of this idea in programming is the idea of *type systems*. By checking the type

correctness of a program we fall far short of verifying it, but we establish a kind of consistency statement. Since languages satisfy (or are supposed to satisfy) type preservation, we know that, if a result is returned, it is a value of the right type. Moreover, during the execution of a program (modelled here by reduction), all intermediate states are well-typed which prevents certain absurd situations, such as adding a natural number to a function. This is often summarized in the slogan that "*well-typed programs cannot go wrong*". Well-typed programs are *safe* in this respect. In terms of machine language, assuming a correct compiler, this guards against irrecoverable faults such as jumping to an address that does not contain valid code, or attempting to write to inaccessible memory location.

There is some room for exploring the continuum between types, as present in current programming languages, and full specifications, the domain of *type theory*. By presenting these elements in a unified framework, we have the basis for such an exploration.

We begin this chapter with a discussion of the universal and existential quantifiers, followed by a number of examples of inductive reasoning with data types.

4.1 Quantification

In this section, we introduce universal and existential quantification. As usual, we follow the method of using introduction and elimination rules to explain the meaning of the connectives. First, universal quantification, written as $\forall x{\in}\tau.\ A(x)$. For this to be well-formed, the body must be well-formed under the assumption that x is a variable of type τ.

$$\frac{\tau\ \mathit{type} \qquad \Gamma, x{\in}\tau \vdash A(x)\ \mathit{prop}}{\Gamma \vdash \forall x{\in}\tau.\ A(x)\ \mathit{prop}}\,\forall F$$

For the introduction rule we require that $A(x)$ be valid for arbitrary x. In other words, the premise contains a parametric judgment.

$$\frac{\Gamma, x{\in}\tau \vdash A(x)\ \mathit{true}}{\Gamma \vdash \forall x{\in}\tau.\ A(x)\ \mathit{true}}\,\forall I$$

If we think of this as the defining property of universal quantification, then a verification of $\forall x{\in}\tau.\ A(x)$ describes a construction by which an arbitrary $t \in \tau$ can be transformed into a proof of $A(t)\ \mathit{true}$.

$$\frac{\Gamma \vdash \forall x{\in}\tau.\ A(x)\ \mathit{true} \qquad \Gamma \vdash t \in \tau}{\Gamma \vdash A(t)\ \mathit{true}}\,\forall E$$

We must verify that $t \in \tau$ so that $A(t)$ is a proposition. We can see that the computational meaning of a proof of $\forall x{\in}\tau.\ A(x)\ \mathit{true}$ is a function which, when

given an argument t of type τ, returns a proof of $A(t)$. If we don't mind overloading application, the proof term assignment for the universal introduction and elimination rule is

$$\frac{\Gamma, x{\in}\tau \vdash M : A(x)}{\Gamma \vdash \lambda x{\in}\tau.\ M : \forall x{\in}\tau.\ A(x)}\,\forall I$$

$$\frac{\Gamma \vdash M : \forall x{\in}\tau.\ A(x) \qquad \Gamma \vdash t \in \tau}{\Gamma \vdash M\,t : A(t)}\,\forall E$$

The computation rule simply performs the required substitution.

$$(\lambda x{\in}\tau.\ M)\,t \quad\Longrightarrow\quad [t/x]M$$

The existential quantifier $\exists x{\in}\tau.\ A(x)$ lies at the heart of constructive mathematics. This should be a proposition if $A(x)$ is a proposition under the assumption that x has type τ.

$$\frac{\tau\ \mathit{type} \qquad \Gamma, x{\in}\tau \vdash A(x)\ \mathit{prop}}{\Gamma \vdash \exists x{\in}\tau.\ A(x)\ \mathit{prop}}\,\exists F$$

The introduction rule requires that we have a *witness* term t and a proof that t satisfies property A.

$$\frac{\Gamma \vdash t \in \tau \qquad \Gamma \vdash A(t)\ \mathit{true}}{\Gamma \vdash \exists x{\in}\tau.\ A(x)\ \mathit{true}}\,\exists I$$

The elimination rule bears some resemblance to disjunction: if we know that we have a verification of $\exists x{\in}\tau.\ A(x)$ we do not know the witness t. As a result we cannot simply write a rule of the form

$$\frac{\Gamma \vdash \exists x{\in}\tau.\ A(x)\ \mathit{true}}{\Gamma \vdash t \in \tau}\,\exists E?$$

since we have no way of referring to the proper t. Instead we reason as follows: If $\exists x{\in}\tau.\ A(x)$ is true, then there is some element of τ for which A holds. Call this element x and assume $A(x)$. Whatever we derive from this assumption must be true, as long as it does not depend on x itself.

$$\frac{\Gamma \vdash \exists x{\in}\tau.\ A(x)\ \mathit{true} \qquad \Gamma, x{\in}\tau, A(x)\ \mathit{true} \vdash C\ \mathit{true}}{\Gamma \vdash C\ \mathit{true}}\,\exists E$$

The derivation of the second premise is parametric in x and hypothetical in $A(x)$, that is, x may not occur in Γ or C.

The proof term assignment and computational contents of these rules is not particularly difficult. The proof term for an existential introduction is a pair

consisting of the witness t and the proof that t satisfies the stated property. The elimination rule destructs the pair, making the components accessible.

$$\frac{\Gamma \vdash t \in \tau \qquad \Gamma \vdash M : A(t)}{\Gamma \vdash \langle t, M\rangle : \exists x{\in}\tau.\ A(x)}\,\exists I$$

$$\frac{\Gamma \vdash M : \exists x{\in}\tau.\ A(x) \qquad \Gamma, x{\in}\tau, u{:}A(x) \vdash N : C}{\Gamma \vdash \mathbf{let}\ \langle x, u\rangle = M\ \mathbf{in}\ N : C}\,\exists E$$

The reduction rule is straightforward, substituting both the witness and the proof term certifying its correctness.

$$\mathbf{let}\langle x, u\rangle = \langle t, M\rangle\ \mathbf{in}\ N \quad\Longrightarrow\quad [M/u]\,[t/x]\,N$$

As in the case of the propositional connectives, we now consider various interactions between quantifiers and connectives to obtain an intuition regarding their properties. We continue to denote a proposition A that depends on a variable x by $A(x)$.

Our first example states that universal quantification distributes over conjunction. In order to make it fit on the page, we have abbreviated $u{:}\forall x{\in}\tau.\ A(x){\wedge}\ B(x)$ by $u{:}-$. Furthermore, we named the parameter introduced into the derivation a (rather than x), to emphasize the distinction between a bound variable in a proposition and a parameter which is bound in a derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{a{\in}\tau, u{:}- \vdash \forall x{\in}\tau.\ A(x) \wedge B(x)\ \mathit{true}}\,u \qquad \dfrac{}{a{\in}\tau, u{:}- \vdash a \in \tau}\,a}{u{:}-, a{\in}\tau \vdash A(a) \wedge B(a)\ \mathit{true}}\,\forall E}{u{:}-, a{\in}\tau \vdash A(a)\ \mathit{true}}\,\wedge E_L}{u{:}- \vdash \forall x{\in}\tau.\ A(x)\ \mathit{true}}\,\forall I^a}{\vdash (\forall x{\in}\tau.\ A(x) \wedge B(x)) \supset (\forall x{\in}\tau.\ A(x))\ \mathit{true}}\,\supset I^u$$

The lists of hypotheses of the form $x{\in}\tau$ and $u{:}A$ in each line of a natural deduction can be reconstructed, so we will use the following abbreviated form familiar from the early development of propositional logic.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x{\in}\tau.\ A(x) \wedge B(x)\ \mathit{true}}\,u \qquad \dfrac{}{a \in \tau}\,a}{A(a) \wedge B(a)\ \mathit{true}}\,\forall E}{A(a)\ \mathit{true}}\,\wedge E_L}{\forall x{\in}\tau.\ A(x)\ \mathit{true}}\,\forall I^a}{(\forall x{\in}\tau.\ A(x) \wedge B(x)) \supset (\forall x{\in}\tau.\ A(x))\ \mathit{true}}\,\supset I^u$$

From this deduction it is easy to see that

$$(\forall x{\in}\tau.\ A(x) \wedge B(x)) \supset (\forall x{\in}\tau.\ A(x)) \wedge (\forall x{\in}\tau.\ B(x))\ \mathit{true}$$

By annotating the derivation above we can construct the following proof term for this judgment (omitting some labels):

$$\begin{array}{ll} \vdash & \lambda u.\ \langle \lambda x{\in}\tau.\ \mathbf{fst}\,(u\,x), \lambda x{\in}\tau.\ \mathbf{snd}\,(u\,x)\rangle \\ : & (\forall x{\in}\tau.\ A(x) \wedge B(x)) \supset (\forall x{\in}\tau.\ A(x)) \wedge (\forall x{\in}\tau.\ B(x)) \end{array}$$

The opposite direction also holds, which means that we can freely move the universal quantifier over conjunctions and vice versa. This judgment (and also the proof above) are parametric in τ. Any instance by a concrete type for τ will be an evident judgment. We show here only the proof term (again omitting some labels):

$$\begin{array}{ll} \vdash & \lambda p.\ \lambda x{\in}\tau.\ \langle (\mathbf{fst}\ p)\ x, (\mathbf{snd}\ p)\ x\rangle \\ : & (\forall x{\in}\tau.\ A(x)) \wedge (\forall x{\in}\tau.\ B(x)) \supset (\forall x{\in}\tau.\ A(x) \wedge B(x)) \end{array}$$

The corresponding property for the existential quantifier allows distributing the existential quantifier over disjunction.

$$(\exists x{\in}\tau.\ A(x) \vee B(x)) \equiv (\exists x{\in}\tau.\ A(x)) \vee (\exists x{\in}\tau.\ B(x))$$

We verify one direction.

$$\dfrac{\dfrac{}{\exists x{\in}\tau.\ A(x) \vee B(x)\ \mathit{true}}\,u \qquad \begin{array}{c}\mathcal{D}\\ (\exists x{\in}\tau.\ A(x)) \vee (\exists x{\in}\tau.\ B(x))\ \mathit{true}\end{array}}{(\exists x{\in}\tau.\ A(x)) \vee (\exists x{\in}\tau.\ B(x))\ \mathit{true}}\,\exists E^{a,w}}{(\exists x{\in}\tau.\ A(x) \vee B(x)) \supset (\exists x{\in}\tau.\ A(x)) \vee (\exists x{\in}\tau.\ B(x))\ \mathit{true}}\,\supset I^u$$

where the deduction $\mathcal{D}$ is the following

$$\dfrac{\dfrac{}{A(a) \vee B(a)\ \mathit{true}}\,w \qquad \dfrac{\dfrac{\dfrac{}{a \in \tau}\,a \qquad \dfrac{}{A(a)\ \mathit{true}}\,v_1}{\exists x{\in}\tau.\ A(x)\ \mathit{true}}\,\exists I}{(\exists x{\in}\tau.\ A(x)) \vee (\exists x{\in}\tau.\ B(x))\ \mathit{true}}\,\vee I_L \qquad \vdots}{(\exists x{\in}\tau.\ A(x)) \vee (\exists x{\in}\tau.\ B(x))\ \mathit{true}}\,\vee E^{v_1,v_2}$$

The omitted derivation of the second case in the disjunction elimination is symmetric to the given case and ends in $\vee I_R$.

It is important to keep in mind the restriction on the existential elimination rule, namely that the parameter must be new in the second premise. The following is an incorrect derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a \in \mathbf{nat}}\,a\mathbf{?}}{\mathbf{s}(a) \in \mathbf{nat}}\,\mathbf{nat}I_s \qquad \dfrac{\dfrac{}{\exists x{\in}\mathbf{nat}.\ A(\mathbf{s}(x))\ \mathit{true}}\,u \qquad \dfrac{}{A(\mathbf{s}(a))\ \mathit{true}}\,w}{A(\mathbf{s}(a))\ \mathit{true}}\,\exists E^{a,w}\mathbf{?}}{\exists y{\in}\mathbf{nat}.\ A(y)\ \mathit{true}}\,\exists I}{(\exists x{\in}\mathbf{nat}.\ A(\mathbf{s}(x))) \supset \exists y{\in}\mathbf{nat}.\ A(y)\ \mathit{true}}\,\supset I^u$$

The problem can be seen in the two questionable rules. In the existential introduction, the term a has not yet been introduced into the derivation and its use can therefore not be justified. Related is the incorrect application of the $\exists E$ rule. It is supposed to introduce a new parameter a and a new assumption w. However, a occurs in the conclusion, invalidating this inference.

In this case, the flaw can be repaired by moving the existential elimination downward, in effect introducing the parameter into the derivation earlier (when viewed from the perspective of normal proof construction).

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a \in \mathbf{nat}}\,a}{\mathbf{s}(a) \in \mathbf{nat}}\,\mathbf{nat}I_s \qquad \dfrac{}{A(\mathbf{s}(a))\ true}\,w}{\exists y{\in}\mathbf{nat}.\ A(y)\ true}\,\exists I \qquad \dfrac{}{\exists x{\in}\mathbf{nat}.\ A(\mathbf{s}(x))\ true}\,u}{\exists y{\in}\mathbf{nat}.\ A(y)\ true}\,\exists E^{a,w}}{(\exists x{\in}\mathbf{nat}.\ A(\mathbf{s}(x))) \supset \exists y{\in}\mathbf{nat}.\ A(y)\ true}\,\supset I^u$$

Of course there are other cases where the flawed rule cannot be repaired. For example, it is easy to construct an incorrect derivation of $(\exists x{\in}\tau.\ A(x)) \supset \forall x{\in}\tau.\ A(x)$.

4.2 First-Order Logic

First-order logic, also called the predicate calculus, is concerned with the study of propositions whose quantifiers range over a domain about which we make no assumptions. In our case this means we allow only quantifiers of the form $\forall x{\in}\tau.\ A(x)$ and $\exists x{\in}\tau.\ A(x)$ that are parametric in a type τ. We assume only that τ *type*, but no other property of τ. When we add particular types, such as natural numbers **nat** or lists τ **list**, we say that we reason within specific theories. The theory of natural numbers, for example, is called *arithmetic*. When we allow essentially arbitrary propositions and types explained via introduction and elimination constructs (including function types, product types, etc.) we say that we reason in *type theory*. It is important that type theory is open-ended: we can always add new propositions and new types and even new judgment forms, as long as we can explain their meaning satisfactorily. On the other hand, first-order logic is essentially closed: when we add new constructs, we work in other theories or logics that include first-order logic, but we go beyond it in essential ways.

We have already seen some examples of reasoning in first-order logic in the previous section. In this section we investigate the truth of various other propositions in order to become comfortable with first-order reasoning. Just like propositional logic, first-order logic has both classical and constructive variants. We pursue the constructive or intuitionistic point of view. We can recover classical truth either via an interpretation such as Gödel's translation[1], or by adding

[1] detailed in a separate note by Jeremy Avigad

the law of excluded middle. The practical difference at the first-order level is the interpretation of the existential quantifier. In classical logic, we can prove a proposition $\exists x{\in}\tau.\ A(x)$ *true* by proving $\neg\forall x{\in}\tau.\ \neg A(x)$ *true* instead. Such a proof may not yield the witness object t such that $A(t)$ is satisfied, which is required under the constructive interpretation of the existential quantifier. But how is it possible to provide witnesses in pure logic, without any assumptions about the domain of quantifiers? The answer is that assumptions about the existence of objects will be introduced locally during the proof. But we have to be careful to verify that the objects we use to witness existential quantifiers or instantiate universal quantifiers are indeed assumed to exist and are available at the right point in the derivation.

As a first concrete example, we investigate the interaction between negation and quantification. We prove

$$(\exists x{\in}\tau.\ \neg A(x)) \supset \neg\forall x \in \tau.\ A(x)\ \textit{true}.$$

The subject of the judgment above is a proposition, assuming τ *type* and $x{\in}\tau \vdash A(x)$ *prop*. Since all quantifiers range over the same type τ, we will omit the type label from quantification in all propositions below. The reader should keep in mind that this is merely a shorthand. Furthermore, we will not explicitly state the assumption about the propositional or predicate parameters such as $A(x)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\exists x.\ \neg A(x)}\,u \qquad \dfrac{\dfrac{}{\neg A(c)}\,w \qquad \dfrac{\dfrac{}{\forall x.\ A(x)}\,v \quad \dfrac{}{c \in \tau}\,c}{A(c)}\,\forall E}{\bot}\,\supset E}{\bot}\,\exists E^{c,w}}{\neg\forall x.\ A(x)}\,\supset I^{v}}{(\exists x.\ \neg A(x)) \supset \neg\forall x.\ A(x)}\,\supset I^{u}$$

The two-dimensional notation for derivations becomes difficult to manage for large proofs, so we extend the linear notation from Section 2.8. We use the following concrete syntax.

$\forall x{\in}\tau.\ A(x)$	`!x:t. A(x)`
$\exists x{\in}\tau.\ A(x)$	`?x:t. A(x)`
$c \in \tau$	`c : t`

The quantifiers $\forall$ and $\exists$ act like a prefix operator with minimal binding strength, so that

$$\forall x{\in}\tau.\ A(x) \supset B$$

is the same as

$$\forall x{\in}\tau.\ (A(x) \supset B).$$

One complication introduced by existential quantification is that the elimination rule introduces two new assumptions, $c \in \tau$ and $A(c)$ *true*. In order to

distinguish between inferred and assumed judgments, new assumptions are separated by commas and terminated by semi-colon. Under these conventions, the four rules for quantification take the following form:

Introduction	Elimination

`c : t;`
`A(c);`
`?x:t. A(x);`
	`?x:t. A(x);`
	`[c : t, A(c);`
	`  ... ;`
	`B];`
	`B;`

`[c : t;`
`  ... ;`
`A(c)];`
`!x:t. A(x)`
	`!x:t. A(x);`
	`c : t;`
	`A(c);`

We use `c` as a new parameter to distinguish parameters more clearly from bound variables. Their confusion is a common source of error in first-order reasoning. And we have the usual assumption that the name chosen for c must be new (that is, may not occur in $A(x)$ or B) in the existential elimination and universal introduction rules.

Below we restate the proof from above in the linear notation.

```
[ ?x:t. ~A(x);
  [ !x:t. A(x);
    [ c : t, ~A(c);
      A(c);
      F ];
    F ];
  ~!x:t. A(x) ];
(?x:t. ~A(x)) => ~!x:t. A(x);
```

The opposite implication does not hold: even if we know that it is impossible that $A(x)$ is true for every x, this does not necessarily provide us with enough information to obtain a witness for $\exists x.\ A(x)$. In order to verify that this cannot be proven without additional information about A, we need to extend our notion of normal and neutral proof. This is straightforward—only the existential elimination rule requires some thought. It is treated in analogy with disjunction.

$$\frac{\Gamma, c{\in}\tau \vdash A(c) \uparrow}{\Gamma \vdash \forall x{\in}\tau.\ A(x) \uparrow}\,\forall I \qquad \frac{\Gamma \vdash \forall x{\in}\tau.\ A(x) \downarrow \quad \Gamma \vdash t \in \tau}{\Gamma \vdash A(t) \downarrow}\,\forall E$$

$$\frac{\Gamma \vdash t \in \tau \quad \Gamma \vdash A(t) \uparrow}{\Gamma \vdash \exists x{\in}\tau.\ A(x) \uparrow}\,\exists I \qquad \frac{\Gamma \vdash \exists x{\in}\tau.\ A(x) \downarrow \quad \Gamma, c{\in}\tau, A(c) \downarrow \vdash C \uparrow}{\Gamma \vdash C \uparrow}\,\exists E$$

In the case of pure first-order logic (that is, quantification is allowed only over one unknown type τ), normal proofs remain complete. A correspondingly strong property *fails* for arithmetic, that is, when we allow the type **nat**. This situation is familiar from mathematics, where we often need to generalize the induction hypothesis in order to prove a theorem. This generalization means that the resulting proof does not have a strong normality property. We will return to this topic in the next section.

Now we return to showing that $(\neg\forall x.\ A(x)) \supset \exists x.\ \neg A(x)$ *true* is not derivable. We search for a normal proof, which means the first step in the bottom-up construction is forced and we are in the state

$$\dfrac{\dfrac{}{\neg\forall x.\ A(x)\downarrow}\,u \quad \vdots \quad \exists x.\ \neg A(x)\uparrow}{(\neg\forall x.\ A(x)) \supset \exists x.\ \neg A(x)\uparrow}\,\supset I^u$$

At this point it is impossible to apply the existential introduction rule, because no witness object of type τ is available. So we can only apply the implication elimination rule, which leads us to the following situation.

$$\dfrac{\dfrac{\dfrac{}{\neg\forall x.\ A(x)\downarrow}\,u \quad \vdots \quad \forall x.\ A(x)\uparrow}{\dfrac{}{\neg\forall x.\ A(x)\downarrow}\,u \quad \bot\downarrow}\,\supset E \quad \exists x.\ \neg A(x)\uparrow}{(\neg\forall x.\ A(x)) \supset \exists x.\ \neg A(x)\uparrow}\,\bot E \,\supset I^u$$

Now we can either repeat the negation elimination (which leads nowhere), or use universal introduction.

$$\dfrac{\dfrac{\dfrac{}{\neg\forall x.\ A(x)\downarrow}\,u \quad \dfrac{}{c\in\tau}\,c \quad \vdots \quad A(c)\uparrow}{\dfrac{}{\neg\forall x.\ A(x)\downarrow}\,u \quad \forall x.\ A(x)\uparrow}\,\forall I^c \quad \bot\downarrow}{(\neg\forall x.\ A(x)) \supset \exists x.\ \neg A(x)\uparrow}\,\supset E \,\bot E \,\supset I^u$$

The only applicable rule for constructing normal deductions now is again the implication elimination rule, applied to the assumption labelled u. This leads to

the identical situation, except that we have an additional assumption $d \in \tau$ and try to prove $A(d)\uparrow$. Clearly, we have made no progress (since the assumption $c \in \tau$ is now useless). Therefore the given proposition has no normal proof and hence, by the completeness of normal proofs, no proof.

As a second example, we see that $(\forall x.\ A(x)) \supset \exists x.\ A(x)$ *true* does not have a normal proof. After one forced step, we have to prove

$$\begin{array}{c} \forall x.\ A(x)\downarrow \\ \vdots \\ \exists x.\ A(x)\uparrow \end{array}$$

At this point, no rule is applicable, since we cannot construct *any* term of type τ. Intuitively, this should make sense: if the type τ is empty, then we cannot prove $\exists x{\in}\tau.\ A(x)$ since we cannot provide a witness object. Since we make no assumptions about τ, τ may in fact denote an empty type (such as $\mathbf{0}$), the above is clearly false.

In classical first-order logic, the assumption is often made that the domain of quantification is non-empty, in which case the implication above is true. In type theory, we can prove this implication for specific types that are known to be non-empty (such as **nat**). We can also model the standard assumption that the domain is non-empty by establishing the corresponding hypothetical judgment:

$$c \in \tau \vdash (\forall x{\in}\tau.\ A(x)) \supset \exists x{\in}\tau.\ A(x)$$

We just give this simple proof in our linear notation.

```
[ c : t;
  [ !x:t. A(x);
    A(c);
    ?x:t. A(x) ];
  (!x:t. A(x)) => ?x:t. A(x)];
```

We can also discharge this assumption to verify that

$$\forall y.\ ((\forall x.\ A(x)) \supset \exists x.\ A(x))\ \mathit{true}$$

without any additional assumption. This shows that, in general, $\forall y.\ B$ is not equivalent to B, even if y does not occur in B! While this may be counterintuitive at first, the example above shows why it must be the case. The point is that while y does not occur in the *proposition*, it does occur in the *proof* and can therefore not be dropped.

4.3 Arithmetic

We obtain the system of *first-order arithmetic* if we restrict quantifiers to elements of type **nat**. Recall the induction principle for natural numbers and the rules for equality $n =_N m$ and the less-than relation $n < m$ summarized in Section 3.10.

As a reminder, we will prove some frequently needed properties of equality. The first is reflexivity of equality.

$$\forall x{\in}\mathbf{nat}.\ x =_N x$$

We first give the informal proof, then its translation into a formal proof language.

Proof: The proof is by induction on x.

Case: $x = \mathbf{0}$. Then $\mathbf{0} =_N \mathbf{0}$ by rule $=_N I_0$.

Case: $x = \mathbf{s}(x')$. Then

$x' =_N x'$ by induction hypothesis
$s(x') =_N s(x')$ by rule $=_N I_s$.

□

As a formal proof in linear format:

```
[ x : nat;                % assumption
  0 = 0;                  % by =I0 (base case)
  [x' : nat, x' = x';     % assumptions
   s(x') = s(x')];        % by =Is (induction step)
  x = x ];                % by induction on x
!x:nat. x = x;            % by !I
```

We can also write out the proof term that corresponds to the proof above.

$$\begin{array}{rcl} \mathit{refl} & : & \forall x{\in}\mathbf{nat}.\ x =_N x \\ & = & \lambda x{\in}\mathbf{nat}.\ \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ r(\mathbf{0}) \Rightarrow \mathbf{eq}_0 \\ & & \qquad\ \mid r(\mathbf{s}(x')) \Rightarrow \mathbf{eq}_s(r(x')) \end{array}$$

As a second example, we consider transitivity of equality.

$$\forall x{\in}\mathbf{nat}.\ \forall y{\in}\mathbf{nat}.\ \forall z{\in}\mathbf{nat}.\ x =_N y \supset y =_N z \supset x =_N z$$

This time we will give the proof in three forms: as an informal mathematical proof, as a formal proof in linear form, and as an equational specification proof term.

Proof: The proof is by induction on x. We need to distinguish subcases on y and z.

Case: $x = \mathbf{0}$. Then we distinguish subcases on y.

Case: $y = \mathbf{0}$. Then we distinguish subcases on z.

Case: $z = \mathbf{0}$. Then $\mathbf{0} =_N \mathbf{0}$ by rule $=_N I_0$.

Case: $z = \mathbf{s}(z')$. Then $y =_N z$ is impossible by rule $=_N E_{0s}$.

Case: $y = \mathbf{s}(y')$. Then $x =_N y$ is impossible by rule $=_N E_{0s}$.

Case: $x = \mathbf{s}(x')$. We assume the induction hypothesis

$$\forall y{\in}\mathbf{nat}.\ \forall z{\in}\mathbf{nat}.\ x' =_N y \supset y =_N z \supset x' =_N z$$

and distinguish subcases on y.

Case: $y = \mathbf{0}$. Then $x =_N y$ is impossible by rule $=_N E_{0s}$.

Case: $y = \mathbf{s}(y')$. Then we distinguish subcases on z.

Case: $z = \mathbf{0}$. Then $y =_N z$ is impossible by rule $=_N E_{s0}$.

Case: $z = \mathbf{s}(z')$. Then we assume $\mathbf{s}(x') =_N \mathbf{s}(y')$ and $\mathbf{s}(y') =_N \mathbf{s}(z')$ and have to show that $\mathbf{s}(x') =_N \mathbf{s}(z')$. We continue:

$x' =_N y'$	by rule $=_N E_{ss}$
$y' =_N z'$	by rule $=_N E_{ss}$
$x' =_N z'$	by universal and implication eliminations from induction hypothesis
$\mathbf{s}(x') =_N \mathbf{s}(z')$	by rule $=_N I_s$.

□

The formal proof of transitivity is a good illustration why mathematical proofs are not written as natural deductions: the granularity of the steps is too small even for relatively simple proofs.

```
[ x : nat;
  [ y : nat;
    [ z : nat;
      [ 0 = 0;
        [ 0 = 0;
          0 = 0 ];                          % eqI0
        0 = 0 => 0 = 0 ];
      0 = 0 => 0 = 0 => 0 = 0;              % case (z = 0)
      [ z' : nat, 0 = 0 => 0 = z' => 0 = z';
        [ 0 = 0;
          [ 0 = s(z');
            0 = s(z') ];                    % eqE0s
          0 = s(z') => 0 = s(z') ];
        0 = 0 => 0 = s(z') => 0 = s(z') ];  % case (z = s(z'))
      0 = 0 => 0 = z => 0 = z ];
    !z:nat. 0 = 0 => 0 = z => 0 = z;        % case (y = 0)

    [ y' : nat, !z:nat. 0 = y' => y' = z => 0 = z;
```

```
      [ z : nat;
        [ 0 = s(y');
          [ s(y') = z;
            0 = z ];                          % eqE0s
          s(y') = z => 0 = z ];
        0 = s(y') => s(y') = z => 0 = z ];
      !z:nat. 0 = s(y') => s(y') = z => 0 = z ]; % case (y = s(y'))

    !z:nat. 0 = y => y = z => 0 = z ];
  !y:nat. !z:nat. 0 = y => y = z => 0 = z;    % base case (x = 0)

  [ x' : nat, !y:nat. !z:nat. x' = y => y = z => x' = z;  % ind hyp (x)
    [ y : nat;
      [ z : nat;
        [ s(x') = 0;
          [ 0 = z;
            s(x') = z ];                      % eqEs0
          0 = z => s(x') = z ];
        s(x') = 0 => 0 = z => s(x') = z ];
      !z:nat. s(x') = 0 => 0 = z => s(x') = z;   % case (y = 0)
      [ y' : nat, !z:nat. s(x') = y' => y' = z => s(x') = z;
        [ z : nat;
          [ s(x') = s(y');
            [ s(y') = 0;
              s(x') = 0 ];                    % eqEs0
            s(y') = 0 => s(x') = 0 ];
          s(x') = s(y') => s(y') = 0 => s(x') = 0;     % case (z = 0)

          [ z' : nat, s(x') = s(y') => s(y') = z' => s(x') = z';
            [ s(x') = s(y');
              [ s(y') = s(z');
                x' = y';                      % eqEss
                y' = z';                      % eqEss
                !z:nat. x' = y' => y' = z => x' = z;
                x' = y' => y' = z' => x' = z';
                y' = z' => x' = z';
                x' = z';
                s(x') = s(z') ];              % eqIs
              s(y') = s(z') => s(x') = s(z') ];
            s(x') = s(y') => s(y') = s(z') => s(x') = s(z') ];
          s(x') = s(y') => s(y') = z => s(x') = z ];
        !z:nat. s(x') = s(y') => s(y') = z => s(x') = z ];  % case (y = s(y'))
      !z:nat. s(x') = y => y = z => s(x') = z ];
    !y:nat. !z:nat. s(x') = y => y = z => s(x') = z ];  % ind step (x = s(x'))
  !y:nat. !z:nat. x = y => y = z => x = z ];
!x:nat. !y:nat. !z:nat. x = y => y = z => x = z;
```

Instead of giving the proof term in full, we give its specification. Recall that

$$\mathit{trans} \quad : \quad \forall x{\in}\mathbf{nat}.\, \forall y{\in}\mathbf{nat}.\, \forall z{\in}\mathbf{nat}.\, x =_N y \supset y =_N z \supset x =_N z$$

and therefore *trans* is a function of five arguments: natural numbers x, y, and z and proof terms $u{:}x =_N y$ and $w{:}y =_N z$. It has to return a proof term $M : x =_N z$. The proof above corresponds to the following specification.

$$\begin{array}{llllllll}
\mathit{trans} & \mathbf{0} & \mathbf{0} & \mathbf{0} & u & w & = & \mathbf{eq}_0 \\
\mathit{trans} & \mathbf{0} & \mathbf{0} & (\mathbf{s}(z')) & u & w & = & \mathbf{eqE}_{0s}(w) \\
\mathit{trans} & \mathbf{0} & (\mathbf{s}(y')) & z & u & w & = & \mathbf{eqE}_{0s}(u) \\
\mathit{trans} & (\mathbf{s}(x')) & \mathbf{0} & z & u & w & = & \mathbf{eqE}_{s0}(u) \\
\mathit{trans} & (\mathbf{s}(x')) & (\mathbf{s}(y')) & \mathbf{0} & u & w & = & \mathbf{eqE}_{s0}(w) \\
\mathit{trans} & (\mathbf{s}(x')) & (\mathbf{s}(y')) & (\mathbf{s}(z')) & u & w & = & \\
\multicolumn{8}{r}{\mathbf{eq}_s(\mathit{trans}\, x'\, y'\, z'\, (\mathbf{eqE}_{ss}(u))\, (\mathbf{eqE}_{ss}(w)))}
\end{array}$$

Note that all but the first and the last case are impossible, for which we provide evidence by applying the right elimination rule to either u or w. We can also see that the first argument to the recursive call to *trans* is at x' and the specificaton above therefore satisfies the restriction on primitive recursion. By comparing this to the formal proof (and also the omitted proof term) we can see the programming with equational specifications of this kind is much simpler and more concise than many other representations. There is ongoing research on directly verifying and compiling specifications, which is close to actual programming practice in languages such as ML or Haskell.

Symmetry of equality can be proven in a similar way. This proof and the corresponding specification and proof term are left as an exercise to the reader.

A second class of example moves us closer the extraction of functional programs on natural numbers from constructive proofs. Keeping in mind the constructive interpretation of the existential quantifier, how could we specify the predecessor operation? There are many possible answers to this. Here we would like express that the predecessor should only be applied to positive natural numbers.

$$\forall x{\in}\mathbf{nat}.\, \neg x =_N \mathbf{0} \supset \exists y{\in}\mathbf{nat}.\, \mathbf{s}(y) =_N x$$

We can prove this by cases on x. Formally, this takes the form of an induction in which the induction hypothesis is not used.

Proof: The proof proceeds by cases on x.

Case: $x = \mathbf{0}$. Then the assumption $\neg 0 =_N 0$ is contradictory.

Case: $x = \mathbf{s}(x')$. Assume $\neg \mathbf{s}(x') =_N 0$. We have to show that $\exists y{\in}\mathbf{nat}.\, \mathbf{s}(y) =_N \mathbf{s}(x')$. This follows with the witness x' for y since $\mathbf{s}(x') =_N \mathbf{s}(x')$ by reflexivity of equality.

□

Here is the same proof in the linear notation for natural deductions.

```
[ x : nat;
  [ ~ 0 = 0;
    0 = 0;
    F;
    ?y:nat. s(y) = 0 ];
  ~ 0 = 0 => ?y:nat. s(y) = 0;    % case (x = 0)
  [ x' : nat, ~ x' = 0 => ?y:nat. s(y) = x';
    [ ~ s(x') = 0;
      !z:nat. z = z;              % reflexivity lemma
      s(x') : nat;
      s(x') = s(x');
      ?y:nat. s(y) = s(x') ];
    ~ s(x') = 0 => ?y:nat. s(y) = s(x') ];  % case (x = s(x'))
  ~ x = 0 => ?y:nat. s(y) = x ];
!x:nat. ~ x = 0 => ?y:nat. s(y) = x;
```

Next we give the equational specification of the function $pred'$ corresponding to this proof. Note that the function takes two arguments: x and a proof u of $\neg x =_{\mathbf{N}} \mathbf{0}$. It returns a pair consisting of a witness n and proof that $\mathbf{s}(n) =_{\mathbf{N}} x$.

$$\begin{array}{lrcl} pred' & \mathbf{0} \quad u & = & \mathbf{abort}(u\ \mathbf{eq}_0) \\ pred' & (\mathbf{s}(x')) \quad u & = & \langle x', refl\,(\mathbf{s}(x')) \end{array}$$

Note that in the case of $x = \mathbf{0}$ we do not explicitly construct a pair, but abort the computation, which may have any type. This specification can be written as a program rather directly.

$$\begin{array}{lcl} pred' & : & \forall x{\in}\mathbf{nat}.\ \neg x =_{\mathbf{N}} \mathbf{0} \supset \exists y{\in}\mathbf{nat}.\ \mathbf{s}(y) =_{\mathbf{N}} x \\ pred' & = & \lambda x{\in}\mathbf{nat}.\ \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow (\lambda u.\ \mathbf{abort}\,(u\ \mathbf{eq}_0)) \\ & & \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow (\lambda u.\ \langle x', refl(\mathbf{s}(x'))\rangle) \end{array}$$

If we erase the parts of this term that are concerned purely with propositions and leave only data types we obtain

$$\begin{array}{lcl} pred' & = & \lambda x{\in}\mathbf{nat}.\ \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \_\_ \\ & & \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow x' \end{array}$$

which is close to our earlier implementation of the predecessor. Erasing the **abort** clause from the impossible case has left a hole, which we denoted by $\_\_$. We return to a more detailed specification of this erasure process in the next section.

First, we discuss and alternative way to connect arithmetic to functional programming. This is to write the program first and prove its properties. Recall the definition of *pred*:

$$\begin{array}{lcl} pred & = & \lambda x{\in}\mathbf{nat}.\ \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{0} \\ & & \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow x' \end{array}$$

Now we can prove that

$$\forall x \in \mathbf{nat}.\ \neg x =_N \mathbf{0} \supset \mathbf{s}(\mathit{pred}(x)) =_N x$$

Proof: The proof is by cases over x.

Case: $x = \mathbf{0}$. Then $\neg\mathbf{0} =_N \mathbf{0}$ is contradictory.

Case: $x = \mathbf{s}(x')$. Then

$$\begin{array}{l}\mathbf{s}(\mathit{pred}(\mathbf{s}(x'))) \\ \quad\Longrightarrow \mathbf{s}(\mathbf{rec}\ \mathbf{s}(x') \\ \qquad\qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{0} \\ \qquad\qquad\quad \mid\ f(\mathbf{s}(x')) \Rightarrow x') \\ \quad\Longrightarrow \mathbf{s}(x')\end{array}$$

and $\mathbf{s}(x') =_N \mathbf{s}(x')$ by reflexivity.

□

This shows that we must be able to use the rules of computation when reasoning about functions. This is not a property particular to natural numbers, but we might have to reason about functions at arbitrary types or proofs of arbitrary propositions. Reduction therefore has to be an integral part of the type theory. We will use two rules of the form

$$\frac{\Gamma \vdash M : A \qquad A \Longrightarrow A' \qquad \Gamma \vdash A\ \mathit{prop}}{\Gamma \vdash M : A'}\ \mathit{conv}$$

$$\frac{\Gamma \vdash M : A' \qquad A \Longrightarrow A' \qquad \Gamma \vdash A\ \mathit{prop}}{\Gamma \vdash M : A}\ \mathit{conv}'$$

where $A \Longrightarrow A'$ allows the reduction of a term occurring in A. A unified form of this rule where $A \Longleftrightarrow A'$ allows an arbitrary number of reduction and expansion steps in both directions is called *type conversion*. While reduction generally preserves well-formedness (see Theorem 3.1), the converse does not. Generally, this is implied either from the premise or the conclusion, depending on whether we are reasoning backward or forward. Note that the conversion rules are "silent" in that the proof term M does not change.

In the formal proof, computations are omitted. They are carried out implicitly by the type checker. The question whether the resulting checking problem is decidable varies, depending on the underlying notion of computation. We return to this question when discussing the operational semantics in Section **??**.

We close this section with the formal version of the proof above. Note the use of the conversion rule *conv′*.

```
[ x : nat;
  [ ~ 0 = 0; 0 = 0; F;
    s(pred(0)) = 0 ];
  ~ 0 = 0 => s(pred(0)) = 0;    % case (x = 0)
  [ x' : nat, ~ x' = 0 => s(pred(x')) = x';
    [ ~ s(x') = 0;
      !z:nat. z = z;            % reflexivity lemma
      s(x') : nat;
      s(pred(s(x'))) = s(x') ]; % since pred(s(x')) ==> x'
    ~ s(x') = 0 => s(pred(s(x'))) = s(x') ];  % case (x = s(x'))
  ~ x = 0 => s(pred(x)) = x ];
!x:nat. ~ x = 0 => s(pred(x)) = x
```

4.4 Contracting Proofs to Programs

In this section we return to an early idea behind the computational interpretation of constructive logic: a proof of $\forall x{\in}\tau.\ \exists y{\in}\sigma.\ A(x,y)$ should describe a function f from elements of type τ to elements of type σ such that $A(x,f(x))$ is true for all x. The proof terms for intuitionistic logic and arithmetic do not quite fill this role. This is because if M is a proof term for $\forall x{\in}\tau.\ \exists y{\in}\sigma.\ A(x,y)$, then it describes a function that returns not only an appropriate term t, but also a proof term that certifies $A(x,t)$.

Thus we would like to contract proofs to programs, ignoring those parts of a proof term that are not of interest. Of course, what is and what is not of interest depends on the application. To illustrate this point and the process of erasing parts of a proof term, we consider the example of even and odd numbers. We define the addition function (in slight variation to the definition in Section 3.5) and the predicates *even* and *odd*.

$$\begin{array}{rcl}
\mathit{plus} & : & \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat} \\
\mathit{plus} & = & \lambda x{\in}\mathbf{nat}.\ \mathbf{rec}\ x \\
 & & \qquad \mathbf{of}\ p(\mathbf{0}) \Rightarrow \lambda y.\ y \\
 & & \qquad\ \mid p(\mathbf{s}(x')) \Rightarrow \lambda y.\ \mathbf{s}(p(x')\,y) \\
\mathit{even}(x) & = & \exists y{\in}\mathbf{nat}.\ \mathit{plus}\,y\,y =_N x \\
\mathit{odd}(x) & = & \exists y{\in}\mathbf{nat}.\ \mathbf{s}(\mathit{plus}\,y\,y) =_N x
\end{array}$$

For the rest of this section, we will use the more familiar notation $m+n$ for $\mathit{plus}\,m\,n$.

We can now prove that every natural number is either even or odd. First, the informal proof. For this we need a lemma (whose proof is left to the reader)

$$\mathit{lmps} \quad : \quad \forall x{\in}\mathbf{nat}.\ \forall y{\in}\mathbf{nat}.\ x + \mathbf{s}(y) =_N \mathbf{s}(x+y)$$

Now back to the main theorem.

$$\forall x{\in}\mathbf{nat}.\ (\exists y{\in}\mathbf{nat}.\ y + y =_N x) \vee (\exists y{\in}\mathbf{nat}.\ \mathbf{s}(y + y) =_N x)$$

Proof: The proof is by induction on x.

Case: $x = \mathbf{0}$. Then x is even, because $\mathbf{0} + \mathbf{0} =_N \mathbf{0}$ by computation and $=_N I_0$. The computation needed here is $\mathbf{0} + \mathbf{0} \Longrightarrow \mathbf{0}$

Case: $x = \mathbf{s}(x')$. By induction hypothesis we know that x' is either even or odd. We distinguish these two subcases.

Subcase: x' is even, that is, $\exists y{\in}\mathbf{nat}.\ y + y =_N x'$. Let's call this element c. Then $c{+}c =_N x'$ and hence $\mathbf{s}(c{+}c) =_N \mathbf{s}(x')$ by rule $=_N I_s$. Therefore $\exists y{\in}\mathbf{nat}.\ \mathbf{s}(y + y) =_N \mathbf{s}(x')$ and x is odd.

Subcase: x' is odd, that is, $\exists y{\in}\mathbf{nat}.\ \mathbf{s}(y + y) =_N x'$. Let's call this element d. Then $\mathbf{s}(d + d) =_N x'$ and $\mathbf{s}(\mathbf{s}(d + d)) =_N \mathbf{s}(x')$ by rule $=_N I_s$. Now, we compute $\mathbf{s}(\mathbf{s}(d + d)) \Longrightarrow \mathbf{s}(\mathbf{s}(d) + d)$ and apply lemma *lmps* to conclude $\mathbf{s}(d){+}\mathbf{s}(d) =_N \mathbf{s}(\mathbf{s}(d{+}d))$. By transitivity, therefore, $\mathbf{s}(d){+}\mathbf{s}(d) =_N \mathbf{s}(x')$. Therefore $\exists y{\in}\mathbf{nat}.\ y + y =_N \mathbf{s}(x')$ and x is even.

□

The proof term corresponding to this informal proof is mostly straightforward.

$$ev\ :\ \forall x{\in}\mathbf{nat}.\ (\exists y{\in}\mathbf{nat}.\ y + y =_N x) \vee (\exists y{\in}\mathbf{nat}.\ \mathbf{s}(y + y) =_N x)$$

$$\begin{array}{l} ev = \lambda x.\ \mathbf{rec}\ x \\ \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{inl}\langle \mathbf{0}, \mathbf{eq}_{\mathbf{0}}\rangle \\ \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow \mathbf{case}\ f(x') \\ \qquad\qquad\qquad \mathbf{of}\ \mathbf{inl}(u) \Rightarrow \mathbf{let}\ \langle c, p\rangle = u\ \mathbf{in}\ \mathbf{inr}\langle c, \mathbf{eq}_{\mathbf{s}}(p)\rangle \\ \qquad\qquad\qquad\ \mid\ \mathbf{inr}(w) \Rightarrow \mathbf{let}\ \langle d, q\rangle = w\ \mathbf{in}\ \mathbf{inl}\langle \mathbf{s}(d), r(x', d, q)\rangle \end{array}$$

Here, $r(x', d, q)$ is a proof term verifying that $\mathbf{s}(d) + \mathbf{s}(d) =_N \mathbf{s}(x')$. It uses transitivity of equality and the lemma *lmps*. Its precise form is not important for the discussion in this section—we give it here only for completeness.

$$\begin{array}{lcl} r(x', d, q) & : & \mathbf{s}(d) + \mathbf{s}(d) =_N \mathbf{s}(x') \\ r(x', d, q) & = & \mathit{trans}\ (\mathbf{s}(d) + \mathbf{s}(d))\ (\mathbf{s}(\mathbf{s}(d) + d))\ (\mathbf{s}(x'))\ (\mathit{lmps}\ (\mathbf{s}(d))\ d)\ (\mathbf{eq}_{\mathbf{s}}\ q) \end{array}$$

Next we consider various versions of this specification and its implementation, erasing "uninteresting" subterms. For the first version, we would like to obtain the witnesses $y{\in}\mathbf{nat}$ in each case, but we do not want to carry the proof that $y + y =_N x$. We indicate this by bracketing the corresponding part of the proposition.

$$ev_1\ :\ \forall x{\in}\mathbf{nat}.\ (\exists y{\in}\mathbf{nat}.\ [y + y =_N x]) \vee (\exists y{\in}\mathbf{nat}.\ [\mathbf{s}(y + y) =_N x])$$

We then bracket the corresponding parts of the proof term. Roughly, every subterm whose type has the form $[A]$ should be bracketed, including variables whose type has this form. The intent is that these subterms will be completely erased before the program is run. In the case of the annotation above we obtain:

$$
\begin{array}{l}
ev_1 \;:\; \forall x{\in}\mathbf{nat}.\, (\exists y{\in}\mathbf{nat}.\, [y + y =_N x]) \vee (\exists y{\in}\mathbf{nat}.\, [\mathbf{s}(y + y) =_N x]) \\
ev_1 = \lambda x.\, \mathbf{rec}\ x \\
\qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{inl}\langle \mathbf{0}, [\mathbf{eq}_0]\rangle \\
\qquad\ \mid f(\mathbf{s}(x')) \Rightarrow \mathbf{case}\ f(x') \\
\qquad\qquad\qquad \mathbf{of}\ \mathbf{inl}(u) \Rightarrow \mathbf{let}\ \langle c, [p]\rangle = u\ \mathbf{in}\ \mathbf{inr}\langle c, [\mathbf{eq}_s(p)]\rangle \\
\qquad\qquad\qquad\ \mid \mathbf{inr}(w) \Rightarrow \mathbf{let}\ \langle d, [q]\rangle = w\ \mathbf{in}\ \mathbf{inl}\langle \mathbf{s}(d), [r(x', d, q)]\rangle
\end{array}
$$

Not every possible bracket annotation of a term is correct. A formal treatment of which bracket annotations are valid is beyond the scope of these notes. However, the main rule is easy to state informally:

Bracketed variables $[x]$ *may occur only inside brackets* $[\ldots x \ldots]$.

This is because bracketed variables are erased before execution of the program. Therefore, an occurrence of a bracketed variable in a term that is *not* erased would lead to a runtime error, since the corresponding value would not be available. We refer to variables of this form as *hidden variables*.

In the example above, $[p]$ and $[q]$ are the only hidden variables. Our restriction is satisfied: p occurs only in $[\mathbf{eq}_s(p)]$ and q only in $[r(x', d, q)]$.

The actual erasure can be seen as proceding in three steps. In the first step, we replace every bracketed proposition $[A]$ by $\top$ and every subterm $[M]$ by its proof term $\langle\,\rangle$. Furthermore, every bracketed variable $[u]$ is replaced by an anonymous variable $\_$, since this variable is not supposed to occur after erasure. We obtain:

$$
\begin{array}{l}
ev_1 \;:\; \forall x{\in}\mathbf{nat}.\, (\exists y{\in}\mathbf{nat}.\, \top) \vee (\exists y{\in}\mathbf{nat}.\, \top) \\
ev_1 = \lambda x.\, \mathbf{rec}\ x \\
\qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{inl}\langle \mathbf{0}, \langle\,\rangle\rangle \\
\qquad\ \mid f(\mathbf{s}(x')) \Rightarrow \mathbf{case}\ f(x') \\
\qquad\qquad\qquad \mathbf{of}\ \mathbf{inl}(u) \Rightarrow \mathbf{let}\ \langle c, \_\rangle = u\ \mathbf{in}\ \mathbf{inr}\langle c, \langle\,\rangle\rangle \\
\qquad\qquad\qquad\ \mid \mathbf{inr}(w) \Rightarrow \mathbf{let}\ \langle d, \_\rangle = w\ \mathbf{in}\ \mathbf{inl}\langle \mathbf{s}(d), \langle\,\rangle\rangle
\end{array}
$$

In the second step we perform simplifications to obtain a function purely operating on data types. For this we have to recall that, under the Curry-Howard isomorphism, for example $\top$ is interpreted as the unit type $\mathbf{1}$, and that disjunction $A \vee B$ is interpreted as a disjoint sum type $\tau + \sigma$.

What happens to universal and existential quantification? Recall that the proof term for $\forall x{\in}\mathbf{nat}.\, A(x)$ is a function which maps every natural number n to a proof term for $A(n)$. When we erase all proof terms, n cannot actually occur in the result of erasing $A(n)$ and the result has the form $\mathbf{nat} \to \tau$, where τ is the erasure of $A(x)$.

Similarly, a proof term for $\exists x{\in}\mathbf{nat}.\, A(x)$ consists of a pair $\langle n, M\rangle$, where n is a natural number (the witness) and M is a proof term for $A(n)$. When we

turn propositions into types by erase, we obtain $\mathbf{nat} \times \tau$, where τ is the erasure of $A(n)$.

Applying this translation operation to our proof, we obtain:

$$
\begin{array}{l}
ev_1 \ : \ \mathbf{nat} \to (\mathbf{nat} \times \mathbf{1}) + (\mathbf{nat} \times \mathbf{1}) \\
ev_1 = \lambda x.\, \mathbf{rec}\ x \\
\qquad\qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{inl}\langle \mathbf{0}, \langle\,\rangle \rangle \\
\qquad\qquad\ \mid\ f(\mathbf{s}(x')) \Rightarrow \mathbf{case}\ f(x') \\
\qquad\qquad\qquad\qquad\qquad \mathbf{of}\ \mathbf{inl}(u) \Rightarrow \mathbf{inr}\langle \mathbf{fst}(u), \langle\,\rangle \rangle \\
\qquad\qquad\qquad\qquad\qquad\ \mid\ \mathbf{inr}(w) \Rightarrow \mathbf{inl}\langle \mathbf{s}(\mathbf{fst}(w)), \langle\,\rangle \rangle
\end{array}
$$

Note that our proof term changes in only two places, because the elimination rules for existentials and pairs do not match up. In all other cases we have overloaded our notation, precisely in anticipation of this correspondence. For existentials, we replace

$$\mathbf{let}\ \langle x, u \rangle = M\ \mathbf{in}\ N$$

by

$$[\mathbf{fst}(M)/x][\mathbf{snd}(M)/u]N$$

Finally, we apply some optimizations by eliminating unnecessary constructions involving the unit type. We take advantage of isomorphisms such as

$$
\begin{array}{rcl}
\tau \times \mathbf{1} & \mapsto & \tau \\
\mathbf{1} \times \tau & \mapsto & \tau \\
\mathbf{1} \to \tau & \mapsto & \tau \\
\tau \to \mathbf{1} & \mapsto & \mathbf{1}
\end{array}
$$

Note that $\mathbf{1} + \mathbf{1}$ can *not* be simplified: it is a type with two elements, $\mathbf{inl}\,\langle\,\rangle$ and $\mathbf{inr}\,\langle\,\rangle$.

An optimization in a type must go along with a corresponding optimization in a term so that it remains well-typed. This is accomplished by the following simplification rules.

$$
\begin{array}{lrcl}
 & \langle t, s \rangle \in \tau \times \mathbf{1} & \mapsto & t \\
\text{for } t \in \tau \times \mathbf{1}, & \mathbf{fst}(t) \in \tau & \mapsto & t \\
\text{for } t \in \tau \times \mathbf{1}, & \mathbf{snd}(t) \in \mathbf{1} & \mapsto & \langle\,\rangle \\
 & \langle s, t \rangle \in \mathbf{1} \times \tau & \mapsto & t \\
\text{for } t \in \mathbf{1} \times \tau, & \mathbf{snd}(t) \in \tau & \mapsto & t \\
\text{for } t \in \mathbf{1} \times \tau, & \mathbf{fst}(t) \in \mathbf{1} & \mapsto & \langle\,\rangle \\
 & (\lambda x{\in}\mathbf{1}.\, t) \in \mathbf{1} \to \tau & \mapsto & t \\
\text{for } t \in \mathbf{1} \to \tau, & t\, s \in \tau & \mapsto & t \\
 & (\lambda x{\in}\tau.\, t) \in \tau \to \mathbf{1} & \mapsto & \langle\,\rangle \\
\text{for } t \in \tau \to \mathbf{1}, & t\, s \in \mathbf{1} & \mapsto & \langle\,\rangle
\end{array}
$$

When we apply these transformation to our running example, we obtain

$$
\begin{array}{l}
ev_1 \;:\; \mathbf{nat} \rightarrow (\mathbf{nat} + \mathbf{nat}) \\
ev_1 = \lambda x.\, \mathbf{rec}\ x \\
\qquad\qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{inl}(\mathbf{0}) \\
\qquad\qquad\ \mid f(\mathbf{s}(x')) \Rightarrow \mathbf{case}\ f(x') \\
\qquad\qquad\qquad\qquad\qquad \mathbf{of}\ \mathbf{inl}(u) \Rightarrow \mathbf{inr}(u) \\
\qquad\qquad\qquad\qquad\qquad\ \mid \mathbf{inr}(w) \Rightarrow \mathbf{inl}(\mathbf{s}(w))
\end{array}
$$

We can see that this function satisfies the following specification:

$$
\begin{array}{llcll}
ev_1 & n & = & \mathbf{inl}(n/2) & \text{if } n \text{ is even} \\
ev_1 & n & = & \mathbf{inr}((n-1)/2) & \text{if } n \text{ is odd}
\end{array}
$$

So $ev_1(n)$ returns the floor of $n/2$, plus a tag which tells us if n was even or odd.

The whole process by which we arrived at this function, starting from the bracket annotation of the original specification can be done automatically by a compiler. A similar process is used, for example, in the Coq system to extract efficient ML functions from constructive proofs in type theory.

Returning to the original specification, assume we want to return only an indication whether the argument is even or odd, but not the result of dividing it by two. In that case, we bracket both existential quantifiers, in effect erasing the witness in addition to the proof term.

$$
\begin{array}{l}
ev_2 \;:\; \forall x{\in}\mathbf{nat}.\, [\exists y{\in}\mathbf{nat}.\, y + y =_N x] \vee [\exists y{\in}\mathbf{nat}.\, \mathbf{s}(y + y) =_N x] \\
ev_2 = \lambda x.\, \mathbf{rec}\ x \\
\qquad\qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{inl}[\langle \mathbf{0}, \mathbf{eq}_{\mathbf{0}} \rangle] \\
\qquad\qquad\ \mid f(\mathbf{s}(x')) \Rightarrow \mathbf{case}\ f(x') \\
\qquad\qquad\qquad\qquad\qquad \mathbf{of}\ \mathbf{inl}[u] \Rightarrow \mathbf{let}\, [\langle c, p \rangle = u]\ \mathbf{in}\ \mathbf{inr}[\langle c, \mathbf{eq}_s(p) \rangle] \\
\qquad\qquad\qquad\qquad\qquad\ \mid \mathbf{inr}[w] \Rightarrow \mathbf{let}\, [\langle d, q \rangle = w]\ \mathbf{in}\ \mathbf{inl}[\langle \mathbf{s}(d), r(x', d, q) \rangle]
\end{array}
$$

Fortunately, our restriction is once again satisfied: bracketed variables (this time, u, c, p, w, d, q) appear only within brackets. The occurrences of c and p in the **let**-expression should be considered bracketed, because u and therefore c and p will not be carried when the program is executed. A similar remark applies to w, d. We now skip several steps, which the reader may want to reconstruct, to arrive at

$$
\begin{array}{l}
ev_2 \;:\; \mathbf{nat} \rightarrow (\mathbf{1} + \mathbf{1}) \\
ev_2 = \lambda x.\, \mathbf{rec}\ x \\
\qquad\qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{inl}\, \langle\,\rangle \\
\qquad\qquad\ \mid f(\mathbf{s}(x')) \Rightarrow \mathbf{case}\ f(x') \\
\qquad\qquad\qquad\qquad\qquad \mathbf{of}\ \mathbf{inl}\_ \Rightarrow \mathbf{inr}\, \langle\,\rangle \\
\qquad\qquad\qquad\qquad\qquad\ \mid \mathbf{inr}\_ \Rightarrow \mathbf{inl}\, \langle\,\rangle
\end{array}
$$

Note that this function simply alternates between $\mathbf{inl}\,\langle\,\rangle$ and $\mathbf{inr}\,\langle\,\rangle$ in each recursive call, thereby keeping track if the number is even or odd. It satisfies

$$\begin{array}{lcll} ev_2 & n & = & \mathbf{inl}\,\langle\,\rangle \quad \text{if } n \text{ is even} \\ ev_2 & n & = & \mathbf{inr}\,\langle\,\rangle \quad \text{if } n \text{ is odd} \end{array}$$

As a third modification, assume we intend to apply ev to even numbers n to obtain $n/2$; if n is odd, we just want an indication that it was not even. The annotation of the type is straightforward.

$$ev_3 : \forall x{\in}\mathbf{nat}.\ (\exists y{\in}\mathbf{nat}.\ [y + y =_N x]) \vee [\exists y{\in}\mathbf{nat}.\ \mathbf{s}(y + y) =_N x]$$

Applying our annotation algorithm to the proof term leads to the following.

$$\begin{array}{l} ev_3 = \lambda x.\ \mathbf{rec}\ x \\ \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{inl}\langle \mathbf{0}, [\mathbf{eq}_0]\rangle \\ \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow \mathbf{case}\ f(x') \\ \qquad\qquad\qquad \mathbf{of}\ \mathbf{inl}(u) \Rightarrow \mathbf{let}\,\langle c, [p]\rangle = u\ \mathbf{in}\ \mathbf{inr}[\langle c, \mathbf{eq}_s(p)\rangle] \\ \qquad\qquad\qquad\ \mid \mathbf{inr}[w] \Rightarrow \mathbf{let}\,[\langle d, q\rangle = w]\ \mathbf{in}\ \mathbf{inl}\langle \mathbf{s}(d), [r(x', d, q)]\rangle \end{array}$$

But this version of ev does not satisfy our restriction: in the last line, the hidden variable $[d]$ occurs outside of brackets. Indeed, if we apply our technique of erasing computationally irrelevant subterms we obtain

$$\begin{array}{l} ev_3\ :\ \mathbf{nat} \rightarrow (\mathbf{nat} + \mathbf{1}) \\ ev_3 = \lambda x.\ \mathbf{rec}\ x \\ \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{inl}(\mathbf{0}) \\ \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow \mathbf{case}\ f(x') \\ \qquad\qquad\qquad \mathbf{of}\ \mathbf{inl}(u) \Rightarrow \mathbf{inr}\langle\,\rangle \\ \qquad\qquad\qquad\ \mid \mathbf{inr}(\_) \Rightarrow \mathbf{inl}(\mathbf{s}(d)) \end{array}$$

where d is required, but not generated by the recursive call. Intuitively, the information flow in the program is such that, in order to compute $n/2$ for even n, we *must* compute $(n-1)/2$ for odd n.

The particular proof we had did not allow the particular bracket annotation we proposed. However, we can give a different proof, which permits this annotation. In this example, it is easier to just write the function with the desired specification directly, using the function ev_1 which preserved the information for the case of an odd number.

$$\begin{array}{l} ev_3\ :\ \mathbf{nat} \rightarrow (\mathbf{nat} + \mathbf{1}) \\ ev_3\, n = \mathbf{inl}(n/2) \qquad\qquad \text{if } n \text{ is even} \\ ev_3\, n = \mathbf{inr}\,\langle\,\rangle \qquad\qquad \text{if } n \text{ is odd} \end{array}$$

$$\begin{array}{l} ev_3 = \lambda x.\ \mathbf{case}\ ev_1(x) \\ \qquad\qquad \mathbf{of}\ \mathbf{inl}(c) \Rightarrow \mathbf{inl}(c) \\ \qquad\qquad\ \mid \mathbf{inr}(d) \Rightarrow \mathbf{inr}\,\langle\,\rangle \end{array}$$

To complete this section, we return to our example of the predecessor specification and proof.

$$\begin{array}{lcl} \mathit{pred}' & : & \forall x{\in}\mathbf{nat}.\, \neg x =_N \mathbf{0} \supset \exists y{\in}\mathbf{nat}.\, \mathbf{s}(y) =_N x \\ \mathit{pred}' & = & \lambda x{\in}\mathbf{nat}.\, \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow (\lambda u.\, \mathbf{abort}\,(u\ \mathbf{eq}_0)) \\ & & \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow (\lambda u.\, \langle x', \mathit{refl}(\mathbf{s}(x'))\rangle) \end{array}$$

If we hide all proof objects we obtain:

$$\begin{array}{lcl} \mathit{pred}' & : & \forall x{\in}\mathbf{nat}.\, [\neg x =_N \mathbf{0}] \supset \exists y{\in}\mathbf{nat}.\, [\mathbf{s}(y) =_N x] \\ \mathit{pred}' & = & \lambda x{\in}\mathbf{nat}.\, \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow (\lambda [u].\, \mathbf{abort}\,(u\ \mathbf{eq}_0)) \\ & & \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow (\lambda [u].\, \langle x', [\mathit{refl}(\mathbf{s}(x'))]\rangle) \end{array}$$

Note that this function does *not* satisfy our restriction: the hidden variable u occurs outside a bracket in the case for $f(\mathbf{0})$. This is because we cannot bracket any subterm of

$$\mathbf{abort}\,(u\ \mathbf{eq}_0) : \exists y{\in}\mathbf{nat}.\, [\mathbf{s}(y) =_N \mathbf{0}]$$

We conclude that our proof of pred' does not lend itself to the particular given annotation. However, we can give a different proof where we supply an arbitrary witness c for y in case x is $\mathbf{0}$ and prove that it satisfies $\mathbf{s}(y) =_N \mathbf{0}$ by $\bot E$ as before. We chose $c = \mathbf{0}$.

$$\begin{array}{lcl} \mathit{pred}' & : & \forall x{\in}\mathbf{nat}.\, \neg x =_N \mathbf{0} \supset \exists y{\in}\mathbf{nat}.\, \mathbf{s}(y) =_N x \\ \mathit{pred}' & = & \lambda x{\in}\mathbf{nat}.\, \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow (\lambda u.\, \langle \mathbf{0}, \mathbf{abort}\,(u\ \mathbf{eq}_0)\rangle) \\ & & \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow (\lambda u.\, \langle x', \mathit{refl}(\mathbf{s}(x'))\rangle) \end{array}$$

Now annotation and extraction succeeds, yielding *pred*. Of course, any natural number would do for the result of $\mathit{pred}(\mathbf{0})$

$$\begin{array}{lcl} \mathit{pred}_2' & : & \forall x{\in}\mathbf{nat}.\, [\neg x =_N \mathbf{0}] \supset \exists y{\in}\mathbf{nat}.\, [\mathbf{s}(y) =_N x] \\ \mathit{pred}_2' & = & \lambda x{\in}\mathbf{nat}.\, \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow (\lambda [u].\, \langle \mathbf{0}, [\mathbf{abort}\,(u\ \mathbf{eq}_0)\rangle]) \\ & & \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow (\lambda [u].\, \langle x', [\mathit{refl}(\mathbf{s}(x'))]\rangle) \\ \mathit{pred} & : & \mathbf{nat} \to \mathbf{nat} \\ \mathit{pred} & = & \lambda x{\in}\mathbf{nat}.\, \mathbf{rec}\ x \\ & & \qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{0} \\ & & \qquad\ \mid f(\mathbf{s}(x')) \Rightarrow x' \end{array}$$

The reader may test his understanding of the erasure process by transforming pred_2' from above step by step into *pred*. It requires some of the simplifications on function types.

4.5 Structural Induction

We now leave arithmetic, that is, the theory of natural numbers, and discuss more general data types. We first return to lists, whose elements are drawn

from arbitray types. The reader may wish to remind himself of the basic computation constructs given in Section 3.7. We recall here only that there are two introduction rules for lists:

$$\frac{}{\Gamma \vdash \mathbf{nil}^{\tau} \in \tau\,\mathbf{list}}\,\mathbf{list}I_n \qquad \frac{\Gamma \vdash h \in \tau \qquad \Gamma \vdash t \in \tau\,\mathbf{list}}{\Gamma \vdash h :: t \in \tau\,\mathbf{list}}\,\mathbf{list}I_c$$

In the induction principle, correspondingly, we have to account for two cases. We first state it informally.

To prove $A(l)$ true for an arbitrary list l, prove

1. *$A(\mathbf{nil})$ true and*
2. *$A(x :: l')$ true for an arbitrary x and l', under the assumption $A(l')$ true.*

The first is the base case, the second the induction step. When we write this as a formal inference rules, we obtain the analogue of primitive recursion.

$$\frac{\Gamma \vdash l \in \tau\,\mathbf{list} \quad \Gamma \vdash A(\mathbf{nil})\ true \quad \Gamma, x{\in}\tau, l'{\in}\tau\,\mathbf{list}, A(l')\ true \vdash A(x :: l')\ true}{\Gamma \vdash A(l)\ true}\,\mathbf{list}E$$

This principle is called *structural induction over lists*. Our first theorem about lists will be a simple property of the append function. In order to formulate this property, we need equality over lists. It is defined in analogy with the propositional equality between natural numbers, based on the structure of lists.

$$\frac{\Gamma \vdash l \in \tau\,\mathbf{list} \qquad \Gamma \vdash k \in \tau\,\mathbf{list}}{\Gamma \vdash l =_L k\ prop} =_L F$$

$$\frac{}{\Gamma \vdash \mathbf{nil} =_L \mathbf{nil}\ true} =_L I_n \qquad \frac{\Gamma \vdash l =_L k\ true}{\Gamma \vdash x :: l =_L x :: k\ true} =_L I_c$$

The second introduction rules requires the heads of the two lists to be identical. We can not require them to be equal, because they are of unknown type τ and we do not have a generic equality proposition that works for arbitrary types. However, in this section, we are interested in proving generic properties of lists, rather than, say, properties of integer lists. For this purpose, the introduction rule above, and the three elimination rules below are sufficient.

$$\frac{\Gamma \vdash x :: l =_L y :: k\ true}{\Gamma \vdash l =_L k\ true} =_L E_{cc}$$

$$\frac{\Gamma \vdash \mathbf{nil} =_L y :: k\ true}{\Gamma \vdash C\ true} =_L E_{nc} \qquad \frac{\Gamma \vdash x :: l =_L \mathbf{nil}\ true}{\Gamma \vdash C\ true} =_L E_{cn}$$

Note that the first elimination rule is incomplete in the sense that we also know that x must be identical to y, but we cannot obtain this information by the rule. A solution to this problem is beyond the scope of these notes.

It is straightforward to show that equality is reflexive, symmetric and transitive, and we will use these properties freely below.

Next we give a definition of a function to append two lists which is a slightly modified version from that in Section 3.7.

$$\begin{array}{llcl} \mathit{app} & \mathbf{nil} \quad k & = & k \\ \mathit{app} & (x :: l') \quad k & = & x :: (\mathit{append}\, l'\, k) \end{array}$$

In the notation of primitive recursion:

$$\begin{array}{lcl} \mathit{app} & \in & \tau\,\mathbf{list} \to \tau\,\mathbf{list} \to \tau\,\mathbf{list} \\ \mathit{app} & = & \lambda l.\ \mathbf{rec}\ l \\ & & \quad \mathbf{of}\ f(\mathbf{nil}) \Rightarrow \lambda k.\ k \\ & & \quad\ \mid f(x :: l') \Rightarrow \lambda k.\ x :: (f(l')\, k) \end{array}$$

We now prove

$$\forall l{\in}\tau\,\mathbf{list}.\ \mathit{app}\ l\ \mathbf{nil} =_L l$$

Proof: By induction on the structure of l.

Case: $l = \mathbf{nil}$. Then $\mathit{app}\ \mathbf{nil}\ \mathbf{nil} =_L \mathbf{nil}$ since

$$\begin{array}{l} \mathit{app}\ \mathbf{nil}\ \mathbf{nil} \\ \Longrightarrow (\mathbf{rec}\ \mathbf{nil} \\ \quad \mathbf{of}\ f(\mathbf{nil}) \Rightarrow \lambda k.\ k \\ \quad\ \mid f(x :: l') \Rightarrow \lambda k.\ x :: f(l')\, k)\, \mathbf{nil} \\ \Longrightarrow (\lambda k.\ k)\, \mathbf{nil} \\ \Longrightarrow \mathbf{nil} \end{array}$$

Case: $l = x :: l'$. Then $\mathit{app}\ l'\ \mathbf{nil} =_L l'$ by induction hypothesis. Therefore

$$x :: (\mathit{app}\ l'\ \mathbf{nil}) =_L x :: l'$$

by rule $=_L I_c$. We have to show

$$\mathit{app}\ (x :: l')\ \mathbf{nil} =_L x :: l'.$$

This follows entirely by computation. Starting from the term in the conclusion:

$$\begin{array}{l} \mathit{app}\ (x :: l')\ \mathbf{nil} \\ \Longrightarrow (\mathbf{rec}\ x :: l' \\ \quad \mathbf{of}\ f(\mathbf{nil}) \Rightarrow \lambda k.\ k \\ \quad\ \mid f(x :: l') \Rightarrow \lambda k.\ x :: f(l')\, k)\, \mathbf{nil} \\ \Longrightarrow (\lambda k.\ x :: (\mathbf{rec}\ l' \\ \quad\quad \mathbf{of}\ f(\mathbf{nil}) \Rightarrow \lambda k.\ k \\ \quad\quad\ \mid f(x :: l') \Rightarrow \lambda k.\ x :: f(l')\, k)\, k)\, \mathbf{nil} \\ \Longrightarrow x :: (\mathbf{rec}\ l' \\ \quad\quad \mathbf{of}\ f(\mathbf{nil}) \Rightarrow \lambda k.\ k \\ \quad\quad\ \mid f(x :: l') \Rightarrow \lambda k.\ x :: f(l')\, k)\, \mathbf{nil} \end{array}$$

We arrive at the same term if we start from the induction hypothesis.

$$\begin{array}{l} x :: (\mathit{app}\ l'\ \mathbf{nil}) \\ \Longrightarrow x :: (\mathbf{rec}\ l' \\ \qquad\qquad \mathbf{of}\ f(\mathbf{nil}) \Rightarrow \lambda k.\ k \\ \qquad\qquad\ \mid\ f(x :: l') \Rightarrow \lambda k.\ x :: f(l')\ k)\ \mathbf{nil} \end{array}$$

Recall that computation is allowed in both directions (see Section 4.3), thereby closing the gap between the induction hypothesis and the conclusion. □

For the next theorem, we recall the specification of the reverse function on lists from Section 3.7, using an auxiliary function *rev* with an accumulator argument a.

$$\begin{array}{rcl} \mathit{rev} & \in & \tau\ \mathbf{list} \to \tau\ \mathbf{list} \to \tau\ \mathbf{list} \\ \mathit{rev}\ \mathbf{nil}\ a & = & a \\ \mathit{rev}\ (x :: l')\ a & = & \mathit{rev}\ l'\ (x :: a) \end{array}$$

$$\begin{array}{rcl} \mathit{reverse} & \in & \tau\ \mathbf{list} \to \tau\ \mathbf{list} \\ \mathit{reverse}\ l & = & \mathit{rev}\ l\ \mathbf{nil} \end{array}$$

The property we will prove is the interaction between *reverse* and *app*.

$$\forall l{\in}\tau\ \mathbf{list}.\ \forall k{\in}\tau\ \mathbf{list}.\ \mathit{reverse}\ (\mathit{app}\ l\ k) =_L \mathit{app}\ (\mathit{reverse}\ k)\ (\mathit{reverse}\ l)$$

Based on general heuristics, an induction on l is indicated, since it allows us to reduce in the left-hand side. However, such a proof attempt will fail. The reason is that *reverse* is not itself recursive, but defined in terms of *rev*. In such a situation, generalizing the induction hypothesis to express a corresponding property of the recursive function is almost always indicated.

It is often quite difficult to find an appropriate generalization of the induction hypothesis. It is useful to analyse the properties of *rev* in terms of *reverse* and *app*. We generalize from an example

$$\mathit{rev}\ (1 :: 2 :: 3 :: \mathbf{nil})\ (4 :: 5 :: \mathbf{nil}) \Longrightarrow 3 :: 2 :: 1 :: 4 :: 5 :: \mathbf{nil}$$

to conjecture that $\mathit{rev}\ l\ k =_L \mathit{app}\ (\mathit{reverse}\ l)\ k$ (omitting the quanitifers on l and k for the sake of brevity). We may or may not need this property, but it will help us to develop conjectures about the interaction between *rev* and *app*. Once again, the problem with this property is that the right-hand side mentions *reverse* and is not expressed in terms of *rev*. If we substitute the right-hand side will be

$$\mathit{rev}\ l\ k =_L \mathit{app}\ (\mathit{rev}\ l\ \mathbf{nil})\ k$$

Again this does not appear general enough, because of the occurrence of **nil**. If we replace this my a new term m, we also need to modify the left-hand side. The right generalization is suggested by our observation about the interaction of *reverse*, *app* and *rev*. We obtain

$$\forall l{\in}\tau\ \mathbf{list}.\ \forall m{\in}\tau\ \mathbf{list}.\ \forall k{\in}\tau\ \mathbf{list}.\ \mathit{rev}\ l\ (\mathit{app}\ m\ k) =_L \mathit{app}\ (\mathit{rev}\ l\ m)\ k$$

Now this can be proven by a straightforward structural induction over l. It most natural to pick l as the induction variable here, since this allows reduction on the right-hand side as well as the left-hand side. In general, it a good heuristic to pick variables that permit reduction when instantiated.

Proof: By structural induction on l.

Case: $l = \mathbf{nil}$. Then we get

$$\begin{array}{rl} \text{left-hand side:} & \mathit{rev}\ \mathbf{nil}\ (\mathit{app}\ m\ k) \Longrightarrow \mathit{app}\ m\ k \\ \text{right-hand side:} & \mathit{app}\ (\mathit{rev}\ \mathbf{nil}\ m)\ k \Longrightarrow \mathit{app}\ m\ k \end{array}$$

so the equality follows by computation and reflexivity of equality.

Case: $l = x{::}l'$. It is often useful to write out the general form of the induction hypothesis before starting the proof in the induction step.

$$\forall m{\in}\tau\ \mathbf{list}.\ \forall k{\in}\tau\ \mathbf{list}.\ \mathit{rev}\ l'\ (\mathit{app}\ m\ k) =_L \mathit{app}\ (\mathit{rev}\ l'\ m)\ k$$

As we will see, the quantifiers over m and k are critical here. Now we follow the general strategy to reduce the left-hand side and the right-hand side to see if we can close the gap by using the induction hypothesis.

$$\begin{array}{lll} \text{lhs:} & \mathit{rev}\ (x :: l')\ (\mathit{app}\ m\ k) & \\ & \Longrightarrow \mathit{rev}\ l'\ (x :: (\mathit{app}\ m\ k)) & \\ \text{rhs:} & \mathit{app}\ (\mathit{rev}\ (x :: l')\ m)\ k & \\ & \Longrightarrow \mathit{app}\ (\mathit{rev}\ l'\ (x :: m))\ k & \\ & =_L \mathit{rev}\ l'\ (\mathit{app}\ (x :: m)\ k) & \text{by ind. hyp} \\ & \Longrightarrow \mathit{rev}\ l'\ (x :: (\mathit{app}\ m\ k)) & \end{array}$$

So by computation and the induction hypothesis the left-hand side and the right-hand side are equal. Note that the universal quantifier on m in the induction hypothesis needed to be instantiated by $x :: m$. This is a frequent pattern when accumulator variables are involved.

□

Returning to our original question, we generalize the term on the left-hand side, $\mathit{reverse}\ (\mathit{app}\ l\ k)$, to $\mathit{rev}\ (\mathit{app}\ l\ k)\ m$. The appropriate generalization of the right-hand side yields

$$\forall l{\in}\tau\ \mathbf{list}.\ \forall k{\in}\tau\ \mathbf{list}.\ \forall m{\in}\tau\ \mathbf{list}.\ \mathit{rev}\ (\mathit{app}\ l\ k)\ m =_L \mathit{rev}\ k\ (\mathit{rev}\ l\ m)$$

In this general form we can easily prove it by induction over l.

Proof: By induction over l.

Case: $l = \mathbf{nil}$. Then

$$\begin{array}{ll} \text{lhs:} & \mathit{rev}\ (\mathit{app}\ \mathbf{nil}\ k)\ m \Longrightarrow \mathit{rev}\ k\ m \\ \text{rhs:} & \mathit{rev}\ k\ (\mathit{rev}\ \mathbf{nil}\ m) \Longrightarrow \mathit{rev}\ k\ m \end{array}$$

So the left- and right-hand side are equal by computation.

Case: $l = x :: l'$. Again, we write out the induction hypothesis:

$$\forall k{\in}\tau\ \mathbf{list}.\ \forall m{\in}\tau\ \mathbf{list}.\ \forall \mathit{rev}\ (\mathit{app}\ l'\ k)\ m =_L \mathit{rev}\ k\ (\mathit{rev}\ l'\ m)$$

Then

$$\begin{array}{ll} \text{lhs} & \mathit{rev}\ (\mathit{app}\ (x :: l')\ k)\ m \\ & \Longrightarrow \mathit{rev}\ (x :: (\mathit{app}\ l'\ k))\ m \\ & \Longrightarrow \mathit{rev}\ (\mathit{app}\ l'\ k)\ (x :: m) \\ \text{rhs} & \mathit{rev}\ k\ (\mathit{rev}\ (x :: l')\ m) \\ & \Longrightarrow \mathit{rev}\ k\ (\mathit{rev}\ l'\ (x :: m)) \end{array}$$

So the left- and right-hand sides are equal by computation and the induction hypothesis. Again, we needed to use $x :: m$ for m in the induction hypothesis.

□

By using these two properties together we can now show that this implies the original theorem directly.

$$\forall l{\in}\tau\ \mathbf{list}.\ \forall k{\in}\tau\ \mathbf{list}.\ \mathit{reverse}\ (\mathit{app}\ l\ k) =_L \mathit{app}\ (\mathit{reverse}\ k)\ (\mathit{reverse}\ l)$$

Proof: Direct, by computation and previous lemmas.

$$\begin{array}{lll} \text{lhs} & \mathit{reverse}\ (\mathit{app}\ l\ k) & \\ & \Longrightarrow \mathit{rev}\ (\mathit{app}\ l\ k)\ \mathbf{nil} & \\ & =_L \mathit{rev}\ k\ (\mathit{rev}\ l\ \mathbf{nil}) & \text{by lemma} \\ \text{rhs} & \mathit{app}\ (\mathit{reverse}\ k)\ (\mathit{reverse}\ l) & \\ & \Longrightarrow \mathit{app}\ (\mathit{rev}\ k\ \mathbf{nil})\ (\mathit{rev}\ l\ \mathbf{nil}) & \\ & =_L \mathit{rev}\ k\ (\mathit{app}\ \mathbf{nil}\ (\mathit{rev}\ l\ \mathbf{nil})) & \text{by lemma} \\ & =_L \mathit{rev}\ k\ (\mathit{rev}\ l\ \mathbf{nil}) & \end{array}$$

So the left- and right-hand sides are equal by computation and the two preceding lemmas. □

4.6 Reasoning about Data Representations

So far, our data types have been "freely generated" from a set of constructors. Equality on such types is structural. This has been true for natural numbers, lists, and booleans. In practice, there are many data representation which does not have this property. In this section we will examine two examples of this form.

The first is a representation of natural numbers in *binary* form, that is, as bit string consisting of zeroes and ones. This representation is of course prevalent in hardware and also much more compact than the unary numbers we have considered so far. The length of the representation of n is logarithmic in n. Thus, almost all work both on practical arithmetic and complexity theory uses binary representations. The main reason to consider unary representations in our context is the induction principle, and the connection between induction and primitive recursion.

We define the binary numbers with three constructors. We have the empty bit string ϵ, the operation of appending a $\mathbf{0}$ at the end, and the operation of appending a $\mathbf{1}$ at the end. We write the latter two in postfix notation, following the usual presentation of numbers as sequences of bits.

$$\frac{}{\Gamma \vdash \mathbf{bin}\ \mathit{type}}\,\mathbf{bin}F$$

$$\frac{}{\Gamma \vdash \epsilon \in \mathbf{bin}}\,\mathbf{bin}I_\epsilon \qquad \frac{\Gamma \vdash b \in \mathbf{bin}}{\Gamma \vdash b\,\mathbf{0} \in \mathbf{bin}}\,\mathbf{bin}I_0 \qquad \frac{\Gamma \vdash b \in \mathbf{bin}}{\Gamma \vdash b\,\mathbf{1} \in \mathbf{bin}}\,\mathbf{bin}I_1$$

The schema of primitive recursion has the following form

$$\begin{array}{lrcl} f & \epsilon & = & t_\epsilon \\ f & (b\,\mathbf{0}) & = & t_0(b, f(b)) \\ f & (b\,\mathbf{1}) & = & t_1(b, f(b)) \end{array}$$

Note that f, the recursive function, can occur only applied to b in the last two cases and not at all in the first case. It should be clear by now how to formulate the corresponding **rec** term and proof term construct. The induction principle is also straightforward.

To prove $A(b)$ true for an arbitrary bit string b, prove

Base Case: *$A(\epsilon)$ true.*

Step Case 0: *$A(b'\,\mathbf{0})$ true assuming $A(b')$ true for an arbitrary b'.*

Step Case 1: *$A(b'\,\mathbf{1})$ true assuming $A(b')$ true for an arbitrary b'.*

In order to describe formally how bitstring represent natural numbers, recall the function on natural numbers doubling its argument, specified as follows:

$$\begin{array}{rrcl} & \mathit{double} & \in & \mathbf{nat} \to \mathbf{nat} \\ \mathit{double} & \mathbf{0} & = & \mathbf{0} \\ \mathit{double} & (\mathbf{s}(x)) & = & \mathbf{s}(\mathbf{s}(\mathit{double}\ x)) \end{array}$$

Then we specify

$$\begin{array}{rrcl} & \mathit{tonat} & \in & \mathbf{bin} \to \mathbf{nat} \\ \mathit{tonat} & \epsilon & = & \mathbf{0} \\ \mathit{tonat} & (b\,\mathbf{0}) & = & \mathit{double}\ (\mathit{tonat}\ b) \\ \mathit{tonat} & (b\,\mathbf{1}) & = & \mathbf{s}(\mathit{double}\ (\mathit{tonat}\ b)) \end{array}$$

Note that this satisfies the schema of primitive recursion. Now we can see why we think of binary numbers as satisying some non-structural equality: every natural number has an infinite number of bit strings as representations, because we can always add leading zeroes without changing the result of *tonat*. For example,

$$tonat(\epsilon\,\mathbf{1}) =_N tonat(\epsilon\,\mathbf{0}\,\mathbf{1}) =_N tonat(\epsilon\,\mathbf{0}\,\mathbf{0}\,\mathbf{1}) =_N \mathbf{s}(\mathbf{0})$$

This has several consequences. If we think of bit strings only as a means to represent natural numbers, we would define equality such that $\epsilon =_B \epsilon\,\mathbf{0}$. Secondly, we can define functions which are ill-defined as far as their interpretation as natural numbers is concerned. For example,

$$\begin{array}{lrcl} \mathit{flip} & \epsilon & = & \epsilon \\ \mathit{flip} & (b\,\mathbf{0}) & = & b\,\mathbf{1} \\ \mathit{flip} & (b\,\mathbf{1}) & = & b\,\mathbf{0} \end{array}$$

may make sense intuitively, but it maps ϵ and $\epsilon\mathbf{0}$ to different results and thus does not respect the intended equality.

A general mechanism to deal with such problems is to define *quotient types*. This is somewhat more complicated than needed in most instances, so we will stick to the simpler idea of just verifying that functions implement the intended operations on natural numbers.

A simple example is the increment function *inc* on binary numbers. Assume a bit string b represents a natural number n. When we can show that $inc(b)$ always represents $\mathbf{s}(n)$ we say that *inc implements* $\mathbf{s}$. In general, a function f *implements* g if $f(b)$ represents $g(n)$ whenever b represents n. Representation is defined via the function *tonat*, so by definition f implements g if we can prove that

$$\forall b{\in}\mathbf{bin}.\ tonat(f\ b) =_N g(tonat\ b)$$

In our case:

$$\forall b{\in}\mathbf{bin}.\ tonat(inc\ b) =_N \mathbf{s}(tonat\ b)$$

The increment function is primitive recursive and defined as follows:

$$\begin{array}{lrcl} & \mathit{inc} & \in & \mathbf{bin} \to \mathbf{bin} \\ \mathit{inc} & \epsilon & = & \epsilon\,\mathbf{1} \\ \mathit{inc} & (b\,\mathbf{0}) & = & b\,\mathbf{1} \\ \mathit{inc} & (b\,\mathbf{1}) & = & (\mathit{inc}\ b)\,\mathbf{0} \end{array}$$

Now we can prove that *inc* correctly implements the successor function.

$$\forall b{\in}\mathbf{bin}.\ tonat(inc\ b) =_N \mathbf{s}(tonat\ b)$$

Proof: By structural induction on b.

Case: $b = \epsilon$.

$$\begin{array}{ll} \text{lhs:} & \mathit{tonat}(\mathit{inc}\ \epsilon) \\ & \Longrightarrow \mathit{tonat}(\epsilon\ \mathbf{1}) \\ & \Longrightarrow \mathbf{s}(\mathit{double}(\mathit{tonat}\ \epsilon)) \\ & \Longrightarrow \mathbf{s}(\mathbf{0}) \\ \text{rhs:} & \mathbf{s}(\mathit{tonat}\ \epsilon) \Longrightarrow \mathbf{s}(\mathbf{0}) \end{array}$$

Case: $b = b'\,\mathbf{0}$. We simply calculate left- and right-hand side without appeal to the induction hypothesis.

$$\begin{array}{ll} \text{lhs:} & \mathit{tonat}(\mathit{inc}(b'\,\mathbf{0})) \\ & \Longrightarrow \mathit{tonat}(b'\,\mathbf{1}) \\ & \Longrightarrow \mathbf{s}(\mathit{double}(\mathit{tonat}\ b')) \\ \text{rhs:} & \mathbf{s}(\mathit{tonat}(b'\,\mathbf{0})) \\ & \Longrightarrow \mathbf{s}(\mathit{double}(\mathit{tonat}\ b')) \end{array}$$

Case: $b = b'\,\mathbf{1}$. In this case we need the induction hypothesis.

$$\mathit{tonat}(\mathit{inc}\ b') =_N \mathbf{s}(\mathit{tonat}\ b')$$

Then we compute as usual, starting from the left- and right-hand sides.

$$\begin{array}{lll} \text{lhs:} & \mathit{tonat}(\mathit{inc}(b'\,\mathbf{1})) & \\ & \Longrightarrow \mathit{tonat}((\mathit{inc}\ b')\,\mathbf{0}) & \\ & \Longrightarrow \mathit{double}(\mathit{tonat}(\mathit{inc}\ b')) & \\ & \Longrightarrow \mathit{double}(\mathbf{s}(\mathit{tonat}\ b')) & \text{by ind. hyp.} \\ & \Longrightarrow \mathbf{s}(\mathbf{s}(\mathit{double}(\mathit{tonat}\ b'))) & \\ \text{rhs:} & \mathbf{s}(\mathit{tonat}(b'\,\mathbf{1})) & \\ & \Longrightarrow \mathbf{s}(\mathbf{s}(\mathit{double}(\mathit{tonat}\ b'))) & \end{array}$$

□

The second case of the proof looks straightforward, but we have swept an important step under the rug. The induction hypothesis had the form $s =_N t$. We used it to conclude $\mathit{double}(s) =_N \mathit{double}(t)$. This is a case of a general substitution principle for equality. However, our notion of equality on natural numbers is defined by introduction and elimination rules, so we need to justify this principle. In general, an application of substitutivity of equality can have one of the two forms

$$\frac{\Gamma \vdash m =_N n\ \mathit{true} \qquad \Gamma \vdash A(m)\ \mathit{true}}{\Gamma \vdash A(n)\ \mathit{true}}\ \mathit{subst}$$

$$\frac{\Gamma \vdash m =_N n\ \mathit{true} \qquad \Gamma \vdash A(n)\ \mathit{true}}{\Gamma \vdash A(m)\ \mathit{true}}\ \mathit{subst}'$$

The second one is easy to justify from the first one by symmetry of equality.

These are examples of *admissible rules of inference*. We cannot derive them directly from the elimination rules for equality, but every instance of them is correct. In general, we say that an inference rule is *admissible* if every instance of the rule is valid.

Theorem: The rule *subst* is admissible.

Proof: By induction on m.

Case: $m = \mathbf{0}$. Then we distinguish cases on n.

Case: $n = \mathbf{0}$. Then $A(m) = A(0) = A(n)$ and the right premise and conclusion are identical.

Case: $n = \mathbf{s}(n')$. Then the right premise is not even needed to obtain the conclusion.

$$\frac{\Gamma \vdash \mathbf{0} =_N \mathbf{s}(n')\ \mathit{true}}{\Gamma \vdash A(\mathbf{s}(n'))\ \mathit{true}} =_N E_{0s}$$

Case: $m = \mathbf{s}(m')$. Then we distinguish cases on n.

Case: $n = \mathbf{0}$. Again, the right premise is not needed to justify the conclusion.

$$\frac{\Gamma \vdash \mathbf{s}(m') =_N \mathbf{0}\ \mathit{true}}{\Gamma \vdash A(\mathbf{0})\ \mathit{true}} =_N E_{s0}$$

Case: $n = \mathbf{s}(n')$. Then we derive the rule as follows.

$$\frac{\dfrac{\Gamma \vdash \mathbf{s}(m') =_N \mathbf{s}(n')\ \mathit{true}}{\Gamma \vdash m' =_N n'\ \mathit{true}} =_N E_{ss} \qquad \Gamma \vdash A(\mathbf{s}(m'))\ \mathit{true}}{\Gamma \vdash A(\mathbf{s}(n'))}\ i.h.$$

Here, a derivation of the conclusion exists by induction hypothesis on m'. Critical is to use the induction hypothesis for $B(m') = A(\mathbf{s}(m'))$ which yields the desired $B(n') = A(\mathbf{s}(n'))$ in the conclusion.

□

In this case, we must formulate the desired principle as a rule of inference. We can write it out as a parametric proposition,

$$\forall m{\in}\mathbf{nat}.\ \forall n{\in}\mathbf{nat}.\ m =_N n \supset A(m) \supset A(n)$$

but this can not be proven parametrically in A. The problem is that we need to use the induction hypothesis with a predicate different from A, as the last case in our proof of admissibility shows. And quantification over A, as in

$$\forall m{\in}\mathbf{nat}.\ \forall n{\in}\mathbf{nat}.\ m =_N n \supset \forall A{:}\mathbf{nat} \to \mathit{prop}.\ A(m) \supset A(n)$$

is outside of our language. In fact, quantification over arbitrary propositions or predicates can not be explained satisfactorily using our approach, since the domain of quantification (such at $\mathbf{nat} \to \mathit{prop}$ in the example), includes the new kind of proposition we are just defining. This is an instance of *impredicativity* which is rejected in constructive type theory in the style of Martin-Löf. The rules for quantification over propositions would be something like

$$\frac{\Gamma, p\ \mathit{prop} \vdash A(p)\ \mathit{prop}}{\Gamma \vdash \forall_2 p{:}\mathit{prop}.\ A(p)\ \mathit{prop}}\,\forall_2 F^p$$

$$\frac{\Gamma, p\ \mathit{prop} \vdash A(p)\ \mathit{true}}{\Gamma \vdash \forall_2 p{:}\mathit{prop}.\ A(p)\ \mathit{true}}\,\forall_2 I^p \qquad \frac{\Gamma \vdash \forall_2 p{:}\mathit{prop}.\ A(p)\ \mathit{true} \qquad \Gamma \vdash B\ \mathit{prop}}{\Gamma \vdash A(B)\ \mathit{true}}\,\forall_2 E$$

The problem is that $A(p)$ is not really a subformula of $\forall_2 p{:}\mathit{prop}.\ A(p)$. For example, we can instantiate a proposition with itself!

$$\frac{\Gamma \vdash \forall_2 p{:}\mathit{prop}.\ p \supset p\ \mathit{true} \qquad \Gamma \vdash \forall_2 p{:}\mathit{prop}.\ p \supset p\ \mathit{prop}}{\Gamma \vdash (\forall_2 p{:}\mathit{prop}.\ p \supset p) \supset (\forall_2 p{:}\mathit{prop}.\ p \supset p)\ \mathit{true}}\,\forall_2 E$$

Nonetheless, it is possible to allow this kind of quantification in constructive or classical logic, in which case we obtain *higher-order logic*. Another solution is to introduce *universes*. In essence, we do not just have one kind of proposition, by a whole hierarchy of propositions, where higher levels may include quantification over propositions at a lower level. We will not take this extra step here and instead simply use admissible rules of inference, as in the case of substitutivity above.

Returning to data representation, some functions are easy to implement. For example,

$$\begin{array}{rcl} \mathit{shiftl} & \in & \mathbf{bin} \to \mathbf{bin} \\ \mathit{shiftl}\ \ b & = & b\,\mathbf{0} \end{array}$$

implements the double function.

$$\forall b{\in}\mathbf{bin}.\ \mathit{tonat}(\mathit{shiftl}\ b) =_N \mathit{double}(\mathit{tonat}\ b)$$

Proof: By computation.

$$\mathit{tonat}(\mathit{shiftl}\ b) \Longrightarrow \mathit{tonat}(b\,\mathbf{0}) \Longrightarrow \mathit{double}(\mathit{tonat}\ b)$$

□

This trival example illustrates why it is convenient to allow multiple representations of natural numbers. According to the definition above, we have $\mathit{shiftl}\ \epsilon \Longrightarrow \epsilon\,\mathbf{0}$. The result has leading zeroes. If we wanted to keep representations in a standard form without leading zeroes, doubling would have to have a more complicated definition. The alternative approach to work only with standard forms in the representation is related to the issue of data structure invariants, which will be discussed in the next section.

In general, proving the representation theorem for some functions may require significant knowledge in the theory under consideration. As an example, we consider addition on binary numbers.

$$
\begin{array}{llrcl}
 & & \mathit{add} & \in & \mathbf{bin} \to \mathbf{bin} \to \mathbf{bin} \\
\mathit{add} & \epsilon & c & = & c \\
\mathit{add} & (b\,\mathbf{0}) & \epsilon & = & b\,\mathbf{0} \\
\mathit{add} & (b\,\mathbf{0}) & (c\,\mathbf{0}) & = & (\mathit{add}\ b\ c)\,\mathbf{0} \\
\mathit{add} & (b\,\mathbf{0}) & (c\,\mathbf{1}) & = & (\mathit{add}\ b\ c)\,\mathbf{1} \\
\mathit{add} & (b\,\mathbf{1}) & \epsilon & = & b\,\mathbf{1} \\
\mathit{add} & (b\,\mathbf{1}) & (c\,\mathbf{0}) & = & (\mathit{add}\ b\ c)\,\mathbf{1} \\
\mathit{add} & (b\,\mathbf{1}) & (c\,\mathbf{1}) & = & (\mathit{inc}\ (\mathit{add}\ b\ c))\,\mathbf{0}
\end{array}
$$

This specification is primitive recursive: all recursive calls to *add* are on b. The representation theorem states

$$\forall b{\in}\mathbf{bin}.\ \forall c{\in}\mathbf{bin}.\ \mathit{tonat}(\mathit{add}\ b\ c) =_N \mathit{plus}\ (\mathit{tonat}\ b)\ (\mathit{tonat}\ c)$$

The proof by induction on b of this property is left as an exercise to the reader. One should be careful to extract the needed properties of the natural numbers and addition and prove them separately as lemmas.

4.7 Complete Induction

In the previous section we have seen an example of a correct rule of inference which was not derivable, only admissible. This was because our logic was not expressive enough to capture this inference rule as a proposition. In this section we investigate a related question: is the logic expressive enough so we can derive different induction principles?

The example we pick is the principle of *complete induction* also known as *course-of-values induction*. On natural numbers, this allows us to use the induction hypothesis on any number smaller than the induction induction variable. The principle of mathematical induction considered so far allows only the immediate predecessor. Corresponding principles exist for structural inductions. For examples, complete induction for lists allows us to apply the induction hypothesis on any tail of the original list.

Complete induction is quite useful in practice. As an example we consider the integer logarithm function. First, recall the specification of *half*.

$$
\begin{array}{lrcl}
 & \mathit{half} & \in & \mathbf{nat} \to \mathbf{nat} \\
\mathit{half} & \mathbf{0} & = & \mathbf{0} \\
\mathit{half} & (\mathbf{s}(\mathbf{0})) & = & \mathbf{0} \\
\mathit{half} & (\mathbf{s}(\mathbf{s}(n))) & = & \mathbf{s}(\mathit{half}\,(n))
\end{array}
$$

This function is not immediately primitive recursive, but it follows the schema of course-of-values recursion. This is because the recursive call to $\mathit{half}(n+$

2) is on n and $n < n + 2$. We have seen how this can be transformed into a primitive recursion using pairs. In a sense, we show in this section that *every* function specified using course-of-values recursion can be implemented by primitive recursion. Since we prove this constructively, we actually have an effective method to implement course-of-values recursion by primitive recursion.

Next we specify the function $lg(n)$ which calculates the number of bits in the binary representation of n. Mathematically, we have $lg(n) = \lfloor log_2(n+1) \rfloor$.

$$\begin{array}{lrcl} & lg & \in & \mathbf{nat} \to \mathbf{nat} \\ lg & \mathbf{0} & = & \mathbf{0} \\ lg & (\mathbf{s}(n)) & = & \mathbf{s}(lg(half(\mathbf{s}(n)))) \end{array}$$

This specifies a terminating function because $half(\mathbf{s}(n)) < \mathbf{s}(n)$. We now introduce the principal of complete induction and then verify the observation that lg is a terminating function.

> **Principle of Complete Induction.** In order to prove $A(n)$, assume $\forall z{\in}\mathbf{nat}.\ z < x \supset A(z)$ and prove $A(x)$ for arbitrary x.

In order to simplify the discussion below, we say the property A is *complete* if $\forall x{\in}\mathbf{nat}.\ (\forall z{\in}\mathbf{nat}.\ z < x \supset A(z)) \supset A(x)$ is true.

Why is this induction principle valid? The idea is as follows: assume A is complete. We want to show that $\forall n{\in}\mathbf{nat}.\ A(n)$ holds. Why does $A(0)$ hold? If A is complete, then $A(0)$ must be true because there is no $z < 0$. Now, inductively, if $A(0), A(1), \ldots, A(n)$ are all true, then $A(\mathbf{s}(n))$ must also be true, because $A(z)$ for every $z < \mathbf{s}(n)$ and hence $A(n)$ by completeness.

More explicitly, we can prove the principle of complete induction correct as follows.

$$(\forall x{\in}\mathbf{nat}.\ (\forall z{\in}\mathbf{nat}.\ z < x \supset A(z)) \supset A(x)) \supset \forall n{\in}\mathbf{nat}.\ A(n)$$

However, a direct proof attempt of this theorem fails—the induction hypothesis needs to be generalized. The structure of the brief informal argument tells us what it must be.

> **Proof:** Assume A is complete, that is
>
> $$(\forall x{\in}\mathbf{nat}.\ (\forall z{\in}\mathbf{nat}.\ z < x \supset A(z)) \supset A(x)).$$
>
> We show that
>
> $$\forall n{\in}\mathbf{nat}.\ \forall m{\in}\mathbf{nat}.\ m < n \supset A(m)$$
>
> by induction on n. From this the theorem follows immediately. Now to the proof of the generalized theorem.
>
> **Case:** $n = \mathbf{0}$. We have to show $\forall m{\in}\mathbf{nat}.\ m < \mathbf{0}.\ A(m)$. So let m be given and assume $m < \mathbf{0}$. But this is contradictory, so we conclude $A(m)$ by rule $<E_0$.

Case: $n = \mathbf{s}(n')$. We assume the induction hypothesis:

$$\forall m{\in}\mathbf{nat}.\ m < n'.\ A(m).$$

We have to show:

$$\forall m{\in}\mathbf{nat}.\ m < \mathbf{s}(n').\ A(m).$$

So let m be given and assume $m < \mathbf{s}(n')$. Then we distinguish two cases: $m =_N n'$ or $m < n'$. It is a straightforward lemma (which have not proven), that $m < \mathbf{s}(n') \supset (m =_N n' \vee m < n')$.

Subcase: $m =_N n'$. From the completeness of A, using n' for x, we get

$$(\forall z{\in}\mathbf{nat}.\ z < n' \supset A(z)) \supset A(n').$$

But, by renaming z to m the left-hand side of this implication is the induction hypothesis and we conclude $A(n')$ and therefore $A(m)$ by substitution from $m =_N n'$.

Subcase: $m < n'$. Then $A(m)$ follows directly from the induction hypothesis.

□

Now we can use this, for example, to show that the *lg* function is total. For this we formalize the specification from above as a proposition. So assume $lg \in \mathbf{nat} \to \mathbf{nat}$, and assume

$$(lg\ \mathbf{0} =_N \mathbf{0}) \wedge$$
$$(\forall n{\in}\mathbf{nat}.\ lg\ (\mathbf{s}(n)) =_N \mathbf{s}(lg(half(\mathbf{s}(n)))))$$

We prove

$$\forall x{\in}\mathbf{nat}.\ \exists y{\in}\mathbf{nat}.\ lg(x) =_N y$$

This expresses that *lg* describes a total function. In fact, from this constructive proof we can eventually *extract* a primitive recursive implementation of *lg*!

Proof: By complete induction on x. Note that in this proof the property

$$A(x) = (\exists y.\ lg(x) =_N y)$$

Assume the complete induction hypothesis:

$$\forall z.\ z < x \supset \exists y.\ lg(z) =_N y$$

Following the structure of the specification, we distinguish two cases: $x = \mathbf{0}$ and $x = \mathbf{s}(x')$.

Case: $x = \mathbf{0}$. Then $y = \mathbf{0}$ satisfies the specification since $lg(\mathbf{0}) =_N \mathbf{0}$.

Case: $x = \mathbf{s}(x')$. Then $\mathit{half}(\mathbf{s}(x')) < \mathbf{s}(x')$ (by an unproven lemma) and we can use the induction hypothesis to obtain a y' such that $\mathit{lg}(\mathit{half}(\mathbf{s}(x'))) =_N y'$. Then $y = \mathbf{s}(y')$ satisfies

$$\mathit{lg}(\mathbf{s}(x')) =_N \mathbf{s}(\mathit{lg}(\mathit{half}(\mathbf{s}(x')))) =_N \mathbf{s}(y')$$

by the specification of lg and transitivity of equality.

□

Next we examine the computational contents of these proofs. First, the correctness of the principle of complete induction. For simplicity, we assume an error element $\mathit{error} \in \mathbf{0}$. Then we hide information in the statement of completeness in the following manner:

$$\forall x{\in}\mathbf{nat}.\ (\forall z{\in}\mathbf{nat}.\ [z < x] \supset A(z)) \supset A(x)$$

If we assume that a type τ represents the computational contents of A, then this corresponds to

$$c \in \mathbf{nat} \to (\mathbf{nat} \to \tau) \to \tau$$

In the proof of complete induction, we assume that A is complete. Computationally, this means we assume a function c of this type. In the inductive part of the proof we show

$$\forall n{\in}\mathbf{nat}.\ \forall m{\in}\mathbf{nat}.\ [m < n] \supset A(m)$$

The the function h extracted from this proof satisfies

$$\begin{array}{llcll} & & h & \in & \mathbf{nat} \to \mathbf{nat} \to \tau & \\ h & \mathbf{0} & m & = & \mathbf{abort}(\mathit{error}) & \\ h & (\mathbf{s}(n')) & m & = & c\ m\ (\lambda m'.\ h(n')\ m') & \text{for } m =_N n' \\ h & (\mathbf{s}(n')) & m & = & h(n')\ m & \text{for } m < n' \end{array}$$

Note that h is clearly primitive recursive in its first argument. In this specification the nature of the proof and the cases it distinguishes are clearly reflected. The overall specification

$$\forall n{\in}\mathbf{nat}.\ A(n)$$

is contracted to a function f where

$$\begin{array}{lcl} f & : & \mathbf{nat} \to \tau \\ f\ \ n & = & h\ (\mathbf{s}(n))\ n \end{array}$$

which is not itself recursive, but just calls h.

Assume a function f is defined by the schema of complete recursion and we want to compute $f(n)$. We compute it by primitive recursion on h, starting with $h\ (\mathbf{s}(n))\ n$. The first argument to h is merely a counter. We start at $\mathbf{s}(n)$ and count it down all the way to $\mathbf{s}(\mathbf{0})$. This is what makes the definition of h primitive recursive.

Meanwhile, in the second argument to h (which is always smaller than the first), we compute as prescribed by f. Assume $f(n')$ calls itself recursively on $g(n') < n'$. Then we compute $h\ (\mathbf{s}(n'))\ n'$ by computing $h\ n'\ (g(n'))$, which is a legal recursive call for h. The situation is complicated by the fact that f might call itself recursively several times on different arguments, so we may need to call h recursively several times. Each time, however, the first argument will be decreased, making the recursion legal.

As an example, consider the specification of lg that satisfies the schema of complete recursion since $\mathit{half}(\mathbf{s}(n)) < \mathbf{s}(n)$.

$$\begin{array}{l}(lg\ \mathbf{0} =_N \mathbf{0}) \wedge \\ (\forall n{\in}\mathbf{nat}.\ lg\ (\mathbf{s}(n)) =_N \mathbf{s}(lg(\mathit{half}(\mathbf{s}(n)))))\end{array}$$

The function c that is extracted from the proof of

$$\forall x{\in}\mathbf{nat}.\ \exists y{\in}\mathbf{nat}.\ [lg(x) =_N y]$$

assuming completeness is

$$\begin{array}{lrcll} & & c & \in & \mathbf{nat} \to (\mathbf{nat} \to \mathbf{nat}) \to \mathbf{nat} \\ c & \mathbf{0} & r & = & \mathbf{0} \\ c & (\mathbf{s}(n')) & r & = & \mathbf{s}(r\ (\mathit{half}\ (\mathbf{s}(n')))) \end{array}$$

Note that the second argument to c called r represents the induction hypothesis. c itself is not recursive since we only *assumed* the principle of complete induction. To obtain an implementation of lg we must use the proof of the principle of complete induction. Next, the helper function h is

$$\begin{array}{lrcll} & & h & \in & \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat} & \\ h & \mathbf{0} & m & = & \mathbf{abort}(\mathit{error}) & \\ h & (\mathbf{s}(n')) & m & = & c\ m\ (\lambda m'.\ h(n')\ m') & \text{for } m =_N n' \\ h & (\mathbf{s}(n')) & m & = & h(n')\ m & \text{for } m < n' \end{array}$$

We can expand the definition of c on the right-hand side for the special case of the logarithm and obtain:

$$\begin{array}{lrrcll} h & \mathbf{0} & m & = & \mathbf{abort}(\mathit{error}) & \\ h & (\mathbf{s}(\mathbf{0})) & \mathbf{0} & = & \mathbf{0} & \\ h & (\mathbf{s}(\mathbf{s}(n''))) & (\mathbf{s}(n'')) & = & \mathbf{s}(h(\mathbf{s}(n''))\ (\mathit{half}\ (\mathbf{s}(n')))) & \\ h & (\mathbf{s}(n')) & m & = & h(n')\ m & \text{for } m < n' \end{array}$$

and

$$\begin{array}{rcl} lg & : & \mathbf{nat} \to \mathbf{nat} \\ lg\ n & : & h\ (\mathbf{s}(n))\ n \end{array}$$

which can easily be seen as a primitive recursive definition of lg since equality and less-than are decidable and we can eliminate the dependency on *error* by returning an arbitrary number in the (impossible) first case in the definition of h.

4.8 Dependent Types

We have encountered a number of constructors for propositions and types. Generally, propositions are constructed from simpler propositions, and types are constructed from simpler types. Furthermore, propositions refer to types (such as $\forall x{\in}\tau.\ A(x)$), and propositions refer to terms (such as $n =_N m$). However, we have not seen a type that refers to either a term or a proposition. In this section we consider the former. As we will see, allowing types to be constructed from terms has a number of applications, but it also creates a number of problems.

As an example we consider lists. Rather than simply keeping track of the types of their elements as we have done so far, we keep track of the length of the list as part of the type. We obtain the following formation rule:

$$\frac{\Gamma \vdash \tau\ \mathit{type} \qquad \Gamma \vdash n \in \mathbf{nat}}{\Gamma \vdash \tau\,\mathbf{list}(n)\ \mathit{type}}\,\mathbf{list}F$$

Note that we now make a context Γ explicit in this judgment, since the term n which occurs inside the type $\tau\,\mathbf{list}(n)$ may contain variables. We call $\tau\,\mathbf{list}$ a *type family* and n its *index term*.

The meaning of the type $\tau\,\mathbf{list}(n)$ is the type of lists with elements of type τ and length n. The introduction rules for this type track the length of the constructed list.

$$\frac{}{\Gamma \vdash \mathbf{nil}^\tau \in \tau\,\mathbf{list}(\mathbf{0})}\,\mathbf{list}I_n \qquad \frac{\Gamma \vdash s \in \tau \qquad \Gamma \vdash l \in \tau\,\mathbf{list}(n)}{\Gamma \vdash s :: l \in \tau\,\mathbf{list}(\mathbf{s}(n))}\,\mathbf{list}I_c$$

The elimination rule now must track the length of the list as well. Written as a schema of primitive recursion, we obtain

$$\begin{array}{lrcl} f & (\mathbf{0}, \mathbf{nil}) & = & s_n \\ f & (\mathbf{s}(n'), x :: l') & = & s_c(n', x, l', f(n', l')) \end{array}$$

where s_n contains no occurrence of f, and all occurrences of f in s_c have the indicated form of $f(n', l')$. Note that coupling occurrences of n and l in the schema guarantees that the typing remains consistent: even occurrence of $f(n, l)$ contains a list l in the second argument and its length in the first argument. Transforming this rule into an elimination rule yields

$$\frac{\begin{array}{l}\Gamma \vdash l \in \tau\,\mathbf{list}(n) \\ \Gamma \vdash s_n \in \sigma(\mathbf{0}, \mathbf{nil}) \\ \Gamma, n'{\in}\mathbf{nat}, x{\in}\tau, l'{\in}\tau\,\mathbf{list}(n'), f(n', l'){\in}\sigma(n', l') \vdash s_c \in \sigma(\mathbf{s}(n'), x :: l')\end{array}}{\Gamma \vdash (\mathbf{rec}\ l\ \mathbf{of}\ f(\mathbf{0}, \mathbf{nil}) \Rightarrow s_n \mid f(\mathbf{s}(n'), x :: l') \Rightarrow s_c) \in \sigma(n, l)}\,\mathbf{list}E$$

Here we have written the premises on top of each other for typographical reasons. There are two complications in this rule. The first is that we have to iterate over the lists and its length at the same time. The second is that now types may depend on terms. Therefore the type σ may actually depend on both n

and l, and this must be reflected in the rule. In fact, it looks very much like a rule of induction if we read the type $\sigma(n, l)$ as a proposition $A(n, l)$. Allowing types to depend on terms make types look even more like propositions than before. In fact, we are close to extending the Curry-Howard isomorphism from the propositional to the first-order case.

Next we consider how to use elements of this new type in some examples. The first is appending of two lists. We would like to say

$$\mathit{app} \in \tau\,\mathbf{list}(n) \to \tau\,\mathbf{list}(m) \to \tau\,\mathbf{list}(n+m)$$

that is, *app* takes a list of length n and a list of length m and returns a list of length $n + m$. But what is the status of n and m in this declaration? We can see that at least n cannot be a global parameter (as τ, for example, since it changes during the recursion. Instead, we make it explicit in the type, using a new type constructor Π. This constructor acts on types exactly the way that $\forall$ acts on propositions. With it, we can write

$$\mathit{app} \in \Pi n{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(n) \to \Pi m{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(m) \to \tau\,\mathbf{list}(n+m)$$

so that *app* is now a function of four arguments: a number n, a list of length n, a number m, and then a list of length m. The function returns a list of length $n + m$.

The rules for Π are constructed in complete analogy with $\forall$.

$$\frac{\Gamma \vdash \tau\ \mathit{type} \qquad \Gamma, x{\in}\tau \vdash \sigma(x)\ \mathit{type}}{\Gamma \vdash \Pi x{\in}\tau.\ \sigma(x)\ \mathit{type}}\,\Pi F$$

$$\frac{\Gamma, x{\in}\tau \vdash s \in \sigma(x)}{\Gamma \vdash \lambda x{\in}\tau.\ s \in \Pi x{\in}\tau.\ \sigma(x)}\,\Pi I \qquad \frac{\Gamma \vdash s \in \Pi x{\in}\tau.\ \sigma(x) \qquad \Gamma \vdash t \in \tau}{\Gamma \vdash s\,t \in \sigma(t)}\,\Pi E$$

$\Pi x \in \tau.\ \sigma(x)$ is called a *dependent function type*, because it denotes a function whose result type depends on the value of the argument. As for universal quantification, substitution is required in the elimination rule. With this in mind, we can first write and then type-check the specification of *app*.

$$\mathit{app} \in \Pi n{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(n) \to \Pi m{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(m) \to \tau\,\mathbf{list}(n+m)$$

$$\begin{array}{llllll} \mathit{app} & \mathbf{0} & \mathbf{nil} & m & k & = & k \\ \mathit{app} & (\mathbf{s}(n')) & (x :: l') & m & k & = & x :: (\mathit{app}\ n'\ l'\ m\ k) \end{array}$$

For each equation in this specification we type-check both the left- and the right-hand side and verify that they are the same. We show the checking of subterm to help the understanding of the type-checking process. First, the left-hand side of the first equation.

$$\begin{array}{lll} \mathit{app}\ \mathbf{0} & \in & \tau\,\mathbf{list}(\mathbf{0}) \to \Pi m{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(m) \to \tau\,\mathbf{list}(\mathbf{0}+m) \\ \mathit{app}\ \mathbf{0}\ \mathbf{nil} & \in & \Pi m{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(m) \to \tau\,\mathbf{list}(\mathbf{0}+m) \\ \mathit{app}\ \mathbf{0}\ \mathbf{nil}\ m & \in & \tau\,\mathbf{list}(m) \to \tau\,\mathbf{list}(\mathbf{0}+m) \qquad \text{for } m{\in}\mathbf{nat} \\ \mathit{app}\ \mathbf{0}\ \mathbf{nil}\ m\ k & \in & \tau\,\mathbf{list}(\mathbf{0}+m) \qquad \text{for } k{\in}\tau\,\mathbf{list}(m) \\ \\ k & \in & \tau\,\mathbf{list}(m) \end{array}$$

While the two types are different, the first one can be reduced to the second. Just like previously for propositions, we therefore need rules of computation for types.

$$\frac{\Gamma \vdash s : \sigma \qquad \sigma \Longrightarrow \sigma' \qquad \Gamma \vdash \sigma\ \mathit{type}}{\Gamma \vdash s : \sigma'}\ \mathit{conv}$$

$$\frac{\Gamma \vdash s : \sigma' \qquad \sigma \Longrightarrow \sigma' \qquad \Gamma \vdash \sigma\ \mathit{type}}{\Gamma \vdash s : \sigma}\ \mathit{conv}'$$

Next, we consider the second equation, first the left-hand and then the right-hand side.

$$\begin{array}{lcl}
\mathit{app}\ (\mathbf{s}(n')) & \in & \tau\ \mathbf{list}(\mathbf{s}(n')) \to \Pi m{\in}\mathbf{nat}.\ \tau\ \mathbf{list}(m) \to \tau\ \mathbf{list}(\mathbf{s}(n') + m) \\
 & & \qquad \text{for } n' \in \mathbf{nat} \\
\mathit{app}\ (\mathbf{s}(n'))\ (x :: l') & \in & \Pi m{\in}\mathbf{nat}.\ \tau\ \mathbf{list}(m) \to \tau\ \mathbf{list}(\mathbf{s}(n') + m) \\
 & & \qquad \text{for } x \in \tau \text{ and } l' \in \tau\ \mathbf{list}(n') \\
\mathit{app}\ (\mathbf{s}(n'))\ (x :: l')\ m & \in & \tau\ \mathbf{list}(m) \to \tau\ \mathbf{list}(\mathbf{s}(n') + m) \qquad \text{for } m{\in}\mathbf{nat} \\
\mathit{app}\ (\mathbf{s}(n'))\ (x :: l')\ m\ k & \in & \tau\ \mathbf{list}(\mathbf{s}(n') + m) \qquad \text{for } k{\in}\tau\ \mathbf{list}(m) \\
 & & \\
\mathit{app}\ n'\ l'\ m\ k & \in & \tau\ \mathbf{list}(n' + m) \\
x :: (\mathit{app}\ n'\ l'\ m\ k) & \in & \tau\ \mathbf{list}(\mathbf{s}(n' + m))
\end{array}$$

Again, we can obtain the right-hand side by computation from the left-hand side

$$\tau\ \mathbf{list}(\mathbf{s}(n') + m) \Longrightarrow \tau\ \mathbf{list}(\mathbf{s}(n' + m))$$

since addition is defined by primitive recursion over the first argument.

For the sake of completeness, we now show an explicit definition of *app* by primitive recursion.

$$\begin{array}{lcl}
\mathit{app} & = & \lambda n{\in}\mathbf{nat}.\ \lambda l{\in}\tau\ \mathbf{list}(n). \\
 & & \quad \mathbf{rec}\ l \\
 & & \quad\ \mathbf{of}\ f(\mathbf{0}, \mathbf{nil}) \Rightarrow \lambda m{\in}\mathbf{nat}.\ \lambda k{\in}\tau\ \mathbf{list}(m).\ k \\
 & & \quad\ \mid f(\mathbf{s}(n'), x :: l') \Rightarrow \lambda m{\in}\mathbf{nat}.\ \lambda k{\in}\tau\ \mathbf{list}(m).\ x :: (f(n', l')\ m\ k)
\end{array}$$

From the practical point of view, we would like to avoid passing the lengths of the lists as arguments to *app*. In the end, we are interested in the list as a result, and not its length. In order to capture this, we extend the erasure notation $[A]$ and $[M]$ from propositions and proof terms to types $[\tau]$ and terms $[t]$. The meaning is completely analogous. Since we don't want to pass length information, we obtain

$$\begin{array}{llllllll}
\multicolumn{8}{l}{\mathit{app} \in \Pi[n]{\in}[\mathbf{nat}].\ \tau\ \mathbf{list}[n] \to \Pi[m]{\in}[\mathbf{nat}].\ \tau\ \mathbf{list}[m] \to \tau\ \mathbf{list}[n + m]} \\
\mathit{app} & \mathbf{[0]} & \mathbf{nil} & [m] & k & = & k & \\
\mathit{app} & [\mathbf{s}(n')] & (x :: l') & [m] & k & = & x :: (\mathit{app}\ [n']\ l'\ [m]\ k) &
\end{array}$$

Fortunately, this annotation is consistent: we never use a bracketed variable outside of brackets. That is, we never try to construct an answer out of a

variable that will not be carried at runtime. After erasure of the bracketed terms and types and simplification, we obtain the prior definition of *app* on lists that are not indexed by their length.

But not every function can be consistently annotated. As a simple counterexample consider the following length function:

$$\begin{array}{rcl} \mathit{length} & \in & \Pi n{\in}\mathbf{nat}.\ \tau\ \mathbf{list}(n) \to \mathbf{nat} \\ \mathit{length}\ \ n\ \ l & = & n \end{array}$$

This is a perfectly valid implementation of *length*: from type-checking we know that l must have length n. However, if we try to annotate this function

$$\begin{array}{rcl} \mathit{length} & \in & \Pi[n]{\in}[\mathbf{nat}].\ \tau\ \mathbf{list}[n] \to \mathbf{nat} \\ \mathit{length}\ \ [n]\ \ l & = & n \end{array}$$

we observe a use of n outside of brackets which is illegal. Indeed, if n is not passed at run-time, then we cannot "compute" the length in this way. Fortunately, there is another obvious definition of length that can be annotated in the desired way.

$$\begin{array}{lrcl} \mathit{length} & [\mathbf{0}] \quad\ \ \mathbf{nil} & = & \mathbf{0} \\ \mathit{length} & [\mathbf{s}(n')] \quad (x :: l') & = & \mathbf{s}(\mathit{length}\ [n']\ l') \end{array}$$

which has the property that the bracketed variable n' from the left-hand side also occurs only bracketed on the right-hand side. Note that dependent type-checking does *not* verify the correctness of this second implementation of *length* in the sense that the type does not exhibit a connection between the length argument n and the natural number that is returned.

The use of dependent types goes very smoothly for the examples above, but what happens when the length of an output list to a function is unknown? Consider the *filter* function which retains only those elements of a list that satisfy a given predicate p. We first give the definition with the ordinary lists not indexed by their length.

$$\begin{array}{rcl} \mathit{filter} & \in & (\tau \to \mathbf{bool}) \to \tau\ \mathbf{list} \to \tau\ \mathbf{list} \\ \mathit{filter}\ \ p\ \ \mathbf{nil} & = & \mathbf{nil} \\ \mathit{filter}\ \ p\ \ (x :: l') & = & \mathbf{if}\ p\,x \\ & & \mathbf{then}\ \ x :: \mathit{filter}\ p\ l' \\ & & \mathbf{else}\ \ \mathit{filter}\ p\ l' \end{array}$$

There is no type of the form

$$\mathit{filter} \quad \in \quad (\tau \to \mathbf{bool}) \to \Pi n{\in}\mathbf{nat}.\ \tau\ \mathbf{list}(n) \to \tau\ \mathbf{list}(?)$$

we can assign to *filter*, since the length of the result depends on p and the length of the input. For this we need an *existential type*. It works analogously to the existential quantifier on propositions and is written as $\Sigma x{\in}\tau.\ \sigma(x)$. With it, we would specify the type as

$$\mathit{filter} \quad \in \quad (\tau \to \mathbf{bool}) \to \Pi n{\in}\mathbf{nat}.\ \tau\ \mathbf{list}(n) \to \Sigma m{\in}\mathbf{nat}.\ \tau\ \mathbf{list}(m)$$

We can read this as "*the function returns a list of length m for some m*" or as "*the function returns a pair consisting of an m and a list of length m*", depending on whether we intend to carry the lengths are runtime. Before we show the specification of *filter* with this new type, we give the rules for Σ types.

$$\frac{\Gamma \vdash \tau \ \mathit{type} \qquad \Gamma, x{\in}\tau \vdash \sigma(x) \ \mathit{type}}{\Gamma \vdash \Sigma x{\in}\tau.\ \sigma(x)}\ \Sigma F$$

$$\frac{\Gamma \vdash t \in \tau \qquad \Gamma \vdash s \in \sigma(t)}{\Gamma \vdash \langle t, s\rangle \in \Sigma x{\in}\tau.\ \sigma(x)}\ \Sigma I$$

$$\frac{\Gamma \vdash t \in \Sigma x{\in}\tau.\ \sigma(x) \qquad \Gamma, x{\in}\tau, y{\in}\sigma(x) \vdash r \in \rho}{\Gamma \vdash (\mathbf{let}\ \langle x, y\rangle = t\ \mathbf{in}\ r) \in \rho}\ \Sigma E$$

If we read $\sigma(x)$ as a proposition $A(x)$ instead of a type, we obtain the usual rules for the existential quantifier. Returning to the function *filter*, we have

$$\mathit{filter} \in (\tau \to \mathbf{bool}) \to \Pi n{\in}\mathbf{nat}.\ \tau\ \mathbf{list}(n) \to \Sigma m{\in}\mathbf{nat}.\ \tau\ \mathbf{list}(m)$$

$$\begin{array}{llrrcl}
\mathit{filter} & p & \mathbf{0} & \mathbf{nil} & = & \langle \mathbf{0}, \mathbf{nil}\rangle \\
\mathit{filter} & p & (\mathbf{s}(n')) & (x :: l') & = & \mathbf{let}\ \langle m', k'\rangle = \mathit{filter}\ p\ n'\ l' \\
 & & & & & \mathbf{in}\ \mathbf{if}\ p\,x \\
 & & & & & \quad \mathbf{then}\ \langle \mathbf{s}(m'), x :: k'\rangle \\
 & & & & & \quad \mathbf{else}\ \langle m', k'\rangle
\end{array}$$

In this code, k' stands for the list resulting from the recursive call, and m' for its length. Now type-checking succeeds, since each branch in each case has type $\Sigma m{\in}\mathbf{nat}.\ \tau\ \mathbf{list}(m)$. Again, we can annotate the type and implementation to erase the part of the code which is not computationally relevant.

$$\mathit{filter} \in (\tau \to \mathbf{bool}) \to \Pi[n]{\in}[\mathbf{nat}].\ \tau\ \mathbf{list}[n] \to \Sigma[m]{\in}[\mathbf{nat}].\ \tau\ \mathbf{list}[m]$$

$$\begin{array}{llrrcl}
\mathit{filter} & p & [\mathbf{0}] & \mathbf{nil} & = & \langle [\mathbf{0}], \mathbf{nil}\rangle \\
\mathit{filter} & p & [\mathbf{s}(n')] & (x :: l') & = & \mathbf{let}\ \langle [m'], k'\rangle = \mathit{filter}\ p\ [n']\ l' \\
 & & & & & \mathbf{in}\ \mathbf{if}\ p\,x \\
 & & & & & \quad \mathbf{then}\ \langle [\mathbf{s}(m')], x :: k'\rangle \\
 & & & & & \quad \mathbf{else}\ \langle [m'], k'\rangle
\end{array}$$

This annotation is consistent, and erasure followed by simplification produces the previous version of *filter* with lists not carrying their length.

Existential types solve a number of potential problems, but they incur a loss of information which may render dependent type-checking less useful than it might first appear. Recall the function *rev*, the generalized version of *reverse* carrying an accumulator argument a.

$$\mathit{rev} \in \tau\ \mathbf{list} \to \tau\ \mathbf{list} \to \tau\ \mathbf{list}$$

$$\begin{array}{llrcl}
\mathit{rev} & \mathbf{nil} & a & = & a \\
\mathit{rev} & (x :: l') & a & = & \mathit{rev}\ l'\ (x :: a)
\end{array}$$

We would like to verify that

$$rev \quad \in \quad \Pi n{\in}\mathbf{nat}.\, \tau\, \mathbf{list}(n) \to \Pi m{\in}\mathbf{nat}.\, \tau\, \mathbf{list}(m) \to \tau\, \mathbf{list}(n+m)$$

where

$$\begin{array}{llllllll} rev & \mathbf{0} & \mathbf{nil} & m & a & = & a \\ rev & (\mathbf{s}(n')) & (x :: l') & m & a & = & rev\ n'\ l'\ (\mathbf{s}(m))\ (x :: a) \end{array}$$

While everything goes according to plan for the first equation, the second equation yields

$$rev\ (\mathbf{s}(n'))\ (x :: l')\ m\ a \quad \in \quad \tau\, \mathbf{list}(\mathbf{s}(n') + m)$$

for the left-hand side, and

$$rev\ n'\ l'\ (\mathbf{s}(m))\ (x :: a) \quad \in \quad \tau\, \mathbf{list}(n' + \mathbf{s}(m))$$

for the right hand side. There is no way to bridge this gap by comptutation alone; we need to *prove* that $\mathbf{s}(n')+m =_N n'+\mathbf{s}(m)$ by induction. Clearly, type-checking can not accomplish this—it would require type-checking to perform theorem proving which would not be feasible inside a compiler.

What can we do? One option is the simply hide the length of the output list by using an existential type.

$$rev \quad \in \quad \Pi n{\in}\mathbf{nat}.\, \tau\, \mathbf{list}(n) \to \Pi m{\in}\mathbf{nat}.\, \tau\, \mathbf{list}(m) \to \Sigma x{\in}\mathbf{nat}.\, \tau\, \mathbf{list}(x)$$

However, this means type-checking guarantees much less about our function than we might hope for. The other is to reintroduce propositions and change our type to something like

$$\begin{array}{rcl} rev & \in & \Pi n{\in}\mathbf{nat}.\, \tau\, \mathbf{list}(n) \to \Pi m{\in}\mathbf{nat}.\, \tau\, \mathbf{list}(m) \\ & & \quad \to \Sigma x{\in}\mathbf{nat}.\, [x =_N n+m] \times \tau\, \mathbf{list}(x). \end{array}$$

That is, we allow the output to be list of length x which is provably, but not necessarily computationally equal to the sum of n and m. Here we consider $[x =_N n + m]$ as a type, even though $x =_N n + m$ is a proposition. This is consistent with our interpretation of erasure, which converts propositions to types before running a program.

As a practical matter, in extensions of programming language with some limited form of dependent types, there are other ways to ensure feasibility of type-checking. Rather than base the comparison of types entirely on computation of the terms embedded in them, we can base it instead on any decidable theory (which is feasible in practice). This is the approach we have taken in the design of DML [XP98, XP99]. In the simplest application, index objects may contain only linear equalities and inequalities between integers, which can be solved effectively during type-checking. As we have seen in the examples above, dependent types (especially when we can also mention propositions $[A]$) permit a continuum of properties of programs to be expressed and verified at type-checking time, all the way from simple types to full specifications. For the

latter, the proof objects either have to be expressed directly in the program or extracted as obligations and verified separately.

We now briefly reexamine the Curry-Howard isomorphism, when extended to the first-order level. We have the following correspondence:

Propositions	$\wedge$	$\supset$	$\top$	$\vee$	$\bot$	$\forall$	$\exists$
Types	$\times$	$\rightarrow$	$\mathbf{1}$	$+$	$\mathbf{0}$	Π	Σ

Note that under erasure, $\forall$ is related to $\rightarrow$ and $\exists$ is related to $\times$. The analogous property holds for Π and Σ: $\Pi x{:}\tau.\ \sigma$ corresponds to $\tau \rightarrow \sigma$ if x does not occur in σ, and $\Sigma x{:}\tau.\ \sigma$ simplifies to $\tau \times \sigma$ if x does not occur in σ.

In view of this strong correspondence, one wonders if propositions are really necessary as a primitive concept. In some systems, they are introduced in order to distinguish those elements with computational contents from those without. However, we have introduced the bracket annotation to serve this purpose, so one can streamline and simplify type theory by eliminating the distinction between propositions and types. Similarly, there is no need to distinguish between terms and proof terms. In fact, we have already used identical notations for them. Propositional constructs such as $n =_{\mathbf{N}} m$ are then considered as types (namely: the types of their proof terms).

Because of the central importance of types and their properties in the design and theory of programming languages, there are many other constructions that are considered both in the literature and in practical languages. Just to name some of them, we have polymorphic types, singleton types, intersection types, union types, subtypes, record types, quotient types, equality types, inductive types, recursive types, linear types, strict types, modal types, temporal types, etc. Because of the essentially open-ended nature of type theory, all of these could be considered in the context of the machinery we have built up so far. We have seen most of the principles which underly the design of type systems (or corresponding logics), thereby providing a foundation for understanding the vast literature on the subject.

Instead of discussing these (which could be subject of another course) we consider one further application of dependent types and then consider theorem proving in various fragments of the full type theory.

4.9 Data Structure Invariants

An important application of dependent types is capturing representation invariants of data structures. An invariant on a data structure restricts valid elements of a type. Dependent types can capture such invariants, so that only valid elements are well-typed.

Our example will be an efficient implementation of finite sets of natural numbers. We start with a required lemma and auxiliary function.

$$\forall x{\in}\mathbf{nat}.\ \forall y{\in}\mathbf{nat}.\ [x < y] \vee [x =_{\mathbf{N}} y] \vee [x > y]$$

From the straightforward proof we can extract a function

$$\mathit{compare} \in \mathbf{nat} \to \mathbf{nat} \to \mathbf{1} + \mathbf{1} + \mathbf{1}.$$

For obvious reasons we use the abbreviations

$$\begin{array}{rcl} \mathit{less} & = & \mathbf{inl}\,\langle\rangle \\ \mathit{equal} & = & \mathbf{inr}\,(\mathbf{inl}\,\langle\rangle) \\ \mathit{greater} & = & \mathbf{inr}\,(\mathbf{inr}\,\langle\rangle) \end{array}$$

and

$$\begin{array}{l} \mathbf{case}\ r \\ \quad \mathbf{of}\ \mathit{less} \Rightarrow t_< \\ \quad\ \mid \mathit{equal} \Rightarrow t_= \\ \quad\ \mid \mathit{greater} \Rightarrow t_> \end{array} \quad = \begin{array}{l} \mathbf{case}\ r \\ \quad \mathbf{of}\ \mathbf{inl}\,\_ \Rightarrow t_< \\ \quad\ \mid \mathbf{inr}\,r' \Rightarrow (\mathbf{case}\ r' \\ \qquad\qquad \mathbf{of}\ \mathbf{inl}\,\_ \Rightarrow t_= \\ \qquad\qquad\ \mid \mathbf{inr}\,\_ \Rightarrow t_>) \end{array}$$

We give an interface for which we want to supply an implementation.

$$\begin{array}{rcl} \mathit{set} & & \mathit{type} \\ \mathit{empty} & \in & \mathit{set} \\ \mathit{insert} & \in & \mathbf{nat} \to \mathit{set} \to \mathit{set} \\ \mathit{member} & \in & \mathbf{nat} \to \mathit{set} \to \mathbf{bool} \end{array}$$

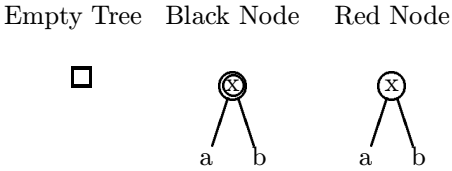
We do not give interfaces a first-class status in our development of type theory, but it is nonetheless a useful conceptual device. We would like to given an implementation via definitions of the form

$$\begin{array}{rcl} \mathit{set} & = & \ldots \\ \mathit{empty} & = & \ldots \\ \mathit{insert} & = & \ldots \\ \mathit{member} & = & \ldots \end{array}$$

that satisfy the types specified in the interface.

The idea is to implement a set as a *red-black tree*. Red-black trees are an efficient data structure for representing dictionaries whose keys are ordered. Here we follow the presentation by Chris Okasaki [Oka99]. The underlying data structure is a binary tree whose nodes are labelled with the members of the set. If we can ensure that the tree is sufficiently balanced, the height of such a tree will be logarithmic in the number of elements of the set. If we also maintain that the tree is ordered, lookup and insertion of an element can be performed in time proportional to the logarithm of the size of the set. The mechanism for ensuring that the tree remains balanced is the coloring of the nodes and the invariants maintained in the representation.

A tree is either empty or consists of a black or red node labelled with a natural number x and two subtrees a and b



We maintain the following representation invariants:

1. The tree is *ordered*: all elements in the left subtree a are smaller than x, while all elements in the right subtree b are larger than x.
2. The tree is *uniform*: every path from the root to a leaf contains the same number of black nodes. This defines the *black height* of a tree, where the black height of the empty tree is taken to be zero.
3. The tree is *well-colored*: the children of red node are always black, where empty trees count as black.

Uniform and well-colored trees are sufficiently balanced to ensure worst-case logarithmic membership test for elements in the set. Other operations can be implemented with similar efficiency, but we concentrate on membership test and insertion.

The ordering invariant is difficult to enforce by dependent types, since it requires propositional reasoning about the less-than relation. We will capture the uniformity invariant via dependent types. It is also possible to capture the coloring invariant via dependencies, but this is more complicated, and we do not attempt it here.

We index a red-black tree by its black height.

$$\frac{\Gamma \vdash n \in \mathbf{nat}}{\Gamma \vdash \mathit{tree}(n)\ \mathit{type}}\ \mathit{tree}F$$

There are three introduction rules, incorporating the three types of nodes (empty, black, red).

$$\frac{}{\Gamma \vdash \mathbf{E} \in \mathit{tree}(\mathbf{0})}\ \mathit{tree}I_E$$

$$\frac{\Gamma \vdash a \in \mathit{tree}(n) \qquad \Gamma \vdash x \in \mathbf{nat} \qquad \Gamma \vdash b \in \mathit{tree}(n)}{\Gamma \vdash \mathbf{B}\ a\ x\ b \in \mathit{tree}(\mathbf{s}(n))}\ \mathit{tree}I_B$$

$$\frac{\Gamma \vdash a \in \mathit{tree}(n) \qquad \Gamma \vdash x \in \mathbf{nat} \qquad \Gamma \vdash b \in \mathit{tree}(n)}{\Gamma \vdash \mathbf{R}\ a\ x\ b \in \mathit{tree}(n)}\ \mathit{tree}I_R$$

The index is increased by one for a black node **B**, but not for a red node **R**. Note that in either case, both subtrees a and b must have the same black height. This

use of indices is different from their use for lists. Any list formed from **nil** and cons (::) without the length index will in fact have a valid length. Here, there are many trees that are ruled out as invalid because of the dependent types. In other words, the dependent types guarantee a data structure invariant by type-checking alone.

Now we can begin filling in the implementation, according to the given interface.

$$\begin{array}{rcl} \mathit{set} & = & \Sigma n{\in}\mathbf{nat}.\ \mathit{tree}(n) \\ \mathit{empty} & = & \langle \mathbf{0}, \mathbf{E} \rangle \\ \mathit{insert} & = & \ldots \\ \mathit{member} & = & \ldots \end{array}$$

Our intent is not to carry the black height n at run-time. If we wanted to make this explicit, we would write $\Sigma[n]{\in}[\mathbf{nat}].\ \mathit{tree}[n]$.

Before we program the *insert* and *member* functions, we write out the elimination form as a schema of primitive recursion.

$$\begin{array}{rcl} f(\mathbf{0}, \mathbf{E}) & = & t_E \\ f(\mathbf{s}(n'), \mathbf{B}\ a\ x\ b) & = & t_B(n', a, x, b, f(n', a), f(n', b)) \\ f(n, \mathbf{R}\ a\ x\ b) & = & t_R(n, a, x, b, f(n, a), f(n, b)) \end{array}$$

Using this schema, we can define the set membership function.

$$\mathit{mem} \ : \ \mathbf{nat} \to \Pi n{\in}\mathbf{nat}.\ \mathit{tree}(n) \to \mathbf{bool}$$

$$\begin{array}{llllcl} \mathit{mem} & x & \mathbf{0} & \mathbf{E} & = & \mathit{false} \\ \mathit{mem} & x & (\mathbf{s}(n')) & (\mathbf{B}\ a\ y\ b) & = & \mathbf{case}\ \mathit{compare}\ x\ y \\ & & & & & \quad \mathbf{of}\ \mathit{less} \Rightarrow \mathit{mem}\ x\ n'\ a \\ & & & & & \quad \mid \mathit{equal} \Rightarrow \mathit{true} \\ & & & & & \quad \mid \mathit{greater} \Rightarrow \mathit{mem}\ x\ n'\ b \\ \mathit{mem} & x & n & (\mathbf{R}\ a\ y\ b) & = & \mathbf{case}\ \mathit{compare}\ x\ y \\ & & & & & \quad \mathbf{of}\ \mathit{less} \Rightarrow \mathit{mem}\ x\ n\ a \\ & & & & & \quad \mid \mathit{equal} \Rightarrow \mathit{true} \\ & & & & & \quad \mid \mathit{greater} \Rightarrow \mathit{mem}\ x\ n\ b \end{array}$$

Note that the cases for black and red nodes are identical, except for their treatment of the indices. This is the price we have to pay for our representation. However, in practice this can be avoided by allowing some type inference rather than just type checking.

From *mem* we can define the *member* function:

$$\begin{array}{rcl} \mathit{member} & \in & \mathbf{nat} \to \mathit{set} \to \mathbf{bool} \\ \mathit{member} & = & \lambda x{\in}\mathbf{nat}.\ \lambda s{\in}\mathit{set}.\ \mathbf{let}\ \langle n, t \rangle = s\ \mathbf{in}\ \mathit{mem}\ x\ n\ t \end{array}$$

Insertion is a much trickier operation. We have to temporarily violate our color invariant and then restore it by a re-balancing operation. Moreover, we sometimes need to increase the black height of the tree (essentially, when we run out of room at the current level). We need an auxiliary function

$$\mathit{ins} \quad \in \quad \mathbf{nat} \to \Pi n{\in}\mathbf{nat}.\ \mathit{tree}(n) \to \mathit{tree}(n)$$

which preserves the black height n, but may violate the red-black invariant at the root. That is, the resulting tree must be a valid red-black tree, except that the root might be red and either the left or the right subtree could also have a red root. At the top-level, we re-establish the color invariant by re-coloring the root black. We first show this step, assuming a function *ins* according to our specification above. Recall that $set = \Sigma n{\in}\mathbf{nat}.\ tree(n)$

$$
\begin{array}{lrrcl}
\multicolumn{5}{l}{recolor \in \Pi n{\in}\mathbf{nat}.\ tree(n) \to set} \\
recolor & \mathbf{0} & \mathbf{E} & = & \langle \mathbf{0}, \mathbf{E} \rangle \\
recolor & (\mathbf{s}(n')) & (\mathbf{B}\ a\ x\ b) & = & \langle \mathbf{s}(n'), \mathbf{B}\ a\ x\ b \rangle \\
recolor & n & (\mathbf{R}\ a\ x\ b) & = & \langle \mathbf{s}(n), \mathbf{B}\ a\ x\ b \rangle
\end{array}
$$

$$
\begin{array}{lcl}
insert & \in & \mathbf{nat} \to set \to set \\
insert & = & \lambda x \in \mathbf{nat}.\ \lambda s \in set. \\
 & & \quad \mathbf{let}\ \langle n, t \rangle = s\ \mathbf{in}\ recolor\ n\ (ins\ x\ t)
\end{array}
$$

Note that *recolor* returns a tree of black height n if the root node is black, and $\mathbf{s}(n)$ if the root node is red. This is how the black height can grow after successive insertion operations.

Now to the auxiliary function *ins*. Recall:

$$ins \quad \in \quad \mathbf{nat} \to \Pi n{\in}\mathbf{nat}.\ tree(n) \to tree(n)$$

It is critical that the black height of the output tree is the same as the input tree, so that the overall balance of the tree is not compromised during the recursion. This forces, for example, the case of insertion into an empty tree to color the new node red.

$$
\begin{array}{llrrcl}
ins & x & \mathbf{0} & \mathbf{E} & = & \mathbf{R}\ \mathbf{E}\ x\ \mathbf{E} \\
ins & x & (\mathbf{s}(n')) & (\mathbf{B}\ a\ y\ b) & = & \mathbf{case}\ compare\ x\ y \\
 & & & & & \quad \mathbf{of}\ less \Rightarrow balance_L\ n'\ (ins\ x\ n'\ a)\ y\ b \\
 & & & & & \quad\ \mid equal \Rightarrow \mathbf{B}\ a\ y\ b \\
 & & & & & \quad\ \mid greater \Rightarrow balance_R\ n'\ a\ y\ (ins\ x\ n'\ a) \\
ins & x & n & (\mathbf{R}\ a\ y\ b) & = & \mathbf{case}\ compare\ x\ y \\
 & & & & & \quad \mathbf{of}\ less \Rightarrow \mathbf{R}\ (ins\ x\ n\ a)\ y\ b \\
 & & & & & \quad\ \mid equal \Rightarrow \mathbf{R}\ a\ y\ b \\
 & & & & & \quad\ \mid greater \Rightarrow \mathbf{R}\ a\ y\ (ins\ x\ n\ a)
\end{array}
$$

We need two auxiliary functions $balance_L$ and $balance_R$ to repair any possible violation of the color invariant in either the left or right subtree, in case the root

node is black.

$$balance_L \in \Pi n' \in \mathbf{nat}.\ tree(n') \to \mathbf{nat} \to tree(n') \to tree(\mathbf{s}(n'))$$

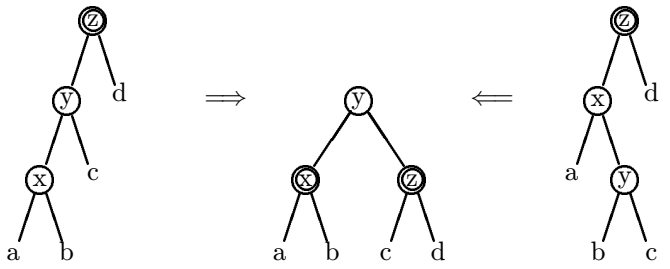
$$\begin{array}{rcll} balance_L\ n'\ (\mathbf{R}\ (\mathbf{R}\ a\ x\ b)\ y\ c)\ z\ d & = & \mathbf{B}\ (\mathbf{R}\ a\ x\ b)\ y\ (\mathbf{R}\ c\ z\ d) & \\ balance_L\ n'\ (\mathbf{R}\ a\ x\ (\mathbf{R}\ b\ y\ c))\ z\ d & = & \mathbf{B}\ (\mathbf{R}\ a\ x\ b)\ y\ (\mathbf{R}\ c\ z\ d) & \\ balance_L\ n'\ a\ x\ b & = & \mathbf{B}\ a\ x\ b & \text{in all other cases} \end{array}$$

$$balance_R \in \Pi n' \in \mathbf{nat}.\ tree(n') \to \mathbf{nat} \to tree(n') \to tree(\mathbf{s}(n'))$$

$$\begin{array}{rcll} balance_R\ n'\ a\ x\ (\mathbf{R}\ (\mathbf{R}\ b\ y\ c)\ z\ d) & = & \mathbf{B}\ (\mathbf{R}\ a\ x\ b)\ y\ (\mathbf{R}\ c\ z\ d) & \\ balance_R\ n'\ a\ x\ (\mathbf{R}\ b\ y\ (\mathbf{R}\ c\ z\ d)) & = & \mathbf{B}\ (\mathbf{R}\ a\ x\ b)\ y\ (\mathbf{R}\ c\ z\ d) & \\ balance_R\ n'\ a\ x\ b & = & \mathbf{B}\ a\ x\ b & \text{in all other cases} \end{array}$$

We have taken the liberty of combining some cases to significantly simplify the specification. It should be clear that this can indeed be implemented. In fact, there is no recursion, only several nested case distinctions.

The following picture illustrates the operation performed by $balance_L$. Note that the tree input trees to the left and the right are never actually built, but that $balance_L$ directly receives the left subtree, z and d as arguments.



Type-checking will verify that the black-height remains invariant under the balancing operation: initially, it is n' for each subtree a, b, c, and d and $\mathbf{s}(n')$ for the whole tree, which is still the case after re-balancing.

Similarly, the order is preserved. Writing $t < x$ to mean that every element in tree t is less than x, we extract the order

$$a < x < b < y < c < z < d$$

from all three trees by traversing it in a "smallest first" fashion.

Finally, we can see that the tree resulting from balancing always satisfies the red-black invariant, if the pictures on the left and right indicate the *only* place where the invariant is violated before we start.

All these proofs can be formalized, using an appropriate formalization of these color and ordering invariants. The only important consideration we have not mentioned is that in the case of insertion into a tree with a red root, we do not to apply the re-balancing operation to the result. This is because (a)

the two immediate subtrees must be black, and (b) inserting into a black tree always yields a valid red-black tree (with *no* possible violation at the root).

This example illustrates how dependent types can be used to enforce a continuum of properties via type-checking, while others are left to explicit proof. From the software engineering point of view, any additional invariants that can be enforced statically without an explosion in the size of the program is likely to be beneficial by catching programming errors early.

Chapter 5

Decidable Fragments

In previous chapters we have concentrated on the basic laws of reasoning and their relationship to types and programming languages. The logics and type theories we considered are very expressive. This is important in many applications, but it has the disadvantage that the question if a given proposition is true is undecidable in general. That is, there is no terminating mechanical procedure which, given a proposition, will tell us whether it is true or not. This is true for first-order logic, arithmetic, and more complex theories such as the theory of lists. Furthermore, the proof that no decision procedure exists (which we do not have time to consider in this course), does not depend on whether we allow the law of excluded middle or not.

In programming language application, we can sometimes work around this limitation, because we are often not interested in theorem proving, but in type-checking. That is, we are given a program (which corresponds to a proof) and a type (which corresponds to a proposition) and we have to check its validity. This is a substantially easier problem than deciding the truth of a proposition—essentially we have to verify the correctness of the applications of inference rules, rather than to guess which rules might have been applied.

However, there are a number of important applications where we would like to solve the *theorem proving problem*: given a proposition, is it true? This can come about either directly (verify a logic property of a program or system) or indirectly (take a general problem and translate it into logic).

In this chapter we consider such applications of logic, mostly to problems in computer science. We limit ourselves to fragments of logic that can be mechanically decided, that is, there is a terminating algorithm which decides whether a given proposition is true or not. This restricts the set of problems we can solve, but it means that in practice we can often obtain answers quickly and reliably. It also means that in principle these developments can be carried out within type theory itself. We demonstrate this for our first application, based on quantified Boolean formulas.

The material here is not intended as an independent introduction, but complements the treatment by Huth and Ryan [HR00].

5.1 Quantified Boolean Formulas

In Section 3.6 we have introduced the data type **bool** of Booleans with two elements, **true** and **false**. We have seen how to define such operations as boolean negation, conjunction, and disjunction using the elimination rule for this type which corresponds to an if-then-else construct. We briefly review these constructs and also the corresponding principle of proof by cases. In accordance with the notation in the literature on this particular subject (and the treatment in Huth and Ryan [HR00]), we write 0 for **false**, 1 for **true**, $b \cdot c$ for *and*, $b + c$ for *or*, and $\overline{b}$ for *not*.

The Boolean type, **bool**, is defined by two introduction rules.

$$\frac{}{\mathbf{bool}\ \mathit{type}}\ \mathbf{bool}F$$

$$\frac{}{\Gamma \vdash 0 \in \mathbf{bool}}\ \mathbf{bool}I_0 \qquad \frac{}{\Gamma \vdash 1 \in \mathbf{bool}}\ \mathbf{bool}I_1$$

The elimination rule distinguishes the two cases for a given Boolean value.

$$\frac{\Gamma \vdash b \in \mathbf{bool} \qquad \Gamma \vdash s_1 \in \tau \qquad \Gamma \vdash s_0 \in \tau}{\Gamma \vdash \mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_0 \in \tau}\ \mathbf{bool}E$$

The reduction rules just distinguish the two cases for the subject of the **if**-expression.

$$\begin{array}{rcl} \mathbf{if}\ 0\ \mathbf{then}\ s_1\ \mathbf{else}\ s_0 & \Longrightarrow & s_0 \\ \mathbf{if}\ 1\ \mathbf{then}\ s_1\ \mathbf{else}\ s_0 & \Longrightarrow & s_1 \end{array}$$

Now we can define typical functions on booleans, such as *and*, *or*, and *not*, transcribing their truth tables. We make no attempt here to optimize the definitions, but simply distinguish all possible cases for the inputs.

$$\begin{array}{rcl} x \cdot y & = & \mathbf{if}\ x\ \mathbf{then}\ (\mathbf{if}\ y\ \mathbf{then}\ 1\ \mathbf{else}\ 0)\ \mathbf{else}\ (\mathbf{if}\ y\ \mathbf{then}\ 0\ \mathbf{else}\ 0) \\ x + y & = & \mathbf{if}\ x\ \mathbf{then}\ (\mathbf{if}\ y\ \mathbf{then}\ 1\ \mathbf{else}\ 1)\ \mathbf{else}\ (\mathbf{if}\ y\ \mathbf{then}\ 1\ \mathbf{else}\ 0) \\ \overline{x} & = & \mathbf{if}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ 1 \end{array}$$

Following this line of thought, it is quite easy to define universal and existential quantification over Booleans. The idea is that $\forall x{\in}\mathbf{bool}.\ f(x)$ where f is a Boolean term dependent on x is representated by $\mathit{forall}(\lambda x{\in}\mathbf{bool}.\ f(x))$ so that

$$\begin{array}{rcl} \mathit{forall} & \in & (\mathbf{bool} \to \mathbf{bool}) \to \mathbf{bool} \\ \mathit{forall} & = & \lambda f{\in}\mathbf{bool} \to \mathbf{bool}.\ f\ 0 \cdot f\ 1 \end{array}$$

Somewhat more informally, we write

$$\forall x.\ f(x) \quad = \quad f(0) \cdot f(1)$$

but this should only be considered a shorthand for the above definition. The existential quantifier works out similarly, replacing conjunction by disjunction. We have

$$\begin{array}{rcl} \mathit{exists} & \in & (\mathbf{bool} \to \mathbf{bool}) \to \mathbf{bool} \\ \mathit{exists} & = & \lambda f{\in}\mathbf{bool} \to \mathbf{bool}.\ f\ 0 + f\ 1 \end{array}$$

or, in alternate notation,

$$\exists x.\, f(x) \quad = \quad f(0) + f(1)$$

The resulting language (formed by $0, 1, +, \cdot, \bar{}, \forall, \exists$) is that of *quantified Boolean formulas*. As the definitions above show, the value of each quantified Boolean formula (without free variables) can simply be computed, using the definitions of the operations in type theory.

Unfortunately, such a computation is extremely inefficient, taking exponential time in the number of quantified variables, not only in the worst, but the typical case. Depending on the operational semantics we employ in type theory, the situation could be even worse, require exponential time in every case.

There are two ways out of this dilemma. One is to leave type theory and give an efficient imperative implementation using, for example, ordered binary decision diagrams as shown in Huth and Ryan. While this does not improve worst-case complexity, it is practical for a large class of examples.

Another possibility is to replace computation by reasoning. Rather than *computing* the value of an expression, we *prove* that it is equal to 1 or 0. For example, it is easy to prove that $(\forall x_1 \ldots \forall x_n.\ x_1 \cdots x_n \cdot 0) =_B 0$ even though it may take exponential time to compute the value of the left-hand side of the equation. In order to model such reasoning, we need the propositional counterpart of the elimination rule for **bool**.

$$\frac{\Gamma \vdash b \in \mathbf{bool} \qquad \Gamma \vdash M_1 : A(1) \qquad \Gamma \vdash M_0 : A(0)}{\Gamma \vdash \mathbf{case}\ b\ \mathbf{of}\ 1 \Rightarrow M_1 \mid 0 \Rightarrow M_0 : A(t)}\ \mathbf{bool}E$$

The rules for propositional equality between Booleans follow the pattern established by equality on natural numbers and lists.

$$\frac{\Gamma \vdash b \in \mathbf{nat} \qquad \Gamma \vdash c \in \mathbf{nat}}{\Gamma \vdash b =_B c\ prop} =_B F$$

$$\frac{}{\Gamma \vdash 0 =_B 0\ true} =_B I_0 \qquad \frac{}{\Gamma \vdash 1 =_B 1\ true} =_B I_1$$

$$no =_B E_{00}\ \textit{elimination rule} \qquad no =_B E_{11}\ \textit{elimination rule}$$

$$\frac{\Gamma \vdash 0 =_B 1\ true}{\Gamma \vdash C\ true} =_B E_{01} \qquad \frac{\Gamma \vdash 1 =_B 0\ true}{\Gamma \vdash C\ true} =_B E_{10}$$

As a simple example, we prove that for every $b \in \mathbf{bool}$ we have $b \cdot 0 =_B 0$. The proof proceeds by cases on b.

Case: $b = 0$. Then we compute $0 \cdot 0 =_B 0$ so by conversion we have $0 \cdot 0 =_N 0$.

Case: $b = 1$. Then we compute $1 \cdot 0 =_B 0$ so by conversion we have $1 \cdot 0 =_N 1$

Note that we can use this theorem for arbitrarily complex terms b. Its transcription into a formal proof using the rules above is straightforward and omitted.

An interesting proposal regarding the combination of efficient computation (using OBDDs) and proof has been made by Harrison [Har95]. From a trace of the operation of the OBDD implementation we can feasibly extract a proof in terms of the primitive inference rules and some other derived rules. This means what we can have a complex, optimizing implementation without giving up the safety of proof by generating a proof object rather than just a yes-or-no answer.

5.2 Boolean Satisfiability

A particularly important problem is Boolean satisfiability (SAT):

> Given a Boolean formula $\exists x_1 \ldots \exists x_n.\ f(x_1, \ldots, x_n)$ without free variables where $f(x_1, \ldots, x_n)$ does not contain quantifiers. Is the formula equal to 1?

Alternatively, we can express this as follows:

> Given a quantifier-free Boolean formula $f(x_1, \ldots, x_n)$, is there an assignment of 0 and 1 to each of the variables x_i which makes f equal to 1?

SAT is an example of an NP-complete problem: it can be solved in nondeterministic polynomial time (by guessing and then checking the satisfying assignment), and every other problem in NP can be translated to SAT in polynomial time. What is perhaps surprising is that this can be practical in many cases. The dual problem of validity (deciding if $\forall x_1 \ldots \forall x_n.\ f(x_1, \ldots, x_n)$ is equal to 1) is also often of interest and is co-NP complete.

There are many graph-based and related problems which can be translated into SAT. As a simple example we consider the question of deciding whether the nodes of a graph can be colored with k colors such that no two nodes that are connected by an edge have the same color.

Assume we are given a graph with nodes numbered $1, \ldots, n$ and colors $1, \ldots, k$. We introduce Boolean variables c_{ij} with the intent that

$$c_{ij} = 1 \quad \text{iff} \quad \text{node } i \text{ has color } j$$

We now have to express that constraints on the colorings by Boolean formulas. First, each node as exactly one color. For each node $1 \leq i \leq n$ we obtain a formula

$$\begin{aligned} u_i \quad = \quad & c_{i1} \cdot \overline{c_{i2}} \cdots \overline{c_{ik}} \\ & + \overline{c_{i1}} \cdot c_{i2} \cdots \overline{c_{ik}} \\ & + \cdots \\ & + \overline{c_{i1}} \cdot \overline{c_{i2}} \cdots c_{ik} \end{aligned}$$

There are n such formulas, each of size $O(n \times k)$. Secondly we want to express that two nodes connected by an edge do not have the same color. For any two

nodes i and m we have

$$\begin{array}{lcll} w_{im} & = & \overline{c_{i1} \cdot c_{m1}} \cdots \overline{c_{ik} \cdot c_{mk}} & \text{if there is an edge between } i \text{ and } m \\ w_{im} & = & 1 & \text{otherwise} \end{array}$$

There are $n \times n$ such formulas, each of size $O(k)$ or $O(1)$. The graph can be colored with k colors if each of the u_i and w_{im} are satisfied simultaneously. Thus the satisfiability problem associated with a graph is

$$(u_1 \cdots u_n) \cdot (w_{11} \cdots w_{1n}) \cdots (w_{n1} \cdots w_{nn})$$

The total size of the resulting formula is $O(n^2 \times k)$ and contains $n \times k$ Boolean variables. Thus the translation is clearly polynomial.

5.3 Constructive Temporal Logic

An important application of logic is model-checking, as explained in Huth and Ryan. Another excellent source on this topic is the book by Clarke, Grumberg and Peled [CGP99].

Here we give a constructive development of a small fragment of *Computation Tree Logic* (CTL). I am not aware of any satisfactory constructive account for all of CTL[1].

In order to model temporal logic we need to relativize our main judgment A *true* to particular states. We have the following judgments:

s *state*	s is a state
$s \to s'$	we can transition from state s to s' in one step
$A \mathbin{@} s$	proposition A is true in state s

We presuppose that s and s' are states when we write $s \to s'$ and that A is a proposition and s a state when writing $A \mathbin{@} s$.

Now all logical rules are viewed as rules for reasoning entirely within a given state. For example:

$$\frac{A \mathbin{@} s \qquad B \mathbin{@} s}{A \wedge B \mathbin{@} s} \wedge I$$

$$\frac{A \wedge B \mathbin{@} s}{A \mathbin{@} s} \wedge E_L \qquad\qquad \frac{A \wedge B \mathbin{@} s}{B \mathbin{@} s} \wedge E_R$$

For disjunction and falsehood elimination, there are two choices, depending on whether we admit conclusions in an arbitrary state s' or only in s, the state in which we have derived the disjunction or falsehood. It would seem that both choices are consistent and lead to slightly different logics.

[1] For linear time temporal logic, Davies [Dav96] gives an extension of the Curry-Howard isomorphism with an interesting application to *partial evaluation*

Next, we model the AX and EX connectives. AX A is true in state s if A is true in every successor state s'. To express "in every successor state" we introduce the assumption $s \to S'$ for a new *parameter* S'.

$$\dfrac{\dfrac{}{s \to S'}\,u \quad \vdots \quad A \mathbin{@} S'}{\mathrm{AX}\ A \mathbin{@} s}\,\mathrm{AX}I^{S',u}$$

The elimination rule allows us to infer that A is true in state s' if AX A is true in s, and s' is a successor state to s.

$$\dfrac{\mathrm{AX}\ A \mathbin{@} s \qquad s \to s'}{A \mathbin{@} s'}\,\mathrm{AX}E$$

It is easy to see that this elimination rule is locally sound.

The rules for the EX connective follow from similar considerations.

$$\dfrac{s \to s' \quad A \mathbin{@} s'}{\mathrm{EX}\ A \mathbin{@} s}\,\mathrm{EX}I$$

$$\dfrac{\mathrm{EX}\ A \mathbin{@} s \qquad \dfrac{}{s \to S'}\,u \quad \dfrac{}{A \mathbin{@} S'}\,w \quad \vdots \quad C \mathbin{@} r}{C \mathbin{@} r}\,\mathrm{EX}E^{S',u,w}$$

We can now prove general laws, such as

$$\mathrm{AX}\ (A \wedge B) \equiv (\mathrm{AX}\ A) \wedge (\mathrm{AX}\ B)$$

Such proofs are carried out parametrically in the sense that we do not assume any particular set of states or particular transition relations. Laws derived in this manner will be true for any particular set of states and transitions.

If we want to reason about a particular system (which is done in model-checking), we have to specify the atomic propositions, states, and transitions. For example, the system on page 157 in Huth and Ryan is represented by the assumptions

$$\begin{array}{l} p\ \mathit{prop}, q\ \mathit{prop}, r\ \mathit{prop}, \\ s_0\ \mathit{state}, s_1\ \mathit{state}, s_2\ \mathit{state}, \\ s_0 \to s_1, s_0 \to s_2, \\ s_1 \to s_0, s_1 \to s_2, \\ s_2 \to s_2, \\ p \mathbin{@} s_0, q \mathbin{@} s_0, \neg r \mathbin{@} s_0, \\ \neg p \mathbin{@} s_1, q \mathbin{@} s_1, r \mathbin{@} s_1, \\ \neg p \mathbin{@} s_2, \neg q \mathbin{@} s_2, r \mathbin{@} s_1 \end{array}$$

Unfortunately, this is not yet enough. We can think of the transition rules above as introduction rules for the $s \to s'$, but we also need elimination rules. Because of the nature of the AX and EX connectives, it appears sufficient if we can distinguish cases on the target of a transition.

$$\frac{s_0 \to s \quad A \mathbin{@} s_1 \quad A \mathbin{@} s_2}{A \mathbin{@} s}\, s_0 \to E$$

$$\frac{s_1 \to s \quad A \mathbin{@} s_0 \quad A \mathbin{@} s_2}{A \mathbin{@} s}\, s_1 \to E$$

$$\frac{s_2 \to s \quad A \mathbin{@} s_2}{A \mathbin{@} s}\, s_2 \to E$$

In general, this would have to be augmented with additional rules, for example, letting is infer anything from an assumption that $s_2 \to s_1$ if there is in fact no such transition.[2] Now we can prove, for example, that $\mathrm{AX}\, r \mathbin{@} s_0$ as follows

$$\dfrac{\dfrac{\overline{s_0 \to S}\, u \quad \overline{r \mathbin{@} s_1} \quad \overline{r \mathbin{@} s_2}}{r \mathbin{@} S}\, s_1 \to E}{\mathrm{AX}\, r \mathbin{@} s_0}\, \mathrm{AX}I^{S,u}$$

Despite the finiteness of the sets of states and the transition relations, the logic presented here is different from the classical formulation of CTL usually used, because we do not assume the law of excluded middle. Of course, this can be done which brings us back to the usual interpretation of the logic.

It is not clear how to carry this constructive development forward to encompass other connectives such as AG, AF, EG, etc. The difficulty here is that paths are infinite, yet we need to reason about global or eventual truth along such paths. In the classical development this is easily handled by introducing appropriate laws for negation and least and greatest fixpoint operations on monotone state transformations. We are currently investigating type-theoretic expressions of this kind of reasoning using inductive and co-inductive techniques.

[2] For the connectives given here, I do not believe that this is necessary.

Bibliography

[CGP99] E.M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 1999.

[CR36] Alonzo Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, May 1936.

[Dav96] Rowan Davies. A temporal logic approach to binding-time analysis. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

[Har95] John Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38:162–170, 1995.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Hitherto unpublished note of 1969, rearranged, corrected, and annotated by Howard.

[HR00] Michael R.A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2000.

[ML80] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

[ML96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[Oka99] Chris Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, July 1999.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 249–257, Montreal, Canada, June 1998. ACM Press.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, January 1999.