Introduction to Computational Logic

Lecture Notes SS 2008

July 14, 2008

Gert Smolka and Chad E. Brown
Department of Computer Science
Saarland University

Contents

1 Introduction

2 Structure of Mathematical Statements

We outline a language for expressing mathematical statements. It employs functions as it main means of expression. Of particular importance are higher-order functions taking functions as arguments. The language has much in common with functional programming languages such as ML and Haskell. It is formal in the sense that it can be realized on a computer. We distinguish between the notational, syntactic and semantic level of the language.

2.1 Functions and Lambda Notation

Everyone knows that a function is something that takes an argument and yields a result. Nowadays, functions are defined as sets of pairs. If $(x, y) \in f$, then the application of the function f to the argument x yields the result y. Here is the precise definition.

Let X and Y be sets. A **function** $X \to Y$ is a set $f \subseteq X \times Y$ such that

1. $\forall x \in X\ \exists y \in Y\colon\ (x, y) \in f$
2. $(x, y) \in f \ \wedge\ (x, y') \in f \implies y = y'$

We use $X \to Y$ to denote the set of all functions $X \to Y$. If $f \in X \to Y$ and $x \in X$, we write fx for the unique y such that $(x, y) \in f$. We write $fx + 5$ for $(fx) + 5$.

The canonical means for describing functions is the **lambda notation** developed by the American logician Alonzo Church [9]. Here is an example:

$$\lambda x{\in}\mathbb{Z}.\ x^2$$

This notation describes the function $\mathbb{Z} \to \mathbb{Z}$ that squares its argument (i.e., yields x^2 for x). The following equation holds:

$$(\lambda x{\in}\mathbb{Z}.\ x^2) = \{\,(x, x^2) \mid x \in \mathbb{Z}\,\}$$

It shows the analogy between the lambda notation and the more common set notation.

According to our definition, functions take a single argument. To represent operations with more than one argument (i.e., addition of two numbers), one often uses functions that are applied to tuples $(x_1, \dots, x_n)$ that contain the arguments $x_1, \dots, x_n$ for the operation. We call such functions **cartesian** and speak of the **cartesian representation** of an operation. The cartesian function representing addition of integers has the type $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. It takes a pair (x, y) as single argument and returns the number $x + y$.

Functions that return functions as results are called **cascaded**. Lambda notation makes it easy to describe **cascaded functions**. As example, consider the definition

$$plus \;:=\; \lambda x{\in}\mathbb{Z}.\, \lambda y{\in}\mathbb{Z}.\, x + y$$

which binds the name *plus* to a function of type $\mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z})$. When we apply *plus* to an argument a, we obtain a function $\mathbb{Z} \to \mathbb{Z}$. When we apply this function to an argument b, we get $a + b$ as result. With symbols:

$$(plus\, a)\, b \;=\; ((\lambda x{\in}\mathbb{Z}.\, \lambda y{\in}\mathbb{Z}.\, x + y)\, a)\, b \;=\; (\lambda y{\in}\mathbb{Z}.\, a + y)\, b \;=\; a + b$$

We say that *plus* is a **cascaded representation** of the addition operation for integers. Cascaded representations are often called Curried representations, after the logician Haskell Curry. The idea goes back to Frege [14]. It was fully developed in a paper by Moses Schönfinkel [35] on the primitives of mathematical language. We prefer the cascaded representation over the cartesian representation since the cascaded representation doesn't require tuples. Following common practice, we omit parentheses as follows:

$$\begin{aligned} fxy \quad &\rightsquigarrow \quad (fx)y \\ X \to Y \to Z \quad &\rightsquigarrow \quad X \to (Y \to Z) \end{aligned}$$

Using this convenience, we can write $plus\ 3\ 7 = 10$ and $plus \in \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$.

2.2 Boolean Operations

We use the numbers 0 and 1 as **Boolean values,** where 0 may be thought of as “false” and 1 as “true”. We define

$$\mathbb{B} \;:=\; \{0, 1\}$$

As in programming languages, we adopt the convention that expressions like $3 \le x$ yield a Boolean value. This explains the following equations:

$$\begin{aligned} &(3 < 7) = 1 \\ &(7 \le 3) = 0 \\ &(3 = 7) = 0 \end{aligned}$$

The following equations define well-known Boolean operations:

$$\begin{aligned}
\neg x &= 1 - x && \textbf{Negation}\\
(x \wedge y) &= \min\{x, y\} && \textbf{Conjunction}\\
(x \vee y) &= \max\{x, y\} && \textbf{Disjunction}\\
(x \rightarrow y) &= (\neg x \vee y) && \textbf{Implication}\\
(x \equiv y) &= (x = y) && \textbf{Equivalence}
\end{aligned}$$

We represent Boolean operations as functions. Negation is a function $\mathbb{B} \rightarrow \mathbb{B}$ and the binary operations are functions $\mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}$. We use the symbols $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\equiv$ as names for these functions.

Exercise 2.2.1 Consider the values

$$\begin{aligned}
0, 1 &\in \mathbb{B}\\
\neg &\in \mathbb{B} \rightarrow \mathbb{B}\\
\wedge, \vee, \rightarrow, \equiv &\in \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}
\end{aligned}$$

With 0 and $\rightarrow$ one can express 1 as follows: $1 = (0 \rightarrow 0)$.

a) Express $\neg$, $\wedge$, $\vee$, and $\equiv$ with 0 and $\rightarrow$.

b) Express 0, $\wedge$, and $\rightarrow$ with 1, $\neg$, and $\vee$.

2.3 Operator Precedence

An operator is a symbol for a binary operation. There are established conventions that make it possible to write operator applications without parentheses. For example:

$$3 \cdot x + y \quad \rightsquigarrow \quad (3 \cdot x) + y$$

The symbols $+$ and $\cdot$ are said to be **infix operators**, and the operator $\cdot$ is said to take its arguments before the operator $+$. We assume the following **precedence hierarchy** for some commonly used operators:

$$\begin{array}{c}
\equiv\\
\rightarrow\\
\vee\\
\wedge\\
\neg\\
= \quad < \quad \leq \quad > \quad \geq\\
+ \quad -\\
\cdot
\end{array}$$

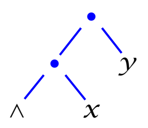
Symbols appearing lower in the hierarchy take their arguments before symbols appearing higher in the hierarchy. Here are examples of notations that omit parentheses according to the precedence hierarchy:

$$\begin{aligned} x \vee x \wedge y \equiv x &\quad\rightsquigarrow\quad (x \vee (x \wedge y)) \equiv x \\ \neg\neg x = y \equiv y = x &\quad\rightsquigarrow\quad \neg(\neg(x = y)) \equiv (y = x) \end{aligned}$$

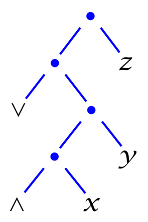
We write $s \neq t$ and $s \not\equiv t$ as abbreviations for $\neg(s = t)$ and $\neg(s \equiv t)$.

2.4 Terms

We distinguish between **notation** and **syntax**. For instance, the notations $x \cdot y + z$ and $(x \cdot y) + z$ are different but both describe the the same syntactic object. We call the syntactic objects described by notations **terms**. Terms can be represented as trees. For instance, the notation $x \wedge y$ describes the term

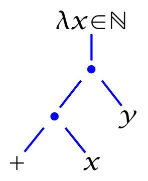


and the notation $x \wedge y \vee z$ describes the term



The inner nodes • of the trees represent function applications. The leaves of the tree are marked with names. Given values for the names appearing in a term, one can **evaluate** the term by performing the applications. Of course, the values involved must have the right types. The value appearing at the left-hand side of an application must be a function that is defined on the value appearing at the right-hand side of the application. Binary applications suffice since operations taking more than one argument are seen as cascaded functions.

The λ-notation decribes special terms called **λ-abstractions** or **λ-terms**. For instance, the notation $\lambda x{\in}\mathbb{N}.\ x + y$ decribes the following λ-term:



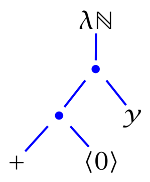
The name x acts as **argument name**. The argument name of a λ-term makes it possible to refer in the **body** of the λ-term to the argument of the function the λ-term decribes.

We distinguish between three levels: the **notational level**, the **syntactic level**, and the **semantic level**. For instance, $\lambda x{\in}\mathbb{N}.\ 2{\cdot}x$ is first of all a notation. This notation describes a certain λ-term, which is a syntactic object. And the λ-term decribes a function, which is a semantic object. Terms abstract from the details of notations. For this reason, there are usually many different notations for the same term. Operator precedence is an issue that belongs to the notational level. Since terms are tree-like objects, there is no need for operator precedence at the syntactic level.

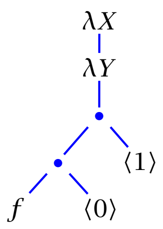
A few words about how we say things. When we say *the term* $\lambda x{\in}\mathbb{N}.\ 2{\cdot}x$, we mean the term described by the notation $\lambda x{\in}\mathbb{N}.\ 2{\cdot}x$. And when we say *the function* $\lambda x{\in}\mathbb{N}.\ 2{\cdot}x$, we mean the function described by the term described by the notation $\lambda x{\in}\mathbb{N}.\ 2{\cdot}x$.

2.5 Locally Nameless Term Representation

λ-terms introduce argument names, which are also called **local names**. It is common to speak of argument variables or local variables. Argument names make it possible to refer in the body of a λ-term to the argument of the function the λ-term decribes. As an alternative to argument names one can use **numeric argument references**, which yield a **locally nameless term representation**. The locally nameless representation of the term $\lambda x{\in}\mathbb{N}.\ x + y$ looks as follows:



The idea behind the locally nameless representation becomes clearer, if we look at the tree representing the term described by the notation $\lambda x{\in}X.\ \lambda y{\in}Y.\ fyx$:



An **argument reference** $\langle n\rangle$ refers to a λ-node on the unique path to the root. The number n says how many λ-nodes are to be skipped before the right λ-node is reached. For instance, $\langle 0\rangle$ refers to the first λ-node encountered on the path to the root, and $\langle 1\rangle$ to the second.

The locally nameless term representation is useful since it represents the binding stucture of a term more explicitly than the name-based term representation. Note that we consider the terms $\lambda x{\in}X.\,x$ and $\lambda y{\in}X.\,y$ to be different although they have the same locally nameless representation.

Numeric argument references are common in machine-oriented languages. For terms, they were invented by the Dutch logician Nicolaas de Bruijn [12]. For this reason, numeric argument references in terms are often called **de Bruijn indices**.

Exercise 2.5.1 Draw the locally nameless representations of the following terms:

a) $\lambda x{\in}X.\,(\lambda y{\in}X.\,fxy)x$

b) $\lambda x{\in}\mathbb{B}.\,\lambda y{\in}\mathbb{B}.\,\neg x \vee y$

c) $\lambda x{\in}X.\,f(\lambda y{\in}Y.\,gyx)xy$

2.6 Formulas, Identities, and Overloading

A **formula** is a term whose type is $\mathbb{B}$. Here are examples of formulas: $3 < 7$, $2+3 > 6$, and $x < 3 \wedge y > 5$. We will represent mathematical statements as formulas.

For every set X, **the identity for** X is the following function:

$$(=_X) \;:=\; \lambda x{\in}X.\,\lambda y{\in}X.\,x{=}y$$

Note that $(=_X) \in X \to X \to \mathbb{B}$. Also note that $=_\mathbb{B}$ and $\equiv$ are different names for the same function. Identities are important since they make it possible to represent equations as terms. For instance, the equation $x + 0 = x$ may be represented as the term $x + 0 =_\mathbb{Z} x$.

As it comes to notation, we will be sloppy and mostly write $=$ rather than the proper name $=_X$. This means that we leave it to the reader to determine the type of the identitity. We speak of the **disambiguation** of **overloaded symbols**. Typical examples of overloaded symbols are $+$, $-$, $<$, and $=$. If we write $x+2 = y$, without further information it is not clear how to disambiguate $+$ and $=$. One possibility would be the use of $+_\mathbb{Z}$ and $=_\mathbb{Z}$.

Exercise 2.6.1 Draw the tree representations of the following formulas. Disambiguate the equality symbol $=$. Recall the specification of the operator precedences in §2.3.

a) $x = 0 \vee x \wedge y \equiv x$

b) $\neg\neg x = y \equiv y = x \wedge x$

2.7 Quantifiers

Mathematical statements often involve quantification. For instance,

$$\forall x \in \mathbb{Z}\ \exists y \in \mathbb{Z}.\ x + y = 0$$

Church [10] realized that the quantifiers $\forall$ and $\exists$ can be represented as functions, and that a quantification can be represented as the application of a quantifier to a λ-term. We may say that Church did for the quantifiers what Boole [7] did for the Boolean operations, that is, explain them as functions.

Let X be a set. We define the **quantifiers $\forall_X$ and $\exists_X$** as follows:

$$\begin{aligned}&\forall_X \in (X \to \mathbb{B}) \to \mathbb{B} \qquad \textbf{universal quantifier}\\ &\forall_X f = (f = (\lambda x{\in}X.1))\end{aligned}$$

$$\begin{aligned}&\exists_X \in (X \to \mathbb{B}) \to \mathbb{B} \qquad \textbf{existential quantifier}\\ &\exists_X f = (f \neq (\lambda x{\in}X.0))\end{aligned}$$

The statement $\exists y \in \mathbb{Z}.\ x + y = 0$ can now be represented as follows:

$$\exists_{\mathbb{Z}}\,(\lambda y{\in}\mathbb{Z}.\ x + y = 0)$$

The usual notation for quantification can be obtained at the notational level:

$$\begin{aligned}\forall x{\in}X.t \quad &:= \quad \forall_X\,(\lambda x{\in}X.t)\\ \exists x{\in}X.t \quad &:= \quad \exists_X\,(\lambda x{\in}X.t)\end{aligned}$$

Frege and Russel understood quantifiers as properties of properties. If we understand under a property on X a function $X \to \mathbb{B}$, then a property on properties on X has the type $(X \to \mathbb{B}) \to \mathbb{B}$. And in fact, this is the type of the quantifiers $\forall_X$ and $\exists_X$. Note the quantifiers are **higher-order functions** (i.e., functions taking functions as arguments).

Exercise 2.7.1 Draw the locally nameless tree representation of the term $(\forall x{\in}X.\ fx \wedge gx) \equiv \forall f \wedge \forall g$.

2.8 Sets as Functions

Let X be a set. The subsets of X can be expressed as functions $X \to \mathbb{B}$. We will represent a subset $A \subseteq X$ as the function $\lambda x{\in}X.\ x{\in}A$, which yields 1 if its argument is an element of A. This function is called the **characteristic function**

of A with respect to X. The following examples illustrate how set operations can be expressed with characteristic functions:

$$\begin{array}{rcl} x \in A & \rightsquigarrow & Ax \\ A \cap B & \rightsquigarrow & \lambda x{\in}X.\ Ax \wedge Bx \\ A \cup B & \rightsquigarrow & \lambda x{\in}X.\ Ax \vee Bx \end{array}$$

Given the representation of sets as functions, there is no need that a functional language provides special primitives for sets. Note that subsets of X as well as properties on X are expressed as functions $X \to \mathbb{B}$.

Exercise 2.8.1 Let X be a set. We use $P(X)$ as abbreviation for $X \to \mathbb{B}$. Express the following set operations with the logical operations $\neg$, $\wedge$, $\vee$, and $\forall_X$. To give you an idea what to do, here is how one would express set intersection $\cap \in P(X) \to P(X) \to P(X)$: $\cap = \lambda f{\in}P(X).\ \lambda g{\in}P(X).\ \lambda x{\in}X.\ fx \wedge gx$.

a) Union $\cup \in P(X) \to P(X) \to P(X)$

b) Difference $- \in P(X) \to P(X) \to P(X)$

c) Subset $\subseteq \in P(X) \to P(X) \to \mathbb{B}$

d) Disjointness $\parallel \in P(X) \to P(X) \to \mathbb{B}$

e) Membership $(\in) \in X \to P(X) \to \mathbb{B}$

2.9 Choice Functions

Sometimes one wants to define an object as the unique x such that a certain property holds. This kind of definitions can be expressed with choice functions. A **choice function for a set X** is a function $(X \to \mathbb{B}) \to X$ that yields for every non-empty subset A of X an element of A. If a choice function for X is applied to a singleton set $\{x\}$, there is no choice and it must return x. Let $C_{\mathbb{Z}}$ be a choice function for $\mathbb{Z}$. Then

$$0 = C_{\mathbb{Z}}(\lambda{\in}\mathbb{Z}.\ x + x = x)$$

since 0 is the unique integer such that $x + x = x$. Moreover, we can describe subtraction with addition and choice since $x - y$ is the unique z such that $x = y + z$:

$$(-) = \lambda x{\in}\mathbb{Z}.\lambda y{\in}\mathbb{Z}.\ \ C_{\mathbb{Z}}(\lambda z{\in}\mathbb{Z}.\ x = y + z)$$

Exercise 2.9.1 How many choice functions are there for $\mathbb{B}$?

Exercise 2.9.2 Describe the following values with a choice function $C_{\mathbb{N}}$ for $\mathbb{N}$, the Boolean operations $\neg$, $\wedge$, $\vee$, $\to$, addition $+ \in \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, and the identity $=_{\mathbb{N}}$.

a) $f \in \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ such that $fxy = x - y$ if $x \geq y$.
b) The existential quantifier $\exists \in (\mathbb{N} \to \mathbb{B}) \to \mathbb{B}$.
c) The less or equal test $\leq \in \mathbb{N} \to \mathbb{N} \to \mathbb{B}$.
d) $max \in \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ such that $max\ x\ y$ yields the maximum of x, y.
e) $if \in \mathbb{B} \to \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ such that $if\ b\ x\ y$ yields x if $b = 1$ and y otherwise.

2.10 Some Logical Laws

Laws are mathematical statements that are universally true. Let us start with the **de Morgan law for conjunction**:

$$\neg(x \wedge y) \equiv \neg x \vee \neg y$$

The law says that we can see a negated conjunction as a disjunction. Seen syntactically, the law is a formula. It involves the names $\neg$, $\wedge$, $\equiv$, $\vee$ and x, y. The names in the first group are called **constants** and the names x, y are called **variables**. While the meaning of constants is fixed, the meaning of variables is not fixed. When we say that a law **holds** or is **valid**, we mean that the respective formula evaluates to 1 no matter how we choose the values of the variables. Of course, every variable comes with a type (here $\mathbb{B}$) and we can only choose values that are elements of the type (here 0 and 1). By means of universal quantification, we can express explicitly that the names x and y are variables:

$$\forall x{\in}\mathbb{B}\ \forall y{\in}\mathbb{B}.\ \neg(x \wedge y) \equiv \neg x \vee \neg y$$

Leibniz' law says that two values x, y are equal if and only if y satisfies every property x satisfies:

$$x{=}_X y \ \equiv\ \forall f{\in}X{\to}\mathbb{B}.\ fx \to fy$$

At first view, Leibniz' law is quite a surprise. Seen logically, it expresses a rather obvious fact. If $x = y$, then the right-hand of the equivalence obviously evaluates to 1. If $x \neq y$, we choose $f = \lambda z.z{=}x$ to see that the right-hand of the equivalence evaluates to 0. Leibniz' law tells us that a language that can express implication and quantification over properties can also express identities.

Henkin's law says that a language that can express identities and universal quantification over functions $X{\to}X{\to}\mathbb{B}$ can also express conjunction:

$$x \wedge y \ \equiv\ \forall f{\in}X{\to}X{\to}\mathbb{B}.\ fxy = f11$$

If $x = y = 1$, then the equivalence obviously holds. If not both x and y are 1, we choose $f = (\wedge)$ to see that the right-hand of the equivalence evaluates to 0.

The **extensionality law** for functions says that two functions are equal if they agree on all arguments:

$$(\forall x{\in}X.\ fx = gx) \ \rightarrow\ f = g$$

The **η-law** for functions expresses a similar fact:

$$f = \lambda x{\in}X.\ fx$$

Both laws hold since functions are sets of pairs. The **α-law** for functions looks as follows:

$$(\lambda x{\in}X.\ fx) = \lambda y{\in}X.\ fy$$

It is a straightforward consequence of the η-law.

The **de Morgan law for universal quantification** says that a negated universal quantification can be seen as an existential quantification:

$$\neg(\forall x{\in}X.\ s) \ \equiv\ \exists x{\in}X.\ \neg s$$

Seen logically, this law is very different from the previous laws since s is a variable that ranges over formulas, that is, syntactic objects. We will avoid such syntactic variables as much as we can. A regular formulation of de Morgan's law for universal quantification looks as follows:

$$\neg(\forall x{\in}X.\ fx) \ \equiv\ \exists x{\in}X.\ \neg fx$$

Here f is a variable that ranges over functions $X \rightarrow \mathbb{B}$.

Finally, we look at the **β-law**. The β-law is a syntactic law for functions. Here are instances of the β-law:

$$\begin{aligned} (\lambda x{\in}\mathbb{N}.x + 2)5 &= 5 + 2 \\ (\lambda x{\in}\mathbb{N}.x + 2)(x + y) &= (x + y) + 2 \end{aligned}$$

The general form of the β-law is as follows:

$$(\lambda x{\in}X.\ s)t \ =\ s^x_t$$

Both s and t are syntactic variables that range over terms. The notation s^x_t stands fo the term that we obtain from s by replacing every free occurrence of the variable x with the term t. The syntactic operation behind the notation s^x_t is called **substitution**. As it turns out, substitution is not a straightforward operation. To say more, we need the formal treatment of terms presented in the next chapter.

Exercise 2.10.1 (Boolean Formulas) Decide whether the following formulas are valid for all values of the variables $x, y, z \in \mathbb{B}$. In case a formula is not valid, find values for the variables for which the formula does not hold.

a) $1 \to x \equiv x$

b) $(x \to y) \to (\neg y \to \neg x) \equiv 1$

c) $x \wedge y \vee \neg x \wedge z \equiv y \vee z$

Exercise 2.10.2 (Quantifiers and Identities) Given some logical operations, one can express many other logical operations. This was demonstrated by Leibniz' and Henkin's law. Express the following:

a) $\forall_X$ with $=_{X \to \mathbb{B}}$ and 1.

b) $\exists_X$ with $\forall_X$ and $\neg$.

c) $\forall_X$ with $\exists_X$ and $\neg$.

d) $=_{X \to Y}$ with $\forall_X$ and $=_Y$.

e) $=_{\mathbb{B}}$ with $\equiv$.

f) $=_X$ with $\forall_{X \to \mathbb{B}}$ and $\to$.

Exercise 2.10.3 (Henkin's Reduction) In a paper [21] published in 1963, Leon Henkin expressed the Boolean operations and the quantifiers with the identities.

a) Express 1 with $=_{\mathbb{B} \to \mathbb{B}}$.

b) Express 0 with 1 and $=_{\mathbb{B} \to \mathbb{B}}$.

c) Express $\neg$ with 0 and $=_{\mathbb{B}}$.

d) Express $\forall_X$ with 1 and $=_{X \to \mathbb{B}}$.

e) Express $\wedge$ with 1, $=_{\mathbb{B}}$, and $\forall_{\mathbb{B} \to \mathbb{B} \to \mathbb{B}}$.

f) Express $\wedge$ with 1 and $=_{(\mathbb{B} \to \mathbb{B} \to \mathbb{B}) \to \mathbb{B}}$.

2.11 Remarks

The outlined logical language is mainly due to Alonzo Church [10]. It is associated with logical systems known as *simple type theory* or *simply-typed higher-order logic*. Church started with an untyped language [9] but technical difficulties forced him to switch to a typed language [10]. Types originated with Bertrand Russell [34]. A logical language with quantification was first studied by Gottlob Frege [13].

One distinguishes between **metalanguage** and **object language**. The metalanguage is the language one uses to explain an object language. Our object language has many features in common with the metalanguage we use to explain it. Still, it is important to keep the two languages separate. For some constructs that appear in both languages we use different notations. For instance, implication and equivalence are written as $\Longrightarrow$ and $\Longleftrightarrow$ at the metalevel and as $\rightarrow$ and $\equiv$ at the object level. Moreover, at the metalevel we write quantifications with a colon (e.g., $\exists x{\in}\mathbb{N}\colon x < 5$) while at the object level we write them with a dot (e.g., $\exists x{\in}\mathbb{N}.\ x < 5$).

In the theory of programming languages one calls *concrete syntax* what we call notation and *abstract syntax* what we call syntax.

Sometimes one speaks of the *intension* and the *extension* of a notation. While the intension refers to the syntactic object decribed by the notation, the extension refers to the semantic object described by the notation.

3 Terms and Types

In this chapter we study syntax and ignore semantics. We first consider untyped terms.

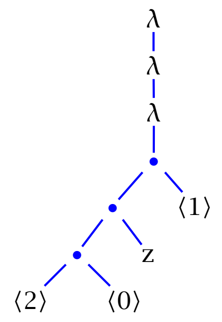
3.1 Untyped Terms

We start from a set Nam of names that is bijective to $\mathbb{N}$. In principle, we could choose $\mathrm{Nam} = \mathbb{N}$, but saying that Nam is bijective to $\mathbb{N}$ gives us more flexibility. The set of terms Ter is obtained recursively from Nam:

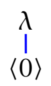
$x, y \in \mathrm{Nam} \;\cong\; \mathbb{N}$	**names**
$s, t \in \mathrm{Ter} \;::=\; x \;\mid\; s\,t \;\mid\; \lambda x.s$	**terms**

We understand the definition such that every term is exactly one of the following: a name x, an **application** st, or a **λ-term** $\lambda x.s$. Moreover, we assume $\mathrm{Nam} \subseteq \mathrm{Ter}$. To be concrete, we represent a term as a pair (i, γ) where the **variant number** $i \in \{1, 2, 3\}$ says whether the term is a name ($i = 1$), an application ($i = 2$), or a λ-term ($i = 3$). For names we have $\gamma \in \mathbb{N}$, for applications $\gamma \in \mathrm{Ter} \times \mathrm{Ter}$, and for λ-abstractions $\gamma \in \mathrm{Nam} \times \mathrm{Ter}$. Note that our definition implies that $(\lambda x.x) \neq \lambda y.y$ if $x \neq y$.

The **locally nameless representation (LNR)** of a term uses numeric argument references instead of local names. For instance, the LNR of $\lambda fxy.fyzx$ looks as follows:



An argument reference $\langle n \rangle$ refers to the $n+1$-th λ-node encountered on the path to the root. Note that $\lambda x.x$ and $\lambda y.y$ have the same LNR:



The **size $|s|$ of a term s** is the number of nodes in its tree representation. The formal definition is recursive and looks as follows:

$$\begin{aligned} |\_| &\in \text{Ter} \to \mathbb{N} \\ |x| &= 1 \\ |st| &= 1 + |s| + |t| \\ |\lambda x.s| &= 1 + |s| \end{aligned}$$

For instance, $|\lambda fxy.fyzx| = 10$. The **free names of a term** are the names that appear in the LNR of the term. The formal definition looks as follows:

$$\begin{aligned} \mathcal{N} &\in \text{Ter} \to \mathcal{P}(\text{Nam}) \\ \mathcal{N}x &= \{x\} \\ \mathcal{N}(st) &= \mathcal{N}s \cup \mathcal{N}t \\ \mathcal{N}(\lambda x.s) &= \mathcal{N}s - \{x\} \end{aligned}$$

For instance, $\mathcal{N}(\lambda fxy.fyzx) = \{z\}$ if we assume that z is different from f, x, y. We say that **x is free in s** if $x \in \mathcal{N}s$. A term s is **closed** if $\mathcal{N}s = \emptyset$, and **open** otherwise.

3.2 Contexts and Subterms

Informally, a context is a term with a hole. Formally, we define **contexts** as follows:

$$C \ ::= \ [] \ \mid \ Cs \ \mid \ sC \ \mid \ \lambda x.C$$

The **instantiation $C[t]$** of a context C with a term t yields the term that is obtained from C by replacing the hole $[]$ with t. For instance, if $C = \lambda xy.f[]$, then $C[gxy] = \lambda xy.f(gxy)$. Formally, we define context instantiation recursively:

$$\begin{aligned} [][t] &= t \\ (Cs)[t] &= (C[t])s \\ (sC)[t] &= s(C[t]) \\ (\lambda x.C)[t] &= \lambda x.C[t] \end{aligned}$$

A term s is a **subterm** of a term t if there exists a context C such that $t = C[s]$. We say that a term **s contains a term t** or that **t occurs in s** if t is a subterm of s. A term is called **combinatorial** if none of its subterms is a λ-term.

Exercise 3.2.1 Give all subterms of the term $\lambda x.fxx$. For each subterm give a corresponding context. Is there a subterm with more than one corresponding context?

Exercise 3.2.2 Determine all pairs C, s such that $C[s] = xxxx$ and s is a application.

Exercise 3.2.3 We say that a name x occurs **bound** in a term s if s has a subterm $\lambda x.t$ such that x is free in t. Give a term s such that x is free in s and also occurs bound in s.

3.3 Substitution

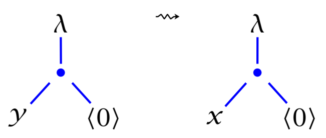
A **substitution** is a function $\theta \in \text{Nam} \to \text{Ter}$. There is an operation $S \in (\text{Nam} \to \text{Ter}) \to \text{Ter} \to \text{Ter}$ that applies a substitution to a term. If s is combinatorial, then $S\theta s$ is obtained from s by replacing every occurrence of a name x in s with the term θx. This can be expressed with two equations:

$$\begin{aligned} S\theta x &= \theta x \\ S\theta(st) &= (\theta s)(\theta t) \end{aligned}$$

For instance, $S\theta(fxy) = (\lambda xy.x)yx$ if $\theta f = (\lambda xy.x)$, $\theta x = y$, $\theta y = x$.

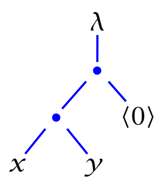
The recursion equation for $S\theta(\lambda x.s)$ is not straightforward. It is clear that the local name x must not be replaced in s. However, there is a second, more servere complication known as **capturing**. Consider $S\theta(\lambda x.y)$ where $x \neq y$ and $\theta y = x$. Then the y in $\lambda x.y$ must be replaced with x. If we do this naively, we obtain $\lambda x.x$, which means that the external x has been captured as local name. To obtain the right semantic properties, substitution must be defined such that capturing does not happen. This can be done by of **renaming local names**, also known as **α-renaming**.

We say that capture-free substitution preserves the binding structure of a term. The binding structure of a term is expressed best by its LNR. The substitution operation S must be defined such that the LNR of the term $S\theta s$ can be obtained by replacing every name x in the LNR of s with the LNR of θx. For instance, if $s = \lambda x.yx$ and $\theta y = x$, we want the following:



Consequently, $S\theta s$ cannot be the term $\lambda x.xx$ but must be some term $\lambda z.xz$.

How should the substitution operation choose names if it has to rename local names? Its best to first consider a simplified problem: How can we choose local names for an LNR such that we obtain a term whose LNR is the given one. For instance, consider the LNR



To obtain a term with this LNR, we can choose any local name that is different from x and y. For instance, $\lambda z.xyz$. If we choose x or y as local name, the local name will capture a free name, which results in a different LNR.

Consider a term $\lambda x.s$. Which local names can we use in place of x without changing the LNR? If you think about it you will see that all names that are not free in $\lambda x.s$ are ok. The names that are free in $\lambda x.s$ are not ok since using such a name as argument name would capture the free occurrences of this name.

A **choice function** is a function $\rho \in \mathrm{Nam} \to \mathcal{P}_{\mathit{fin}}(\mathrm{Nam}) \to \mathrm{Nam}$ such that $\rho x N \notin N$ for all x and N. $\mathcal{P}_{\mathit{fin}}(\mathrm{Nam})$ is the set of all finite subsets of Nam. A choice function ρ is **conservative** if $\rho x N = x$ whenever $x \notin N$. Given a choice function ρ, the following recursion equations yield a substitution operation S:

$$\begin{aligned} S\theta x &= \theta x \\ S\theta(st) &= (S\theta s)(S\theta t) \\ S\theta(\lambda x.s) &= \lambda y.\, S(\theta[x{:=}y])s \qquad \text{where } y = \rho x(\cup\{\,\mathcal{N}(\theta z) \mid z \in \mathcal{N}(\lambda x.s)\,\}) \end{aligned}$$

The notation $\theta[x{:=}y]$ describes the substitution that is like θ except that it maps x to y. We call a substitution operation S **conservative** if it is obtained with a conservative choice function.

Proposition 3.3.1 For every substitution operation S:

1. $\mathcal{N}(\theta s) = \cup\{\,\mathcal{N}(\theta x) \mid x \in \mathcal{N}s\,\}$
2. $(\forall x \in \mathcal{N}s\colon\ \theta x = \theta' x) \implies S\theta s = S\theta' s$ *coincidence*

Proposition 3.3.2 For every conservative substitution operation S:
$(\forall x{\in}\mathcal{N}s\colon \theta x = x) \implies S\theta s = s$.

We fix some conservative substitution operator S and define the notation

$$s^x_t \;:=\; S\,(\lambda y{\in}\mathrm{Nam}.\ \text{if } y = x \text{ then } t \text{ else } y)\, s$$

Proposition 3.3.3

1. $(\lambda x.s)^x_t = (\lambda x.s)$
2. $(\lambda x.s)^y_t = (\lambda x.s^y_t)$ if $x \neq y$ and $x \notin \mathcal{N}t$
3. $s^x_t = s$ if $x \notin \mathcal{N}s$

Exercise 3.3.4 Apply the following substitutions.

a) $((\lambda x.y)y)^y_x$

b) $(\lambda x.y)^y_{fxy}$

c) $(\lambda x.y)^x_{fxy}$

Exercise 3.3.5 Let $x \neq y$. Find C such that $(C[x])^x_y \neq C[y]$. Hint: Exploit that $C[x]$ may capture x.

3.4 Alpha Equivalence

Two terms are *α-equivalent* if the have the same LNR. This doesn't suffice for a formal definition since the LNR of terms is not formally defined. It turns out that there is a non-conservative substitution operator that provides for an elegant formal definition of α-equivalence.

Let ρ_0 be the choice function such that $\rho_0 xN$ is the least name that is not in N (we exploit a bijection between Nam and $\mathbb{N}$ for the order). Moreover, let S_0 be the substitution operation obtained with ρ_0. Finally, let ϵ be the identity substitution such that $\epsilon x = x$ for all names x. We define **α-equivalence** as follows:

$$s \sim_\alpha t \;:\Longleftrightarrow\; S_0\epsilon s = S_0\epsilon t$$

Intuitively, this definition works since $S_0\epsilon s$ yields a variant of s where all local names are chosen to be the least possible ones. Here are examples ($x \cong 0$, $y \cong 1$):

$$\begin{aligned} S_0\epsilon(\lambda x.x) &= \lambda x.x \\ S_0\epsilon(\lambda y.y) &= \lambda x.x \\ S_0\epsilon(\lambda yx.xy) &= \lambda xy.yx \end{aligned}$$

A careful study of S_0 with complete proofs of its basic properties can be found in a paper by Allen Stoughton [36].

Proposition 3.4.1

1. $S_0\epsilon(S_0\epsilon s) = S_0\epsilon s$
2. $S_0\epsilon s \sim_\alpha s$

Proposition 3.4.2 Let z be a name that is not free in $\lambda x.s$ or $\lambda y.t$. Then $\lambda x.s \sim_\alpha \lambda y.t \iff s^x_z \sim_\alpha t^y_z$.

Proposition 3.4.3 $\lambda x.s \sim_\alpha \lambda y.t \iff y \notin \mathcal{N}(\lambda x.s) \wedge t = s^x_y$

Proposition 3.4.4 Alpha equivalence $\sim_\alpha$ is an equivalence relation on Ter satisfying the following properties:

1. $s \sim_\alpha t \implies C[s] \sim_\alpha C[t]$ *compatibility*
2. $s \sim_\alpha t \implies S\theta s \sim_\alpha S\theta t$ *stability*

Since we will see several other relations on the set of terms that are compatible and stable, we define these properties in a general way. A binary relation R on the set of all terms is

- **compatible** if $\forall (s,t) \in R \;\; \forall C\colon\; (C[s], C[t]) \in R$
- **stable** if $\forall (s,t) \in R \;\; \forall \theta\colon\; (S\theta s, S\theta t) \in R$

Exercise 3.4.5 Which of the following terms are α-equivalent?

$$\lambda xyz.xyz, \quad \lambda yxz.yxz, \quad \lambda zyx.zyx, \quad \lambda xyz.zyx, \quad \lambda yxz.zxy$$

Exercise 3.4.6 Determine $S_0\epsilon t$ for the following terms t. Assume $x \cong 0$, $y \cong 1$, and $z \cong 2$.

a) $\lambda z.z$

b) $\lambda yx.yx$

c) $\lambda xy.yx$

d) $\lambda xy.y$

e) $\lambda zxy.xyz$

f) $\lambda z.x$

Exercise 3.4.7 Find counterexamples that falsify the following statements.

a) $\lambda x.s \sim_\alpha \lambda y.t \iff \exists z : s^x_z \sim_\alpha t^y_z$

b) $\lambda x.s \sim_\alpha \lambda y.t \iff s^x_y \sim_\alpha t$

3.5 Confluence and Termination

We define some notions for binary relations that we need for α-equivalence and other relations on terms. Read Chapter 2 of Baader and Nipkow [3] to learn more.

Let X be a non-empty set and $\to \subseteq X \times X$. We write $x \to y$ if and only if $(x,y) \in \to$. Moreover, we write $x \to^* y$ if there exist $x_1, \dots, x_n$ such that $x = x_1 \to \cdots \to x_n = y$. We use $\to^*$ to denote the corresponding reflexive and transitive relation. We write $x \downarrow y$ and say that x and y are **$\to$-joinable** if there exists a z such that $x \to^* z$ and $y \to^* z$. We use $\downarrow$ to denote the corresponding relation. We say that x is **$\to$-normal** if there is no y such that $x \to y$. We say that y is a **$\to$-normal form** of x if $x \to^* y$ and y is $\to$-normal. We say that $\to$ is **confluent** if for all x, y, z: $\quad x \to^* y \wedge x \to^* z \Longrightarrow y \downarrow z$.

Proposition 3.5.1 If $\to$ is confluent, then no x has more than one $\to$-normal form.

Proposition 3.5.2 If $\to$ is confluent, then $\downarrow$ is the least equivalence relation $\sim$ on X such that $\to \subseteq \sim$.

Proposition 3.5.3 $\to$ is confluent if and only if $\downarrow$ is an equivalence relation.

We say that $\to$ **terminates on x** if there is no infinite chain $x \to x_1 \to x_2 \to \cdots$. We say that $\to$ is **terminating** if there is no x on which $\to$ doesn't terminate.

Proposition 3.5.4 If $\to$ terminates, every x has a $\to$-normal form. If $\to$ terminates and is confluent, every x has a unique $\to$-normal form.

We now return to α-equivalence of terms. The **α-rule**

$$\lambda x.s \;\to\; \lambda y.s^x_y \qquad \text{if } y \notin \mathcal{N}(\lambda x.s) \text{ and } y < x$$

replaces the local name of a λ-term with a smaller name. We define **α-reduction** as the binary relation $\to_\alpha$ on Ter satisfying the following condition:

$$s \to_\alpha t \iff \exists C, x, u, y\colon\; s = C[\lambda x.u] \wedge t = C[\lambda y.u^x_y] \wedge y \notin \mathcal{N}(\lambda x.u) \wedge y < x$$

Proposition 3.5.5 $\to_\alpha$ is confluent an terminating. Moreover, $\downarrow_\alpha = \sim_\alpha$.

Thus, two terms are α-equivalent if and only if they have the same α-normal form (i.e., $\to_\alpha$-normal form).

Hint for the exercises: Draw finite relations as graphs.

Exercise 3.5.6 Give finite relations $\to$ such that:

a) $\to$ is confluent but not terminating.

b) $\to$ is terminating but not confluent.

c) $\to$ is not confluent and not terminating.

d) $\to$ is confluent, does not terminate on x, and y is a $\to$-normal form of x.

Exercise 3.5.7 Consider the relation $\to := \{\,(x,y) \in \mathbb{N}^2 \mid 2 \le 2y \le x\,\}$.

a) Is $\to$ terminating?

b) Is $\to$ confluent?

c) Give a $\to$-normal form of 7.

d) Give all $\to$-normal $n \in \mathbb{N}$.

Exercise 3.5.8 A relation $\to$ is *locally confluent* if for all x, y, z: $\quad x \to y \wedge x \to z \Longrightarrow y \downarrow z$. Find a finite relation that is locally confluent but not confluent.

3.6 Beta and Eta Reduction

A **β-redex** is a term of the form $(\lambda x.s)t$ (i.e., an application whose left-hand side is a λ-term). The **β-rule**

$$(\lambda x.s)t \;\to\; s^x_t$$

rewrites a β-redex $(\lambda x.s)t$ to the term s^x_y. We define **β-reduction** as the binary relation $\to_\beta$ on Ter satisfying the following condition:

$$s \to_\beta t \iff \exists C, x, u, v\colon\; s = C[(\lambda x.u)v] \;\wedge\; t = C[u^x_v]$$

A term is **β-normal** if it doesn't contain a β-redex. We say that t is a **β-normal form of s** if $s \to^*_\beta t$ and t is β-normal. Here is an example:

$$\begin{aligned}(\lambda xy.y)((\lambda x.x)y)z \;&\to_\beta\; (\lambda xy.y)yz\\ &\to_\beta\; (\lambda y.y)z\\ &\to_\beta\; z\end{aligned}$$

Note that the term z is a β-normal form of $(\lambda xy.y)((\lambda x.x)y)z$. Since the term $(\lambda xy.y)((\lambda x.x)y)z$ contains two β-redexes, we can reduce it also as follows:

$$(\lambda xy.y)((\lambda x.x)y)z \;\to_\beta\; (\lambda y.y)z \;\to_\beta\; z$$

Consider the term $\omega := \lambda x.xx$. We have $\omega\omega \to_\beta \omega\omega$. Hence $\omega\omega$ is a term that has no β-normal form.

Proposition 3.6.1

1. $s \to_\beta t \implies C[s] \to_\beta C[t]$ *compatibility*
2. $s \to_\beta t \implies S\theta s \to_\beta S\theta t$ *stability*
3. $s_1 \sim_\alpha s_2 \wedge s_1 \to_\beta t_1 \implies \exists t_2\colon s_2 \to_\beta t_2 \wedge t_1 \sim_\alpha t_2$ *α-compatibility*

Two terms s_1, s_2 are **β-joinable** if there exist terms t_1, t_2 such that $s_1 \to^*_\beta t_1$, $s_2 \to^*_\beta t_2$, and $t_1 \sim_\alpha t_2$.

Theorem 3.6.2 (Confluence) If $s \to^*_\beta t_1$ and $s \to^*_\beta t_2$, then t_1 and t_2 are β-joinable.

The Confluence Theorem was first shown in 1936 by Church and Rosser. The proof is not straightforward. You find it in Barendregt [5].

Corollary 3.6.3 If a term has a β-normal form, then it is unique up to α-equivalence.

An **η-redex** is a term of the form $\lambda x.sx$ where $x \notin \mathcal{N}s$. The **η-rule**

$$\lambda x.sx \;\to\; s \qquad \text{if } x \notin \mathcal{N}s$$

rewrites an η-redex $\lambda x.sx$ to the term s. We define **η-reduction** as the binary relation $\to_\eta$ on Ter satisfying the following condition:

$$s \to_\eta t \iff \exists C, x, u\colon\; s = C[\lambda x.ux] \;\wedge\; t = C[u] \;\wedge\; x \notin \mathcal{N}u$$

A term is **η-normal** if it doesn't contain an η-redex. We say that t is a **η-normal form of s** if $s \to_\eta^* t$ and t is η-normal. Here is an example:

$$\lambda xy.fxy \;\to_\eta\; \lambda x.fx \;\to_\eta\; f$$

Note that the term f is an η-normal form of $\lambda xy.fxy$. Also note that $\lambda xy.fyx$ is η-normal.

Proposition 3.6.4 $\to_\eta$ is terminating.

Proposition 3.6.5 If $s \to_\eta t$ and s is β-normal, then t is β-normal.

The relation $\to_{\beta\eta} := \to_\beta \cup \to_\eta$ is called **$\beta\eta$-reduction**. $\beta\eta$-normal forms are defined as one would expect. Confluence modulo α-equivalence also holds for $\beta\eta$-reduction. Finally, we define **λ-reduction**: $\to_\lambda := \to_\alpha \cup \to_\beta \cup \to_\eta$.

Theorem 3.6.6 (Confluence) $\to_\lambda$ is confluent.

The equivalence relation $\sim_\lambda := \downarrow_\lambda$ is called **λ-equivalence** or **lambda equivalence**.

Proposition 3.6.7 $\to_\lambda$ and $\sim_\lambda$ are compatible and stable relations.

Proposition 3.6.8 Lambda equivalence is the least relation R on the set of terms such that R is symmetric, transitive and $\to_\beta \cup \to_\eta \subseteq R$.

Proof One direction is straightforward. For the other direction, let R be a relation with the required properties. Let s be a term. Then $(s,s) \in R$ since $\lambda x.s \to_\beta s$ for all x and R is symmetric and transitive. It remains to show $\to_\alpha \subseteq R$. Let $y \notin \mathcal{N}(\lambda x.s)$. Then $\lambda y.(\lambda x.s)y \to_\eta \lambda x.s$ and $\lambda y.(\lambda x.s)y \to_\beta \lambda x.s^x_y$. Hence $(\lambda x.s,\ \lambda y.(\lambda x.s)y) \in R$ since R is symmetric and transitive. ∎

Exercise 3.6.9 Give the β-normal forms of the following terms.

a) $(\lambda xy.fyx)ab$

b) $(\lambda fxy.fyx)(\lambda xy.yx)ab$

c) $(\lambda x.xx)((\lambda xy.y)((\lambda xy.x)ab))$

d) $(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))a$

e) $(\lambda xx.x)yz$

Exercise 3.6.10 Give the $\beta\eta$-normal forms of the following terms.

a) $\lambda xy.fx$

b) $\lambda xy.fy$

c) $\lambda xy.fxy$

Exercise 3.6.11 Determine all pairs C, s such that $C[s] = \lambda xyz.(\lambda x.x)yxz$ and s is a β- or η-redex.

Exercise 3.6.12 Find terms as follows.

a) A term that has no β-normal form.

b) A term that has a β-normal form but is not terminating.

Exercise 3.6.13

a) Find a term that has more than one β-normal form.

b) Find a term s such that there infinitely many terms t such that $s \to_\beta^* t$.

c) Find terms s, t such that $s \to_\beta t$, s contains no η-redex, and t contains an η-redex.

3.7 Typed Terms

Types are syntactic objects like terms. We define them as follows:

$\alpha, \beta \in \mathit{Sor} \;\cong\; \mathbb{N}$	**sorts**
$\sigma, \tau \in \mathrm{Ty} \;::=\; \alpha \mid \sigma\tau$	**types**

Sorts are primitive types. We assume $\mathit{Sor} \subseteq \mathrm{Ty}$. Types of the form $\sigma\tau$ are called **functional**. Here are examples of functional types: $\alpha\beta$, $(\alpha\beta)\beta$, $((\alpha\beta)\beta)\beta$. We omit parentheses according to $\sigma_1\sigma_2\sigma_3 \rightsquigarrow \sigma_1(\sigma_2\sigma_3)$. You can see a type as a binary tree whose leaves are labeled with sorts. **Typed terms** are defined as follows:

$x, y \in \mathrm{Nam} \;\cong\; \mathbb{N} \times \mathrm{Ty}$	**names**
$s, t \in \mathrm{Ter} \;::=\; x \mid st \mid \lambda x.s$	**terms**

The difference to untyped terms is that typed terms employ typed names. Every name x comes with unique type τx. By definition there are infinitely many names for every type. We define a **typing relation** $(:) \subseteq \mathrm{Ter} \times \mathrm{Ty}$ that relates terms and types. The definition is recursive:

1. If x is a name, then $x : \tau x$.
2. If $s : \sigma\tau$ and $t : \sigma$, then $st : \tau$.
3. If $x : \sigma$ and $s : \tau$, then $\lambda x.s : \sigma\tau$.

The definition of the typing relation can also be formulated with inference rules:

$$\frac{}{x : \sigma}\ \tau x = \sigma \qquad \frac{s : \sigma\tau \quad t : \sigma}{st : \tau} \qquad \frac{x : \sigma \quad s : \tau}{\lambda x.s : \sigma\tau}$$

A term s is **well-typed** if there exists a type σ such that $s : \sigma$. Terms that are not well-typed are called **ill-typed**. A substitution θ is **well-typed** if $\theta x : \tau x$ for all names x.

For typed terms (not necessarily well-typed) the choice function $\rho x N$ for the substitution operation must be restricted such that it returns a name that has the same type as x. Moreover, α-reduction must replace the local name of a λ-term with a name of the same type. All other definitions for untyped terms carry through to typed terms without change.

Proposition 3.7.1

1. $s : \sigma \wedge s : \tau \Rightarrow \sigma = \tau$
2. If s is well-typed, then every subterm of s is well-typed.
3. If θ is well-typed and $s : \sigma$, then $\theta s : \sigma$.
4. If $s \to_\lambda t$ and $s : \sigma$, then $t : \sigma$.

In words, the proposition states the following:

- The type of a well-typed term is unique.
- Application of a well-typed substitution to a well-typed term preserves the type of the term.
- Lambda reduction preserves the type of a term.

Consider the term $\omega = \lambda x.xx$. This term is ill-typed. We prove this by contradiction. Suppose ω is well-typed. Then, by the above proposition, the subterm xx is well-typed. Hence there is a type $\sigma\tau$ such that $\sigma\tau = \sigma$. This is a contradiction since the type σ is smaller than the functional type $\sigma\tau$.

Theorem 3.7.2 (Termination) $\to_\lambda$ terminates on well-typed terms.

The Termination Theorem was first shown in 1967 by Tait [37]. The proof is not straightforward and can be found in [17].

Corollary 3.7.3 Every well-typed term has a unique λ-normal form and a β-normal form that is unique up to α-equivalence.

Proof The existence of a normal form follows from the Termination Theorem. The uniqueness follows with the Confluence Theorem. ■

Corollary 3.7.4 Lambda equivalence of well-typed terms is decidable.

Proof Follows with the Confluence and the Termination Theorem. The decision algorithm computes the λ-normal forms of the given terms and checks whether they are equal. ■

Exercise 3.7.5 For each of the following terms finds types for the names occurring in the term such that the term becomes well-typed.

a) $\lambda xy.x$

b) $\lambda f.fyx$

c) $\lambda fgx.fx(gx)$

Exercise 3.7.6 Find closed terms that have the following types.

a) $\alpha\alpha$

b) $\alpha\beta\alpha$

c) $(\alpha\beta)(\beta\gamma)\alpha\gamma$

d) $\alpha(\alpha\beta)\beta$

Exercise 3.7.7 Find terms s, t such that $s \to_\beta t$, s is ill-typed, and t is well-typed.

3.8 Remarks

We have defined terms and types as syntactic objects. We have stated many propositions and two famous theorems. We have not given proofs. The proofs are typically inductive. The proofs of the propositions are not difficult but require technical details and care. The proofs of the theorems are non-trivial.

The system based on untyped terms is known as **untyped lambda calculus**, and the system based on typed terms is known as **simply typed lambda calculus**. Both systems originated with Church [9, 10]. Church's final account of the untyped lambda calculus can be found in [11]. Nowadays, the standard reference for the untyped lambda calculus is Barendregt [5]. A textbook introducing the untyped and the typed lambda calculus is Hindley [22]. The lambda calculus is essential in the theory of programming languages. Pierce [31] contains a programming languages oriented introduction to the lambda calculus. A more advanced textbook is Mitchell [28].

One can formalize LNRs and work with LNRs rather than terms. The advantage of LNRs is that the definition of substitution is straightforward and that α-equivalence is not needed. As it comes to the semantic interpretation of terms, the representation of local names is redundant. However, the LNR approach has also drawbacks. The notions of subterm and context need to be revised. If you look at the tree representation, its clear that we need to admit LNRs with dangling argument references to account for the β-reduction of subterms that occur below λ-nodes. What one ends up with is de Bruijn's representation [12], which was conceived for an implementation of terms. Currently, the adequate formalization of terms is a hot research topic and there are several competing approaches (e.g., [39, 2, 27]). Following Barendregt [5], most modern presentations of the lambda calculus sweep the problems under the carpet by saying that α-equivalent terms are identified.

4 Interpretation and Specification

In this chapter we define the semantics of well-typed terms and set up *simple type theory*, the logic we will be working with. The highlight of this chapter is a specification of the natural numbers in simple type theory.

We tacitly assume that all terms and substitutions are well-typed and that relations like $\sim_\alpha$, $\to_\beta$, and $\sim_\lambda$ are restricted to well-typed terms. Moreover, Ter will denote from now on the set of all well-typed terms.

4.1 Interpretations

If we associate with every sort a non-empty set, every type describes a non-empty set. We require that a functional type $\sigma\tau$ describes the set of all functions $A \to B$ where A is the set described by σ and B is the set described by τ.

If we associate with every name x an element of the set described by the type of x, every well-typed term s describes a value that is an element of the set described by the type of s. The well-typedness condition for applications st ensures that s describes a function that is defined for the value described by t.

The interpretation of types and terms is something you are familiar with through years of mathematical training, maybe with the exception of λ-terms. Since we have a formal account of terms and types, we can also formalize their interpretation. We will do this in the following.

We assume that the sets Ty (all types) and Ter (all well-typed terms) are disjoint. We distinguish between interpretations and evaluations. An interpretation is a function that assigns a set to every type and a value to every name. An evaluation is a function that assigns a value to every term. Every interpretation yields a unique evaluation.

For interpretations and evaluation we need a general notion of function. A **function** f is a set of pairs such that for no pair $(x, y) \in f$ there is a $z \neq y$ such that $(x, z) \in f$. The **domain** and the **range** of a function f are defined as follows:

$$\begin{aligned} \mathrm{Dom}\, f &:= \{\, x \mid \exists y\colon (x, y) \in f \,\} \\ \mathrm{Ran}\, f &:= \{\, y \mid \exists x\colon (x, y) \in f \,\} \end{aligned}$$

An **interpretation** is a function $\mathcal{I}$ such that:

1. $\mathrm{Dom}\,\mathcal{I} = \mathrm{Ty} \cup \mathrm{Nam}$
2. $\mathcal{I}\alpha$ is a non-empty set for all sorts α
3. $\mathcal{I}(\sigma\tau) = \{\, \varphi \mid \varphi \text{ function } \mathcal{I}\sigma \to \mathcal{I}\tau \,\}$ for all types σ, τ
4. $\mathcal{I}x \in \mathcal{I}(\tau x)$ for all names x

Proposition 4.1.1 (Coincidence) Two interpretations are equal if they agree on all sorts and all names.

Given an interpretation $\mathcal{I}$, a name x, and a value $v \in \mathcal{I}(\tau x)$, we use $\mathcal{I}_{x,v}$ to denote the interpretation $\mathcal{I}[x{:=}v]$ ($\mathcal{I}[x{:=}v]$ is like $\mathcal{I}$ but maps x to v).

Proposition 4.1.2 (Evaluation) For every interpretation $\mathcal{I}$ there exists exactly one function $\hat{\mathcal{I}}$ such that:

1. $\mathrm{Dom}\,\hat{\mathcal{I}} = \mathrm{Ter}$
2. $s : \sigma \implies \hat{\mathcal{I}}s \in \mathcal{I}\sigma$ for all terms s and all types σ
3. $\hat{\mathcal{I}}x = \mathcal{I}x$ for all names x
4. $\hat{\mathcal{I}}(st) = (\hat{\mathcal{I}}s)(\hat{\mathcal{I}}t)$ for all applications st
5. $\hat{\mathcal{I}}(\lambda x.s) = \lambda v{\in}\mathcal{I}(\tau x).\, \hat{\mathcal{I}}_{x,v}\, s$ for all terms $\lambda x.s$

Given an interpretation $\mathcal{I}$ and a substitution θ, we use $\mathcal{I}_\theta$ to denote the interpretation satisfying the following equations:

$$\mathcal{I}_\theta\sigma = \mathcal{I}\sigma \qquad \text{for every type } \sigma$$
$$\mathcal{I}_\theta x = \hat{\mathcal{I}}(\theta x) \qquad \text{for every name } x$$

Lemma 4.1.3 (Substitution) $\hat{\mathcal{I}}(S\theta s) = \hat{\mathcal{I}}_\theta s$.

The lemma says that substitution and interpretation interact smoothly. The lemma is of great technical importance. If substitution was not defined capture-free, the lemma would not hold.

The atoms of a term are all sorts and names the interpretation of the term depends on. For instance, the atoms of $\lambda x.x$ are all sorts that occur in τx. Formally, we define the **atoms occurring in a type or term** as follows:

$$\begin{aligned}
\mathrm{Atom}\,\alpha &= \{\alpha\} \\
\mathrm{Atom}\,(\sigma\tau) &= \mathrm{Atom}\,\sigma \cup \mathrm{Atom}\,\tau \\
\mathrm{Atom}\,(x) &= \{x\} \cup \mathrm{Atom}\,(\tau x) \\
\mathrm{Atom}\,(st) &= \mathrm{Atom}\,s \cup \mathrm{Atom}\,t \\
\mathrm{Atom}\,(\lambda x.s) &= \mathrm{Atom}\,(\tau x) \cup (\mathrm{Atom}\,s - \{x\})
\end{aligned}$$

If an atom occurs in a term or type, we also say that the term or type **contains** the atom or **depends** on the atom.

Proposition 4.1.4 (Coincidence) If $\mathcal{I}$ and $\mathcal{I}'$ agree on $\mathrm{Atom}\,s$, then $\hat{\mathcal{I}}s = \hat{\mathcal{I}}'s$.

4.2 Semantic Equivalence

We define **semantic equivalence** of terms as follows:

$$s \approx t \quad :\Longleftrightarrow \quad (\exists \sigma\colon\ s : \sigma \wedge t : \sigma) \ \wedge\ \forall \mathcal{I}\colon\ \hat{\mathcal{I}}s = \hat{\mathcal{I}}t$$

In words, two terms are semantically equivalent if they have the same type and yield the same value for every interpretation. Semantically equivalent terms cannot be distinguished through interpretations.

Proposition 4.2.1 (Soundness) $s \sim_\lambda t \implies s \approx t$

Theorem 4.2.2 (Completeness) $s \approx t \implies s \sim_\lambda t$

Together, soundness and completeness say that lambda equivalence (a syntactic notion) and semantic equivalence (a semantic notion) coincide. Soundness says that lambda reduction rewrites a term in such a way that its denotation does not change, no matter how the interpretation is chosen. Completeness says that lambda reduction is powerful enough to decide semantic equivalence of terms.

For the Soundness Proposition one has to show the soundness of β-reduction ($\to_\beta\ \subseteq\ \approx$) and the soundness of η-reduction ($\to_\eta\ \subseteq\ \approx$). For the soundness of β-reduction the Substitution Lemma 4.1.3 is essential. The Completeness Theorem was first shown by Harvey Friedman [15] in 1975. A simpler proof by Gordon Plotkin [32] uses Tait's termination theorem.

4.3 Formulas and Models

We fix a sort B (read bool) and choose names for the **logical operations** (cf. Chapter 2):

$$\begin{aligned}
\bot, \top &: B \\
\neg &: BB \\
\vee, \wedge, \rightarrow &: BBB \\
=_\sigma &: \sigma\sigma B && \text{for every type } \sigma \\
\exists_\sigma, \forall_\sigma &: (\sigma B)B && \text{for every type } \sigma
\end{aligned}$$

Terms whose type is B are called **formulas**. Formulas of the form $s =_\sigma t$ are called **equations**. If the context suffices for disambiguation, we omit the type subscripts of $=_\sigma$, $\exists_\sigma$, and $\forall_\sigma$. We write $s \neq t$ for $\neg(s = t)$, $\equiv$ for $=_B$, and $s \not\equiv t$ for $\neg(s \equiv t)$. The operator precedences stated in § 2.3 apply.

A **logical interpretation** is an interpretation that interprets B as $\mathbb{B}$, $\bot$ as 0, $\top$ as 1, and every name for a logical operation o as the operation o. In particular,

$$\begin{aligned}
\top &\equiv (\lambda x.x) =_{BB} \lambda x.x\\
\bot &\equiv (\lambda x.x) =_{BB} \lambda x.\top\\
\neg &= \lambda x.\, x{=}\bot\\
\forall_\sigma &= \lambda f.\, f{=}\lambda x.\top\\
\exists_\sigma &= \lambda f.\, f{\neq}\lambda x.\bot\\
\wedge &= \lambda xy.\, \forall f.\, fxy \equiv f\top\top\\
\vee &= \lambda xy.\, \neg(\neg x \wedge \neg y)\\
\rightarrow &= \lambda xy.\, \neg x \vee y
\end{aligned}$$

Figure 4.1: Henkin Equations

we have $\mathcal{I}(=_\sigma) = \lambda v_1{\in}\mathcal{I}\sigma.\, \lambda v_2{\in}\mathcal{I}\sigma.\, (v_1{=}v_2)$ for every logical interpretation $\mathcal{I}$ and every type σ.

An interpretation $\mathcal{I}$ **satisfies a formula** s if $\mathcal{I}$ is logical and $\hat{\mathcal{I}}s = 1$. A formula is **satisfiable** if there is a logical interpretation that satisfies it, and **valid** if it satisfied by every logical interpretation. A formula is **unsatisfiable** if it is not satisfiable. An interpretation is a **model** of a formula if it satisfies the formula. We write $\mathcal{I} \vDash s$ if s is a formula and $\mathcal{I}$ satisfies s. We say that s **holds in** $\mathcal{I}$ if $\mathcal{I} \vDash s$.

Figure 4.1 shows a set of valid equations, which we will refer to as **Henkin equations** (cf. Exercise 2.10.3). The equations are stated with schemes, where a scheme parameterized with a type σ describes an equation for every type σ. The Henkin equations can be taken as definitions of the logical operations that appear at the left.

Proposition 4.3.1 A formula s is valid if and only if $\neg s$ is unsatisfiable.

The language given by well-typed terms, formulas and logical interpretations is called **simple type theory**. It originated 1940 with Church [10]. Major contributions came from Leon Henkin [20, 21]. Andrews [1] is the best availble textbook that introduces simple type theory in Church-Henkin style. Simple type theory provides the logical base for proof assistants like Isabelle [29] and HOL [19]. Simple type theory is one prominent example of a **higher-order logic**, but often higher-order logic or HOL are used as synonyms for simple type theory. The decomposition of simple type theory into the simply typed lambda calculus (interpretations) and the logic component (logical interpretations) introduced in this section is not standard although obvious. The syntactic part of the (simply typed) lambda calculus has become the base of the theory of programming languages [31, 28].

In the following we will mostly use simple type theory as the underlying framework. To simplify our language we will refer to logical interpretations simply as interpretations.

4.4 Specification of the Natural Numbers

Can we specify the natural numbers in simple type theory? Yes, we can. To see how, let's first ask another question: What are the natural numbers? For centuries, mathematicians have just assumed that the natural numbers exist and where happy to study their properties. Nowadays, sets are taken as starting point and hence we can define the natural numbers. Given sets, we have many possibilities to construct the natural numbers. Maybe the most straightforward possibility is to see the natural numbers as the sets

$$\emptyset,\ \{\emptyset\},\ \{\{\emptyset\}\},\ \{\{\{\emptyset\}\}\},\ \dots$$

where $\emptyset$ is taken as 0, $\{\emptyset\}$ as 1, and so on. Based on this construction, one can define addition, multiplication, and the order relation.

In simple type theory, we cannot mimic the set-theoretic construction of $\mathbb{N}$. However, a different, more abstract approach works that you may know from abstract data types. We start with a sort N and two names $O : N$ and $S : NN$. Our goal is a formula nat such that every interpretation that interprets N as $\mathbb{N}$, O as 0, and S as $\lambda n{\in}\mathbb{N}.n+1$ satisfies nat. We call such interpretations **canonical**. The function $\lambda n{\in}\mathbb{N}.n + 1$ is known as **successor function**.

We need to require more of nat since the formula $\top$ satisfies what we have said so far. What we need in addition is that for every formula s the formula nat $\to$ s is valid if and only if s is satisfied by every canonical model of nat. Since $O{\neq}SO$ is satisfied by every canonical model but $\top \to O{\neq}SO$ is not valid, $\top$ doesn't suffice for nat.

We cannot expect to find a formula nat whose models are exactly the canonical models. The reason is that formulas cannot distinguish the many equivalent constructions of the natural numbers. We call an interpretation **quasi-canonical** if it is a canonical up to the fact that it uses a different construction of $\mathbb{N}$.

So what is a construction of the natural numbers? We require that we are given a set N, a value $O \in N$, and a function $S \in N \to N$ such that two conditions are satisfied:

1. The values $O,\ SO,\ S(SO),\ S(S(SO)),\ \dots$ are pairwise distinct.
2. $N = \{O,\ SO,\ S(SO),\ S(S(SO)),\ \dots\}$

The second condition says that every $x \in N$ can be obtained from O by applying S a certain number of times. The first condition says that S each time yields a new value in N. This ensures that N is infinite.

It is not difficult to express the first condition in simple type theory. It suffices to say that S is injective and always yields a value that is different from O:

$$\begin{aligned} \forall xy.\ \ & Sx{=}Sy \to x{=}y \\ \forall x.\ \ & Sx{\neq}O \end{aligned}$$

The second condition requires an insight. We need to say that every element of N is reachable from O with S. We can do this by saying that every subset of N that contains O and is closed under S is the full set N:

$$\forall p.\ \ pO \wedge (\forall x.\, px \to p(Sx)) \to \forall x.px$$

This formula was first given by Guiseppe Peano in 1889 [30] and is known as **induction axiom**. Read the article *Peano axioms* in Wikipedia to know more. The conjunction of the three formulas stated above yields the formula nat we were looking for.

So far, we have the natural numbers just with O and S. Given O and S, it is straightforward to specify addition $+ : NNN$:

$$\begin{aligned} \forall y.\ \ & O + y = y \\ \forall xy.\ \ & Sx + y = x + Sy \end{aligned}$$

The two formulas specify the meaning of the name $+$ by recursion over the first argument. Recursion is a fundamental programming technique that has been used by mathematicians for a long time. In case you don't feel comfortable with recursion, get yourself acquainted with functional programming (there are plenty of textbooks, ML and Haskell are the most popular languages).

Exercise 4.4.1 (Multiplication) Extend the specification of the natural number with a formula that specifies the name $\cdot : NNN$ as multiplication.

Exercise 4.4.2 (Pairs) Let the names pair : $\sigma\tau P$, fst : $P\sigma$, and snd : $P\tau$ be given. Find a formula that is satisfied by a logical interpretation $\mathcal{I}$ if and only if $\mathcal{I}P \cong \mathcal{I}\sigma \times \mathcal{I}\tau$ and pair, fst, and snd are interpreted as the pairing and projection functions.

Exercise 4.4.3 (Termination) Let $r : \alpha\alpha B$ be a name. Find a formula that is satisfied by a logical interpretation $\mathcal{I}$ if and only if $\mathcal{I}r$ is the functional coding of a terminating relation.

Exercise 4.4.4 (Finiteness) Let $f : \sigma\sigma$ be a name.

a) Find a term injective : $(\sigma\sigma)B$ such that a logical interpretation satisfies the formula injective f if and only if it interprets f as an injective function.

b) Find a term $\mathsf{surjective} : (\sigma\sigma)B$ such that a logical interpretation satisfies the formula $\mathsf{surjective}\, f$ if and only if it interprets f as a surjective function.

c) Find a formula finite that is satisfied by a logical interpretation $\mathcal{I}$ if and only if $\mathcal{I}\sigma$ is a finite set.

Exercise 4.4.5 (Lists) Let the names $\mathsf{nil} : L$, $\mathsf{cons} : \sigma LL$, $\mathsf{hd} : L\sigma$, and $\mathsf{tl} : LL$ be given. Find a formula that is satisfied by a logical interpretation $\mathcal{I}$ if and only if L represents all lists over σ and nil, cons, hd, and tl represent the list operations. Make sure that L contains no junk elements.

5 Formal Proofs

There are two essential requirements for a proof system:

- *Soundness:* If a formula has a proof, the formula must be valid.
- *Decidability:* Given a proof object p and a formula s, it must be algorithmically decidable whether p is a proof of s.

The first proof system was devised by Frege in 1879 [13]. Fifty years later, Gentzen [16] invented the now predominant sequent-based proof systems.

5.1 Abstract Proof Systems

We start with an abstract notion of proof system. A **proof step** is a pair $(\{x_1, \dots, x_n\}, x)$, which may be written as

$$\frac{x_1 \quad \dots \quad x_n}{x}$$

The objects $x_1, \dots, x_n$ are the **premises** and the object x is the **conclusion** of the proof step. A proof step is **primitive** if it has no premises. A **proof system** is a set of proof steps. The premises and the conclusions of the steps of a proof system are jointly refered to as **propositions**. Given a proof system S and a set P, the **closure** $S[P]$ is defined recursively:

1. If $x \in P$, then $x \in S[P]$.
2. If $(Q, x) \in S$ and $Q \subseteq S[P]$, then $x \in S[P]$.

Due to the recursive definition of closures, we obtain **proof trees** that verify statements of the form $x \in S[P]$. The proof tree

$$\frac{\dfrac{\dfrac{}{x_1} \qquad x_2}{x_4} \qquad \dfrac{\dfrac{}{x_3}}{x_5}}{x_6}$$

verifies the statement $x_6 \in S[\{x_2\}]$ provided the following pairs are proof steps of S: $(\emptyset, x_1)$, $(\emptyset, x_3)$, $(\{x_1, x_2\}, x_4)$, $(\{x_3\}, x_5)$, $(\{x_4, x_5\}, x_6)$. Obviously, we have $x \in S[P]$ if and only if there is a proof tree that verifies $x \in S[P]$.

Proposition 5.1.1 Let S be a proof system. Then:

1. $P \subseteq S[P]$
2. $P \subseteq Q \implies S[P] \subseteq S[Q]$
3. $Q \subseteq S[P] \implies S[P \cup Q] = S[P]$

In practice, proof systems are supposed to be decidable. To this goal a decidable set X of propositions is fixed and S is chosen as a decidable set of proof steps for X. Given a decidable set $P \subseteq X$, it is decidable whether a given tree is a proof tree that verifies $x \in S[P]$. Consequently the closure $S[P]$ is semi-decidable.

A proposition x is **derivable** in a proof system S if $x \in S[\emptyset]$. A proof step (P, x) is **derivable** in a proof system S if $x \in S[P]$. Derivability of a proof step means that it can be simulated with proof steps that are in S. If we extend a proof system with derivable steps, the closures do not change. However, we may obtain smaller proof trees for given x and P. A proof tree that uses derivable rules can always be compiled into a proof tree just using basic rules.

Let V be a set. A proof step (P, x) **applies to** V if $P \subseteq V$. A proof step (P, x) is **sound for** V if $x \in V$ if (P, x) applies to V. A proof system S is **sound for** V if every proof step of S is sound for V.

Proposition 5.1.2 A proof system S is sound for V if and only if $S[V] = V$.

Proposition 5.1.3 If S is sound for V, then $S[\emptyset] \subseteq V$.

Exercise 5.1.4 Let the proof system $S = \{ (\{x, y\}, z) \mid x, y, z \in \mathbb{N} \wedge x \cdot y = z \}$ be given.

a) Determine $S[\emptyset]$.

b) Determine $S[\{2\}]$.

c) Give a proof step $(\{x\}, y) \in S$ that has only one premise.

d) Derive the proof step $(\{2, 3\}, 12)$.

e) Is S sound for the even numbers?

f) Is S sound for the odd numbers?

g) Does $S[\{2x | x \in \mathbb{N}\}]$ contain an odd number?

5.2 Deducible Sequents

Our goal are formal proofs for the validity of formulas. The obvious approach is to establish a proof system for formulas that is sound for the set of valid formulas. Systems of this kind are known as *Hilbert systems*. The first such system was developed by Frege [13]. Hilbert systems have the disadvantage that their proof trees are difficult to construct. The reason is that with Hilbert systems one cannot obtain proof trees whose structure resembles natural proofs.

Natural proofs work with assumptions. For instance, if we want to proof an implication $s \Rightarrow t$, we say "assume s" and then prove t under the assumption s. Similarly, to prove a statement $\forall x{\in}X\colon s$, we say "let $x \in X$" and then prove s under the assumption $x \in X$. Gerhard Gentzen [16] was the first who designed proof systems that maintain assumptions in the way natural proofs do. To make this possible, Gentzen uses sequents rather than formulas as propositions.

A **sequent** is a pair $A \mathrel{\dot\vdash} s$ where A is a finite set of formulas and s is a formula. The formulas in A are called the **assumptions** of the sequent, and s is called the **claim** of the sequent. A sequent is **valid** if every logical interpretation that satisfies all assumptions of the sequent also satisfies the claim of the sequent. We write $\vDash$ for the set of all valid sequents, and $A \vDash s$ if $A \mathrel{\dot\vdash} s$ is valid.

Proposition 5.2.1

1. A formula s is valid if and only if the sequent $\emptyset \mathrel{\dot\vdash} s$ is valid.
2. A sequent $\{s_1, \ldots, s_n\} \mathrel{\dot\vdash} s$ is valid if and only if the formula $s_1 \to \cdots \to s_n \to s$ is valid.

We refer to the names for the logical operations (cf. § 4.3) as **constants**. All other names are called **variables**. From now on we tacitly assume the following:

- Constants are not used as local names.
- Substitutions leave constants unchanged. That is, $\theta c = c$ for every constant c and every substitution θ.
- Interpretations are always logical.

Some conventions and definitions:

- A always denotes a finite set of formulas.
- A context C **captures** a name x if C's hole is in the scope of a λx. For instance, the context $\lambda x.(\lambda y.[\,])x$ captures x and y but no other name.
- A context C is **admissible for** A if C captures no $x \in \mathcal{N}A$.
- A substitution θ is **admissible for** A if $\theta x = x$ for all $x \in \mathcal{N}A$.
- $\theta s := \mathcal{S}\theta s$
- $\theta A := \{\, \theta s \mid s \in A \,\}$

We define the following notations for sequents:

- $\dot\vdash s$ for $\emptyset \mathrel{\dot\vdash} s$.
- $A, s \mathrel{\dot\vdash} t$ for $A \cup \{s\} \mathrel{\dot\vdash} t$.
- $s, t \mathrel{\dot\vdash} u$ for $\{s, t\} \mathrel{\dot\vdash} u$.

$$\textbf{Triv}\quad \frac{}{A,\, s \vdash s} \qquad \textbf{Weak}\quad \frac{A \vdash s}{B \vdash s}\; A \subseteq B \qquad \textbf{Sub}\quad \frac{A \vdash s}{\theta A \vdash \theta s} \qquad \textbf{Lam}\quad \frac{A \vdash s}{A \vdash t}\; s \sim_\lambda t$$

$$\textbf{Ded}\quad \frac{A, s \vdash t}{A \vdash s \to t} \qquad\qquad \textbf{MP}\quad \frac{A \vdash s \to t \qquad A \vdash s}{A \vdash t}$$

$$\textbf{Ref}\quad \frac{}{A \vdash s{=}s} \qquad\qquad \textbf{Rew}\quad \frac{A \vdash s{=}t \qquad A \vdash C[s]}{A \vdash C[t]}\; C \text{ admissible for } A$$

$$\textbf{D}\top \quad \vdash \top \equiv \bot{\to}\bot \qquad\qquad \textbf{D}\neg \quad \vdash \neg x \equiv x{\to}\bot$$

$$\textbf{D}\vee \quad \vdash x{\vee}y \equiv \neg x{\to}y \qquad\qquad \textbf{D}\wedge \quad \vdash x{\wedge}y \equiv \neg(\neg x{\vee}\neg y)$$

$$\textbf{D}\forall \quad \vdash \forall f \equiv f{=}\lambda x.\top \qquad\qquad \textbf{D}\exists \quad \vdash \exists f \equiv \neg\forall x.\neg f x$$

$$\textbf{BCA} \quad f\bot,\; f\top \vdash f x \qquad\qquad \textbf{Choice} \quad \vdash \exists c.\forall f.\, \exists f \to f(cf)$$

Figure 5.1: The basic proof system **B** defining $\vdash$

Figure 5.1 defines a proof system **B** for sequents. We refer to this system as basic proof system. The proof steps of **B** are described by means of schemes called **rules**. Given a rule, the proof steps described by the rule are obtained by taking the sequents above the horizontal line as premises and the sequent below the horizontal line as conclusion. The rules Lam and Rew come with side conditions that constrain the obtainable proof steps. Starting with D$\neg$, we omit the horizontal line for primitive rules (i.e., rules without premises). Following a common abuse of notation, we write the sequents appearing as premises and conclusions of the rules with $\vdash$ rather than $\dot\vdash$. Every rule comes with a name that appears to its left.

Proposition 5.2.2 The proof system **B** is sound for $\models$.

Proof We have to check for every rule in Figure 5.1 that the conclusion is valid if all the premises are valid. It is ok to think of A as a formula and of $\dot\vdash$ as implication. With this provisio the soundness of most rules should be intuitively clear. The soundness of Lam follows with Proposition 4.2.1. ■

We define $\vdash := \mathbf{B}[\emptyset]$ and write $A \vdash s$ for $(A \mathrel{\dot\vdash} s) \in \vdash$. A sequent $A \mathrel{\dot\vdash} s$ is **deducible** if $A \vdash s$. Since **B** is sound for $\models$, deducible sequents are valid and $\vdash \subseteq \models$. The symbol $\vdash$ originated with Frege and is pronounced *turnstile*.

It is important to distinguish between the set $\vdash$ of deducible sequents and the proof system **B** defining it. The set $\vdash$ is canonical and we will work with it for a long time. The proof system **B** is not canonical. The set of deducible sequents can be defined with many other proof systems.

We will study **B** and $\vdash$ by deriving more and more useful proof steps. We start with some remarks on the rules given in Figure 5.1. The first line lists general rules that are not committed to particular logic operations. The rules Ded and MP in the second line concern implication. Next follow the rules Ref and Rew, which concern identities. The next 6 rules act as equational definitions of $\top$, $\neg$, $\vee$, $\wedge$, $\forall$, and $\exists$. The final two rules BCA and Choice state properties that cannot be obtained with the previous rules. BCA says that a function $f : BB$ that yields $\top$ for both $\bot$ and $\top$ yiels $\top$ for every $x : B$. Choice states the existence of choice functions.

The full names of the rules are as follows: Triviality, Weakening, Substitution, Lambda, Deductivity, Modus Ponens, Reflexivity, Rewriting, and Boolean Case Analysis. The D stands for definition.

Example 5.2.3 Here is a proof tree that verifies $(x{\to}x){\to}x \vdash x$:

$$\dfrac{\dfrac{}{(x{\to}x){\to}x \vdash (x{\to}x){\to}x}\,\text{Triv} \qquad \dfrac{\dfrac{}{(x{\to}x){\to}x,\ x \vdash x}\,\text{Triv}}{(x{\to}x){\to}x \vdash x{\to}x}\,\text{Ded}}{(x{\to}x){\to}x \vdash x}\,\text{MP}$$

5.3 Derived Rules for Turnstile and Implication

The rules in the first 2 lines of Figure 5.1 constitute the kernel of the basic proof system. They fix the basic properties of $\vdash$ and $\to$. Figure 5.2 shows additional rules that are derivable from this set of basic rules. A rule is **derivable** if all its proof steps are derivable.

Derivation of Ded$^-$

Ded$^-$ acts as the inverse of Ded. It is derivable with the following proof tree:

$$\dfrac{\dfrac{A \vdash s{\to}t}{A, s \vdash s{\to}t}\,\text{Weak} \qquad \dfrac{}{A, s \vdash s}\,\text{Triv}}{A, s \vdash t}\,\text{MP}$$

The best way to construct and understand a proof tree is backwards, that is, from the root to the leaves. To prove a sequent, one looks for a rule that yields

$$\textbf{Ded}^- \ \frac{A \vdash s{\to}t}{A, s \vdash t} \qquad \textbf{Lam'} \ \frac{A, u \vdash t}{A, s \vdash t} \ u \sim_\lambda s \qquad \textbf{Cut} \ \frac{A \vdash s \qquad A, s \vdash t}{A \vdash t}$$

$$\textbf{Cut} \ \frac{s \vdash t \qquad A \vdash s}{A \vdash t} \qquad \textbf{Cut} \ \frac{s_1, s_2 \vdash t \qquad A \vdash s_1 \qquad A \vdash s_2}{A \vdash t}$$

$$\textbf{MP'} \ \frac{A \vdash s \qquad A, t \vdash u}{A,\ s{\to}t \vdash u} \qquad \textbf{MP'} \ \frac{A, t \vdash u}{A,\ s,\ s{\to}t \vdash u}$$

Figure 5.2: Derived rules for $\vdash$ and $\to$

the sequent from hopefully weaker premises. One then continues recursively with proving the premises.

Derivation of Cut

The first cut rule is derivable as follows:

$$\dfrac{\dfrac{A, s \vdash t}{A \vdash s{\to}t}\,\textsf{Ded} \qquad A \vdash s}{A \vdash t}\,\textsf{MP}$$

It's best to read the cut rules backwards. The first cut rule says that $A \vdash t$ can be proven by using a "lemma" $A \vdash s$ and proving the weaker statement $A, s \vdash t$.

Exercise 5.3.1 Show that Lam' is derivable. Exploit that with Ded$^-$ assumptions can be shifted to the right of $\vdash$, and that with Ded they can be shifted back.

Exercise 5.3.2 Prove the second and the third cut rule.

Exercise 5.3.3 Prove MP'.

Exercise 5.3.4 Let θ be admissible for A and $A \vdash s$. Show $A \vdash \theta s$.

Exercise 5.3.5 Show $x \vdash (x{\to}\bot){\to}\bot$.

5.4 Derived Rules for Identities

We now look at the basic rules Ref and Rew which handle identities. Figure 5.3 shows the most important derived rules for identities.

$$\textbf{Sym}\ \frac{A \vdash s{=}t}{A \vdash t{=}s} \qquad\qquad \textbf{Trans}\ \frac{A \vdash s{=}t \qquad A \vdash t{=}u}{A \vdash s{=}u}$$

$$\textbf{Con}\ \frac{A \vdash s{=}t}{A \vdash C[s]{=}C[t]}\ C \text{ admissible for } A$$

$$\textbf{Rep}\ \frac{A \vdash s{=}t \qquad A \vdash C[\theta s]}{A \vdash C[\theta t]}\quad C,\ \theta \text{ admissible for } A$$

$$\textbf{Rep}\ \frac{A \vdash s{=}t \qquad A \vdash C[\theta t]}{A \vdash C[\theta s]}\quad C,\ \theta \text{ admissible for } A$$

$$\textbf{Rep}\ \frac{A \vdash s{=}u \qquad A, C[\theta s] \vdash t}{A, C[\theta u] \vdash t}\quad C,\ \theta \text{ admissible for } A$$

$$\textbf{Rep}\ \frac{A \vdash s{=}u \qquad A, C[\theta u] \vdash t}{A, C[\theta s] \vdash t}\quad C,\ \theta \text{ admissible for } A$$

$$\textbf{FE}\ \frac{A \vdash sx{=}tx}{A \vdash s{=}t}\ x \notin \mathcal{N}A \cup \mathcal{N}(s{=}t)$$

$$\textbf{Abs}\ \frac{A \vdash fx{=}s}{A \vdash f{=}\lambda x.s}\ x \notin \mathcal{N}A \qquad\qquad \textbf{Abs}\ \frac{A \vdash fxy{=}s}{A \vdash f{=}\lambda xy.s}\ x, y \notin \mathcal{N}A$$

Figure 5.3: Derived rules for identities

Derivation of Sym

$$\dfrac{A \vdash s{=}t \qquad \dfrac{}{A \vdash s{=}s}\,\textbf{Ref}}{A \vdash t{=}s}\,\textbf{Rew}$$

Derivation of Con

$$\dfrac{A \vdash s{=}t \qquad \dfrac{}{A \vdash C[s]{=}C[s]}\,\textbf{Ref}}{A \vdash C[s]{=}C[t]}\,\textbf{Rew}$$

Top $\vdash \top$ **BCAR** $\dfrac{A \vdash s^x_\bot \qquad A \vdash s^x_\top}{A \vdash s}$ **Bot** $\bot \vdash s$

Equiv $\dfrac{A, s \vdash t \qquad A, t \vdash s}{A \vdash s \equiv t}$ **Eq** $\vdash x \equiv x = \top$ **DN** $\vdash \neg\neg x \equiv x$

Contra $\dfrac{A, \neg s \vdash \bot}{A \vdash s}$ **Contraposition** $\vdash x \to y \equiv \neg y \to \neg x$ **Icon** $x, \neg x \vdash y$

N⊥ $\vdash \neg\bot \equiv \top$ **N⊤** $\vdash \neg\top \equiv \bot$ **N≡** $\vdash x \not\equiv y \equiv x = \neg y$

Figure 5.4: Derived Rules for ⊤ and ¬

Derivation of FE

$$\dfrac{\dfrac{A \vdash sx = tx}{A \vdash (\lambda x.sx) = \lambda x.tx}\,\textbf{Con}}{A \vdash s = t}\,\textbf{Lam}$$

Exercise 5.4.1 Prove $x = y \vdash y = x$ with the basic rules.

Exercise 5.4.2 Show that Trans, Rep, and Abs are derivable.

5.5 BCA and Tautologies

A formula is **propositional** if it can be obtained with the following grammar:

$$s ::= \bot \mid \top \mid p \mid \neg s \mid s \to s \mid s \wedge s \mid s \vee s \mid s \equiv s$$
$$p : B \quad \text{is a variable}$$

A **tautology** is a propositional formula that is valid. Typical examples of tautologies are ⊤ and $x \wedge y \equiv y \wedge x$. We will show that all tautologies are deducible. For this result the rule BCA is essential.

Figure 5.4 shows rules that are derivable in **B** with BCA.

Derivation of Top

$$\dfrac{\dfrac{}{\vdash \top \equiv \bot \to \bot}\,\textbf{D}\top \qquad \dfrac{\dfrac{}{\bot \vdash \bot}\,\textbf{Triv}}{\vdash \bot \to \bot}\,\textbf{Ded}}{\vdash \top}\,\textbf{Rep}$$

Derivation of BCAR

The sequent $s^x_\bot,\ s^x_\top \vdash s$ can be derived as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f\bot,\ f\top \vdash fx}\,\text{BCA}}{(\lambda x.s)\bot,\ (\lambda x.s)\top \vdash (\lambda x.s)x}\,\text{Sub}}{s^x_\bot,\ s^x_\top \vdash (\lambda x.s)x}\,\text{Lam', Lam'}}{s^x_\bot,\ s^x_\top \vdash s}\,\text{Lam}$$

Now BCAR can be derived with Cut.

Derivation of Bot

$$\dfrac{\dfrac{\dfrac{}{\bot \vdash \bot}\,\text{Triv} \qquad \dfrac{\dfrac{}{\vdash \top}\,\text{Top}}{\bot \vdash \top}\,\text{Weak}}{\bot \vdash x}\,\text{BCAR}}{\bot \vdash s}\,\text{Sub}$$

Derivation of Equiv

With Sub, Cut, and Ded we can derive Equiv from $\vdash (x\to y) \to (y\to x) \to x\equiv y$. This sequent can be derived with BCAR from the four instances obtained by replacing x and y with $\bot$ and $\top$. Two of the instances will end with trivial equations and can thus be obtained with Ded and Ref. Here is a derivation of one of the remaining instances:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \top}\,\text{Top} \qquad \dfrac{}{\bot \vdash \bot}\,\text{Triv}}{\top\to\bot \vdash \bot}\,\text{MP'}}{\top\to\bot \vdash \bot\equiv\top}\,\text{Bot, Cut}}{\vdash (\bot\to\top) \to (\top\to\bot) \to \bot\equiv\top}\,\text{Weak, Ded, Ded}$$

Derivation of DN

Here is a proof tree for DN that has one open premise.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash ((x\to\bot)\to\bot) \to x}{(x\to\bot)\to\bot \vdash x}\,\text{Ded}^- \qquad \dfrac{\dfrac{\dfrac{}{\bot \vdash \bot}\,\text{Triv}}{x,\ x\to\bot \vdash \bot}\,\text{MP'}}{x \vdash (x\to\bot)\to\bot}\,\text{Ded}}{\vdash (x\to\bot)\to\bot \equiv x}\,\text{Equiv}}{\vdash \neg x\to\bot \equiv x}\,\text{Rep with D}\neg}{\vdash \neg\neg x \equiv x}\,\text{Rep with D}\neg$$

The open premise can be shown with BCAR.

Exercise 5.5.1 Prove $\vdash ((x \to \bot) \to \bot) \to x$ with BCAR. Don't use DN.

Derivation of Contra

$$\dfrac{\dfrac{\dfrac{A,\ \neg s \vdash \bot}{A \vdash \neg s \to \bot}\ \textbf{Ded}}{A \vdash \neg\neg s}\ \textbf{Rep with D}\neg}{A \vdash s}\ \textbf{Rep with DN}$$

Exercise 5.5.2 Derive $\vdash \neg s \equiv s = \bot$.

5.6 Natural Deduction

We already mentioned that sequent-based proof systems were invented by Gentzen [16]. His goal was a system whose proof rules are natural in the sense that they formalize proof patterns used in mathematical proofs. Figure 5.5 shows a proof system **ND** for sequents that closely corresponds to one of Gentzen's systems. **ND** is a well-known proof system of theoretical and practical importance. Its rules can be found in most proof assistants (e.g, Isabelle/HOL [29]).

Proposition 5.6.1 The proof steps of **ND** are derivable in **B**.

We will derive some of the rules of **ND**, but this is not our main interest. Rather, we want to explain the structure behind the rules and how the rules relate to mathematical proof patterns.

First, we observe that $\top$, negation, and identities do not explicitly occur in **ND**. Consequently, **ND** is weaker than **B** in that it can prove less sequents. The omission of $\top$ and negation is not essential since they can be treated as abbreviations for $\bot \to \bot$ and $s \to \bot$ (cf. D$\neg$ and D$\top$ in Figure 5.1). The omission of identities, however, forgives expressive power and important proof patterns. The obmission of identities is typical for natural deduction-like proof systems.

ND has exactly two rules for each of the logical operations $\to$, $\wedge$, $\vee$, $\forall$, and $\exists$. Let's refer to these operations as **regular**. An important modularity property of **ND** is the fact that the rules for a regular operation do not employ other operations. For every regular operation the basic proof steps are thus provided by exactly two rules that don't employ the other operations. The exception to this pattern is the rule CB, which employs $\bot$ and implication. CB is the only rule that employs $\bot$. Note that CB is a variant of the Contra rule.

$$\textbf{Triv}\quad \frac{}{A,\ s \vdash s} \qquad\qquad \textbf{Weak}\quad \frac{A \vdash s}{B \vdash s}\ A \subseteq B$$

$$\textbf{Ded}\quad \frac{A, s \vdash t}{A \vdash s \to t} \qquad\qquad \textbf{MP}\quad \frac{A \vdash s \to t \qquad A \vdash s}{A \vdash t}$$

$$\textbf{I}\land\quad \frac{A \vdash s_1 \qquad A \vdash s_2}{A \vdash s_1 \land s_2} \qquad\qquad \textbf{E}\land\quad \frac{A \vdash s_1 \land s_2}{A \vdash s_i}$$

$$\textbf{I}\lor\quad \frac{A \vdash s_i}{A \vdash s_1 \lor s_2} \qquad\qquad \textbf{E}\lor\quad \frac{A \vdash s_1 \lor s_2 \qquad A, s_1 \vdash t \qquad A, s_2 \vdash t}{A \vdash t}$$

$$\textbf{I}\forall\quad \frac{A \vdash s^x_y}{A \vdash \forall x.s}\ y \notin \mathcal{N}A \cup \mathcal{N}(\forall x.s) \qquad\qquad \textbf{E}\forall\quad \frac{A \vdash \forall x.s}{A \vdash s^x_t}$$

$$\textbf{I}\exists\quad \frac{A \vdash s^x_t}{A \vdash \exists x.s} \qquad\qquad \textbf{E}\exists\quad \frac{A \vdash \exists x.s \qquad A, s^x_y \vdash t}{A \vdash t}\ y \notin \mathcal{N}A \cup \mathcal{N}(\exists x.s) \cup \mathcal{N}t$$

$$\textbf{CB}\quad \frac{A,\ s \to \bot \vdash \bot}{A \vdash s}$$

Figure 5.5: The natural deduction system **ND**

One distinguishes between **introduction and elimination rules**. For every regular operation, **ND** has one introduction and one elimination rule. In Figure 5.5, the introduction rules appear left and the elimination rules appear right. For implication, the introduction rule is Ded and the elimination rule is MP. For every regular operation o, the introduction rule formalizes the natural pattern for proving formulas built with o. The corresponding elimination rule formalizes a natural pattern for making use of an already proven formula built with o. Note that the logical operations appear exclusively in the claims of the sequents appearing as premises and conclusions of the introduction and elimination rules.

The introduction rules can be paraphrased as follows:

- To prove $s \to t$, assume s and prove t.
- To prove $s_1 \land s_2$, prove both s_1 and s_2.
- To prove $s_1 \lor s_2$, prove either s_1 or s_2.
- To prove $\forall x.s$, prove s^x_y for some fresh name y.

- To prove $\exists x.s$, prove s^x_t for some term t. The term t is referred to as **witness**.

We now see why one speaks of natural deduction: The introduction rules in fact formalize common proof patterns from mathematical proofs. This is also the case for the elimination rules, which can be paraphrased as follows: If you have already proven

- $s \to t$, you can prove the claim t by proving s.
- $s_1 \wedge s_2$, you have also proven s_1 and s_2.
- $s_1 \vee s_2$, you can prove a claim t by **case analysis**: first with the assumption s_1, then with the assumption s_2.
- $\forall x.s$, you have also proven every **instance** s^x_t.
- $\exists x.s$, you can prove a claim t with the additional assumption s^x_y where y is a fresh name.

Proposition 5.6.2 Ded$^-$, Cut, and MP' are derivable in **ND**.

Proof Follows with proof trees for **B** in § 5.3 since only the common rules Triv, Weak, Ded, and MP are used. ■

Proposition 5.6.3 Sub is derivable in **ND**.

Proof We show how Sub can be derived if only one variable is replaced.

$$\dfrac{\dfrac{\dfrac{\dfrac{s \vdash t}{\vdash s \to t}\,\text{Ded}}{\vdash \forall x.s \to t}\,\text{I}\forall}{\vdash s^x_u \to t^x_u}\,\text{E}\forall}{s^x_u \vdash t^x_u}\,\text{Ded}^-$$

The proof can be generalized to the case where several variables are replaced. ■

Which rules do we have to add to **ND** so that we can derive all rules of **B**? One can show that Lam, Ref, Rew, BCA, and Choice do the job. If we add BCA, we can drop CB since CB can be derived with **ND** and BCA (see the derivation of Contra in **B** in § 5.5). We now see that we can account for the operations $\wedge$, $\vee$, $\forall$ and $\exists$ in two ways: through defining equations as done by **B**, or through proof rules as done by **ND**.

Exercise 5.6.4 Consider the sequent $f(x\to x) \vdash f(y\to y)$. This sequent cannot be proven in **ND**. Sketch a proof in **B**. Derived rules are fine.

5.7 Predicate Logic and Completeness

A formula is called **first-order** if it can be obtained with the following grammar:

$$\begin{aligned} s &::= \bot \mid \top \mid p\,t\ldots t \mid t = t \mid \\ &\quad\;\; \neg s \mid s \to s \mid s \wedge s \mid s \vee s \mid s \equiv s \mid \forall x.s \mid \exists x.s \\ t &::= f\,t\ldots t \\ &\textit{where } p : \alpha\ldots\alpha B,\;\; x : \alpha,\;\; \text{and } f : \alpha\ldots\alpha\alpha \text{ are variables and } \alpha \neq B \end{aligned}$$

The logic obtained with propositional formulas is called **propositional logic**, and the one obtained with first-order formulas is called (first-order) **predicate logic**. Gentzen devised his natural deduction system for predicate logic without equality (i.e., without identities). Predicate logic was studied long before the more general simple type theory was invented. In a native account of predicate logic, both quantifiers act as variable binders, hence there is no need for lambda terms.

A sequent is **propositional** [**first-order**] if all its formulas are propositional [first-order]. A proof tree is **propositional** [**first-order**] if it only involves propositional [first-order] sequents.

Theorem 5.7.1 (Completeness) For every valid sequent S that doesn't contain $\top$, $\neg$ and identities, the following statements hold:

1. If S is first-order, then S can be derived with a first-order **ND** proof tree.
2. If S is propositional, then S can be derived with a propositional **ND** proof tree.

A theorem of this type was first shown by Gentzen [16]. The first completeness theorem for first-order formulas was shown by Gödel [18] (for a Hilbert system).

A proof system that is complete for all valid sequents cannot exist since the set of valid formulas is not semi-decidable. This follows from the fact that we can construct for every Turing machine M a formula s such that M holds for all inputs if and only if s is valid.

5.8 Natural Language Proofs and Natural Deduction Proofs

In this section we give examples of provable sequents, natural language proofs of the sequents, and the corresponding **ND** proof trees. We begin with a very simple example.

Example 5.8.1 $\vdash \forall x.px \to px$ where $x : \alpha$ and $p : \alpha B$.
Natural Language Proof: Let $x : \alpha$ be given. Assume px holds. By assumption,

px holds, so we are done.
(Instead of writing “we are done” we may also write “QED.”)
Formal Proof:

$$\dfrac{\dfrac{\dfrac{}{px \vdash px}\,\textbf{Triv}}{\vdash px \to px}\,\textbf{Ded}}{\vdash \forall x.px \to px}\,\textbf{I}\forall$$

Let’s consider a similar example involving a quantifier over the type B of Booleans.

Example 5.8.2 $\vdash \forall p.p \to p$ where $p : B$.
Natural Language Proof: Let p be a Boolean value. Assume p holds. By assumption, p holds. QED.
Formal Proof:

$$\dfrac{\dfrac{\dfrac{}{p \vdash p}\,\textbf{Triv}}{\vdash p \to p}\,\textbf{Ded}}{\vdash \forall p.p \to p}\,\textbf{I}\forall$$

When the sequent has a nonempty set assumptions, then we can make use of these assumptions in the proof of the conclusion. Let’s consider an example with a nonempty set of assumptions.

Example 5.8.3 $\mathsf{man}\,\mathsf{Socrates}, \forall x.\mathsf{man}x \to \mathsf{mortal}x \vdash \mathsf{mortal}\,\mathsf{Socrates}$
Natural Language Proof: Since Socrates is a man and all men are mortal, Socrates is mortal.
Formal Proof: Let A denote the set $\{\mathsf{man}\,\mathsf{Socrates}, \forall x.\mathsf{man}x \to \mathsf{mortal}x\}$.

$$\dfrac{\dfrac{\dfrac{}{A \vdash \forall x.\mathsf{man}x \to \mathsf{mortal}x}\,\textbf{Triv}}{A \vdash \mathsf{man}\,\mathsf{Socrates} \to \mathsf{mortal}\,\mathsf{Socrates}}\,\textbf{E}\forall \qquad \dfrac{}{A \vdash \mathsf{man}\,\mathsf{Socrates}}\,\textbf{Triv}}{A \vdash \mathsf{mortal}\,\mathsf{Socrates}}\,\textbf{MP}$$

“Socrates is mortal” is a common example of deductive reasoning dating back to Aristotle. The proof demonstrates a truth that is independent of the suggestive names man, mortal and $\mathsf{Socrates}$. Reconsider the example with less suggestive names. We still write “x is a p” to mean “px holds.” This makes sense in the Socrates example, since $\mathsf{man}\,\mathsf{Socrates}$ means Socrates is a man. Other ways to say “px holds” include “x has property p,” “x is in p” and “x satisfies p.”

Example 5.8.4 $pa, \forall x.px \to qx \vdash qa$
Natural Language Proof: Since a is a p and every p is a q, a is a q.
Formal Proof: Let A be the set $\{pa, \forall x.px \to qx\}$.

$$\dfrac{\dfrac{\dfrac{}{A \vdash \forall x.px \to qx}\,\text{Triv}}{A \vdash pa \to qa}\,\text{E}\forall \qquad \dfrac{}{A \vdash pa}\,\text{Triv}}{A \vdash qa}\,\text{MP}$$

So far we have only shown examples involving $\forall$ and $\to$. We next consider an example with $\land$ and $\lor$.

Example 5.8.5 $a \lor b \land c \vdash (a \lor b) \land (a \lor c)$ where $a, b, c : B$.
Natural Language Proof: By case analysis. Case 1: Assume a holds. In this case $a \lor b$ and $a \lor c$ both hold. Hence their conjunction holds. Case 2: Assume $b \land c$ holds. This means both b and c hold. Consequently, the disjunctions $a \lor b$ and $a \lor c$ both hold. Thus the conjunction holds and we are done.
Formal Proof: Let A be the set $\{a \lor b \land c\}$ and A' be the set $\{a \lor b \land c, b \land c, b, c\}$. The main proof tree is

$$\dfrac{\dfrac{}{A \vdash a \lor b \land c}\,\text{Triv} \quad \textbf{(Case 1)} \qquad \textbf{(Case 2)}}{A \vdash (a \lor b) \land (a \lor c)}\,\text{E}\lor$$

where **(Case 1)** is the proof tree

$$\dfrac{\dfrac{\dfrac{}{A, a \vdash a}\,\text{Triv}}{A, a \vdash a \lor b}\,\text{I}\lor \qquad \dfrac{\dfrac{}{A, a \vdash a}\,\text{Triv}}{A, a \vdash a \lor c}\,\text{I}\lor}{A, a \vdash (a \lor b) \land (a \lor c)}\,\text{I}\land$$

(Case 2) is the proof tree

$$\dfrac{\dfrac{\dfrac{}{A, b \land c \vdash b \land c}\,\text{Triv}}{A, b \land c \vdash b}\,\text{E}\land \quad \dfrac{\dfrac{\dfrac{}{A, b \land c, b \vdash b \land c}\,\text{Triv}}{A, b \land c, b \vdash c}\,\text{E}\land \quad \dfrac{\dfrac{\dfrac{}{A' \vdash b}\,\text{Triv}}{A' \vdash a \lor b}\,\text{I}\lor \quad \dfrac{\dfrac{}{A' \vdash c}\,\text{Triv}}{A' \vdash a \lor c}\,\text{I}\lor}{A' \vdash (a \lor b) \land (a \lor c)}\,\text{I}\land}{A, b \land c, b \vdash (a \lor b) \land (a \lor c)}\,\text{Cut}}{A, b \land c \vdash (a \lor b) \land (a \lor c)}\,\text{Cut}$$

We next consider the existential quantifier $\exists$.

Example 5.8.6 $pa \vdash \exists x.px$ where $x, a : \alpha$ and $p : \alpha B$.
Natural Language Proof: We must prove there is some p. We know there is a p since a is a p.
Formal Proof:

$$\dfrac{\dfrac{}{pa \vdash pa}\,\textbf{Triv}}{pa \vdash \exists x.px}\,\textbf{I}\exists$$

This example using $\exists$ was too easy. Let's consider one that requires a bit more work.

Example 5.8.7 $pa \vee pb, \forall x.px \to qx \vdash \exists x.qx$ where $x, a, b : \alpha$ and $p, q : \alpha B$.
Natural Language Proof: We must prove there is some q. We consider two cases. Case 1: Assume a is a p. Since every p is also a q, a is a q. Hence there is some q in this case. Case 2: Assume b is a p. Since every p is also a q, b is a q. Hence there is some q in this case as well and we are done.
Formal Proof: Let A be $\{pa \vee pb, \forall x.px \to qx\}$. The main proof tree is

$$\dfrac{\dfrac{}{A \vdash pa \vee pb}\,\textbf{Triv} \quad \textbf{(Case 1)} \quad \textbf{(Case 2)}}{A \vdash \exists x.qx}\,\textbf{E}\vee$$

where **(Case 1)** is the proof tree

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A, pa \vdash \forall x.px \to qx}\,\textbf{Triv}}{A, pa \vdash pa \to qa}\,\textbf{E}\forall \qquad \dfrac{}{A, pa \vdash pa}\,\textbf{Triv}}{A, pa \vdash qa}\,\textbf{MP}}{A, pa \vdash \exists x.qx}\,\textbf{I}\exists$$

and **(Case 2)** is the proof tree

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A, pb \vdash \forall x.px \to qx}\,\textbf{Triv}}{A, pb \vdash pb \to qb}\,\textbf{E}\forall \qquad \dfrac{}{A, pb \vdash pb}\,\textbf{Triv}}{A, pb \vdash qb}\,\textbf{MP}}{A, pb \vdash \exists x.qx}\,\textbf{I}\exists$$

The next example shows how to use an existential assumption.

Example 5.8.8 $\exists z.pz, \forall x.px \to q(fx) \vdash \exists y.qy$ where $x, y, z : \alpha$, $p, q : \alpha B$ and $f : \alpha\alpha$.
Natural Language Proof: We must prove there is some q. We know there is some p. Let a be a p. We know if x is a p then fx is a q. In particular, fa is a q. We conclude that there is some q. QED.
Formal Proof: Let A be $\{\forall x.px \to q(fx), \exists z.pz\}$.

$$\dfrac{\dfrac{}{A \vdash \exists z.pz}\,\textbf{Triv} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A, pa \vdash \forall x.px \to q(fx)}\,\textbf{Triv}}{A, pa \vdash pa \to q(fa)}\,\mathrm{E}\forall \qquad \dfrac{}{A, pa \vdash pa}\,\textbf{Triv}}{A, pa \vdash q(fa)}\,\textbf{MP}}{A, pa \vdash \exists y.qy}\,\mathrm{I}\exists}{A \vdash \exists y.qy}\,\mathrm{E}\exists$$

Question: Would the proof tree verify $A \vdash \exists y.qy$ if A were $\{a = a, \forall x.px \to q(fx), \exists z.pz\}$? □

The **Lam** and **Ref** rules are not part of the **ND** system. If we add them to the system, then we can prove the following example.

Example 5.8.9 $\vdash \exists f.\forall x.fx = x$ where $f : \alpha\alpha$ and $x : \alpha$.
Natural Language Proof: We must prove there is some function f such that $fx = x$ for any x. We can simply take f to be $\lambda x.x$. Note that for any x, $(\lambda x.x)x$ is the same as x up to λ-equivalence.
Formal Proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash x = x}\,\textbf{Ref}}{\vdash \forall x.x = x}\,\mathrm{I}\forall}{\vdash \forall x.(\lambda x.x)x = x}\,\textbf{Lam}}{\vdash \exists f.\forall x.fx = x}\,\mathrm{I}\exists$$

5.9 More Derivations

First we derive I∀ and E∀ in **B**.

Example 5.9.1 (Universal Generalization, I∀)

$$\dfrac{\dfrac{\dfrac{A \vdash s}{A \vdash s = \top}\,\textbf{Rep with Eq}}{A \vdash (\lambda x.s) = \lambda x.\top}\,\textbf{Con, } x \notin \mathcal{N}A}{A \vdash \forall x.s}\,\textbf{Rep with D}\forall$$

Example 5.9.2 (Universal Instantiation, E∀)

$$\begin{array}{cl} A \vdash \forall x.s & \\ \hline A \vdash (\lambda x.s) = \lambda x.\top & \textbf{Rep with D}\forall \\ \hline A \vdash (\lambda x.s)t = (\lambda x.\top)t & \textbf{Con} \\ \hline A \vdash s^x_t = \top & \textbf{Lam} \\ \hline A \vdash s^x_t & \textbf{Rep with Eq} \end{array}$$

Example 5.9.3 (D∀) While **B** accomodates $\wedge$, $\vee$, $\forall$, and $\exists$ through defining equations (the rules starting with D), **ND** accommodates these operations through introduction an elimination rules. Both approaches have the same power. We have just shown that I∀ and E∀ are derivable with D∀. We now show that D∀ is derivable with I∀ and E∀. Because of Equiv it suffices to show $\forall f \vdash f = \lambda x.\top$ and $f = \lambda x.\top \vdash \forall f$. We show the former and leave the latter as an exercise.

$$\begin{array}{cl} \hline \forall x.fx \vdash fx & \textbf{Triv, E}\forall \\ \hline \forall f \vdash fx & \textbf{Lam'} \\ \hline \forall f \vdash fx = \top & \textbf{Rep with Eq} \\ \hline \forall f \vdash (\lambda x.fx) = \lambda x.\top & \textbf{Con} \\ \hline \forall f \vdash f = \lambda x.\top & \textbf{Lam} \end{array}$$

Exercise 5.9.4 Derive $f = \lambda x.\top \vdash \forall f$ with I∀.

Example 5.9.5 (Extensionality) We have $\forall x.\, fx{=}gx \vdash f{=}g$. The respective rule is known as extensionality. Here is a proof tree:

$$\begin{array}{cl} \hline \forall x.\, fx{=}gx \vdash fx{=}gx & \textbf{Triv, E}\forall \\ \hline \forall x.\, fx{=}gx \vdash (\lambda x.fx){=}\lambda x.gx & \textbf{Con} \\ \hline \forall x.\, fx{=}gx \vdash f{=}g & \textbf{Lam} \end{array}$$

5.10 Simplifying Lam and Sub

The basic proof system can be simplified by replacing Lam and Sub by simpler rules. This simplification makes it easier to prove soundness. Lam and Sub then appear as derived rules. Here are the simpler rules:

$$\beta \quad \vdash (\lambda x.s)t = s^x_t \qquad\qquad \eta \quad \vdash (\lambda x.fx) = f \qquad\qquad \textbf{Eq} \quad \vdash x \equiv x{=}\top$$

First we observe that we can prove Ref with Rew and β:

$$\dfrac{\dfrac{}{\vdash (\lambda x.s)x = s}\,\beta \qquad \dfrac{}{\vdash (\lambda x.s)x = s}\,\beta}{\vdash s = s}\,\text{Rew}$$

So we may also drop Ref. Lam can be derived with Ref, β, η, Con, Sym, and Trans. To see that Sub is derivable, we first show that the steps $\dfrac{\vdash s = t}{\vdash s^x_u = t^x_u}$ are derivable.

$$\dfrac{\dfrac{}{(\lambda x.t)u = t^x_u}\,\beta \qquad \dfrac{\dfrac{}{(\lambda x.s)u = s^x_u}\,\beta \qquad \dfrac{s = t}{(\lambda x.s)u = (\lambda x.t)u}\,\text{Con}}{s^x_u = (\lambda x.t)u}\,\text{Rew}}{s^x_u = t^x_u}\,\text{Rew}$$

Next we show that the steps $\dfrac{\vdash s}{\vdash s^x_t}$ are derivable.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash s}{\vdash s = \top}\,\text{Rew with Eq}}{\vdash (\lambda x.s)t = (\lambda x.\top)t}\,\text{Con}}{\vdash s^x_t = \top}\,\text{Lam}}{\vdash s^x_t}\,\text{Rew with Eq}$$

Now we obtain the derivability of Sub with Ded$^-$ and Ded.

5.11 Remarks

In the first two chapters of their textbook [23], Huth and Ryan give an elementary introduction to natural deduction. They use a popular graphical notation for natural deduction proofs and give many examples. To know more about proof systems, you may consult the textbook by Troelstra and Schwichtenberg [38]. An excellent source of historical information on logic is the Stanford Encyclopedia of Philosophy [41], which is available online. The following text is taken from the entry *The Development of Proof Theory* (with minor changes).

Before the work of Frege in 1879 [13], no one seems to have maintained that there could be a complete set of principles of proof, in the sense expressed by Frege when he wrote that in his symbolic language, "all that is necessary for a correct inference is expressed in full, but what is not necessary is generally not indicated; nothing is left to guesswork." Even after Frege, logicians such as Peano

$$\beta \quad \frac{}{(\lambda x.s)t = s^x_t} \qquad \eta \quad \frac{}{(\lambda x.fx) = f} \qquad \textbf{Rew} \quad \frac{s = t \qquad C[s]}{C[t]}$$

Figure 5.6: The proof system **L**

kept formalizing the language of mathematical arguments, but without any explicit list of rules of proof. Frege's step ahead was decisive for the development of logic and foundational study.

Russell took up Frege's logic, but used the notation of Peano, and thus formulated an axiomatic approach to logic. The idea was that the axioms express basic logical truths, and other logical truths are derived from these through modus ponens and universal generalization (I∀), the two principles Frege had identified. Mathematics was to be reduced to logic, so that its proofs would become presented in the same axiomatic pattern.

In his thesis Untersuchungen über das logische Schliessen (Investigations into Logical Inference, under the supervision of Bernays, accepted in June 1933 and published in two parts in 1934-35), Gentzen states that he set as his task the analysis of mathematical proofs as they occur in practice. Hilbert's lecture in Hamburg in 1930 is one obvious source of inspiration for this venture. The first observation is that actual proofs are not based on axioms expressed in a logical language, as in Hilbert's axiomatic proof theory. The most typical feature is instead that theorems make their claims under some assumptions.

5.12 Bonus: Equational Deduction

The **proof system L** consists of the proof steps described by the rules in Figure 5.6. The premises and conclusions of the steps of **L** are formulas. Note that every formula in **L**$[\emptyset]$ is an equation.

Proposition 5.12.1 (Soundness)

1. **L** is sound for $\{\, s{=}t \mid s \sim_\lambda t \,\}$.
2. **L** is sound for the set of valid formulas.

Proof We have $\to_\beta \cup \to_\eta \subseteq \sim_\lambda$ by the definition of lambda equivalence in § 3.6. Hence the proof steps described by β and η are sound for lambda equivalent equations. The soundness of the steps described by Rew follows from the compatibility, symmetry, and transitivity of lambda equivalence (Proposition 3.6.7).

We omit the soundness proof for valid formulas. ∎

$$\textbf{Ref}\ \frac{}{s = s} \qquad \textbf{Sym}\ \frac{s = t}{t = s} \qquad \textbf{Trans}\ \frac{s = t \qquad t = u}{s = u} \qquad \textbf{Con}\ \frac{s = t}{C[s] = C[t]}$$

$$\textbf{Sub}\ \frac{s = t}{\theta s = \theta t} \qquad \alpha\ \frac{}{(\lambda x.s) = \lambda y.s^x_y}\ \ y \notin \mathcal{N}(\lambda x.s) \qquad \textbf{Lam}\ \frac{s}{t}\ \ s \sim_\lambda t$$

$$\textbf{Rep}\ \frac{s = t \qquad C[\theta s]}{C[\theta t]} \qquad\qquad \textbf{Rep}\ \frac{s = t \qquad C[\theta t]}{C[\theta s]}$$

$$\textbf{App}\ \frac{f = \lambda x.u}{fs = u^x_s} \qquad\qquad \textbf{App}\ \frac{f = \lambda xy.u}{fst = u^{x\,y}_{s\,t}}$$

$$\textbf{Abs}\ \frac{fx = s}{f = \lambda x.s} \qquad\qquad \textbf{Abs}\ \frac{fxy = s}{f = \lambda xy.s}$$

$$\textbf{FE}\ \frac{sx = tx}{s = t}\ \ x \notin \mathcal{N}(s = t)$$

Figure 5.7: Rules derivable in **L**

Figure 5.7 shows some rules that are derivable in **L**. We give some of the derivations and leave the others as exercises.

Derivation of Ref

$$\cfrac{\cfrac{}{(\lambda x.s)x = s}\,\beta \qquad \cfrac{}{(\lambda x.s)x = s}\,\beta}{s = s}\,\textbf{Rew}$$

Derivation of Sym

$$\cfrac{s = t \qquad \cfrac{}{s = s}\,\textbf{Ref}}{t = s}\,\textbf{Rew}$$

Exercise 5.12.2 Derive Trans in **L**.

Derivation of Con

$$\dfrac{s=t \qquad \dfrac{}{C[s]=C[s]}\;\text{Ref}}{C[s]=C[t]}\;\text{Rew}$$

Derivation of Sub

We show how Sub can be derived if only one variable is replaced.

$$\dfrac{\dfrac{}{(\lambda x.t)u = t^x_u}\;\beta \qquad \dfrac{\dfrac{}{(\lambda x.s)u = s^x_u}\;\beta \qquad \dfrac{s=t}{(\lambda x.s)u=(\lambda x.t)u}\;\text{Con}}{s^x_u = (\lambda x.t)u}\;\text{Rew}}{s^x_u = t^x_u}\;\text{Rew}$$

The derivation can be generalized to the case where several variables are replaced.

Derivation of α

$$\dfrac{\dfrac{}{(\lambda x.s)y = s^x_y}\;\beta \qquad \dfrac{\dfrac{\dfrac{}{(\lambda y.fy)=f}\;\eta}{f=\lambda y.fy}\;\text{Sym}}{(\lambda x.s)=\lambda y.(\lambda x.s)y}\;\text{Sub},\ y\notin\mathcal{N}(\lambda x.s)}{(\lambda x.s)=\lambda y.s^x_y}\;\text{Rew}$$

Derivation of Lam

Proposition 5.12.3 $\mathbf{L}[\emptyset] = \{\, s{=}t \mid s \sim_\lambda t \,\}$.

Proof The direction $\subseteq$ follows from the fact that **L** is sound for lambda equivalent equations (Proposition 5.12.1). To see the other direction, we first observe that $s = t$ is derivable in L if $s \rightarrow_\lambda t$ (rules β, η, α, Con). Now the rest follows from the definition of $\sim_\lambda$ with Ref, Sym, and Trans. ■

The derivability of Lam now follows with Rew.

Exercise 5.12.4 Derive Rep, App, Abs, and FE in **L**.

Equational Deduction with Sequents

The **proof system ED** consists of the proof steps described by the rules in Figure 5.8. The premises and conclusions of the steps of **ED** are sequents.

Proposition 5.12.5 (Soundness) **ED** is sound for valid sequents.

$$\beta\ \frac{}{\vdash (\lambda x.s)t = s^x_t} \qquad \eta\ \frac{}{\vdash (\lambda x.fx) = f}$$

$$\textbf{Rew}\ \frac{A \vdash s{=}t \qquad A \vdash C[s]}{A \vdash C[t]}\ C \text{ admissible for } A$$

$$\textbf{Triv}\ \frac{}{A,\, s \vdash s} \qquad \textbf{Weak}\ \frac{A \vdash s}{B \vdash s}\ A \subseteq B$$

$$\textbf{Cut}\ \frac{A \vdash s \qquad A, s \vdash t}{A \vdash t} \qquad \textbf{Sub}\ \frac{A \vdash s}{\theta A \vdash \theta s}$$

Figure 5.8: The proof system **ED**

Lemma 5.12.6 A proof step $(\{s_1, \dots, s_n\}, s)$ is derivable in **L** if and only if the proof step $(\{\dot\vdash s_1, \dots, \dot\vdash s_n\}, \dot\vdash s)$ is derivable in **ED**.

Proof The direction from **L** to **ED** is easy. The other direction follows from the fact that for every step $(P, A \dot\vdash s) \in$ **ED** and every premise $(B \dot\vdash t) \in P$ we have $B \subseteq A$. Hence a proof tree for a sequent $\dot\vdash s$ will only involve sequents with an empty assumption set and hence will not involve the rules Triv and Weak. ■

Proposition 5.12.7 $\dot\vdash s{=}t$ derivable in **ED** if and only if $s \sim_\lambda t$.

Exercise 5.12.8 Generalize the derived rules in Figure 5.7 to **ED**. For instance, Sym generalizes to $\dfrac{A \vdash s{=}t}{A \vdash t{=}s}$. Of particular interest are the rules Con, Sub, Rep, Abs and FE, which need side conditions.

6 Tableau Proofs

In Chapter 4 we learned about logical interpretations. Logical interpretations give us a notion of validity. In Chapter 5 we learned about proof systems. The proof system **B** gives us a notion of deducibility. Using the basic and derived proof rules presented in Chapter 5 to deduce a sequent often requires a great deal of creativity. In this chapter we will make the process of proving a sequent more mechanical.

6.1 An Example: Peirce's Law

Before going on, let us consider an interesting formula called **Peirce's Law**:

$$((p \to q) \to p) \to p$$

Here we assume $p, q : B$ are (distinct) names. We can argue that Peirce's Law is valid by considering logical interpretations.

Let $\mathcal{I}$ be any logical interpretation. The interpretation $\mathcal{I}(\to)$ of implication must be the function in $\mathbb{B} \to \mathbb{B} \to \mathbb{B}$ such that

$$\mathcal{I}(\to)\, a\, b \text{ equals } \begin{cases} 0 & \text{if } a = 1 \text{ and } b = 0 \\ 1 & \text{if } a = 0 \text{ or } b = 1 \end{cases}$$

for $a, b \in \mathbb{B}$. In other words, $\mathcal{I} \vDash s \to t$ iff (if and only if) either $\mathcal{I} \vDash \neg s$ or $\mathcal{I} \vDash t$. Likewise, $\mathcal{I} \vDash \neg(s \to t)$ iff both $\mathcal{I} \vDash s$ and $\mathcal{I} \vDash \neg t$.

Assume Peirce's Law were not valid. Then there must be some logical interpretation $\mathcal{I}$ such that

$$\mathcal{I} \vDash \neg(((p \to q) \to p) \to p).$$

Consequently,

$$\mathcal{I} \vDash \neg p$$

and

$$\mathcal{I} \vDash ((p \to q) \to p.$$

Hence either $\mathcal{I} \vDash p$ or $\mathcal{I} \vDash \neg(p \to q)$. We cannot have $\mathcal{I} \vDash p$ since this contradicts $\mathcal{I} \vDash \neg p$. So we must have $\mathcal{I} \vDash \neg(p \to q)$. This means $\mathcal{I} \vDash \neg q$ and $\mathcal{I} \vDash p$, which again contradicts $\mathcal{I} \vDash \neg p$. Therefore, there can be no such logical interpretation

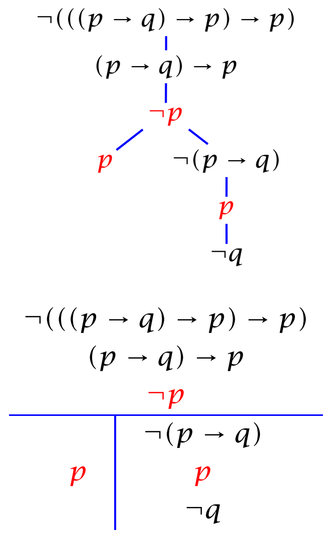


Figure 6.1: Tableau for Peirce's Law (in two formats)

1. In other words, Peirce's Law is valid. We summarize this argument in the form of a tree (in two formats) in Figure 6.1. Let us call this tree a **tableau** for Peirce's law. (We will soon define the general notion of a tableau.) The root of the tree contains the negation of Peirce's Law. Each child node represents a consequence of its ancestors. We represent the case split by splitting the main branch into two branches.

Now that we have an idea of why Peirce's Law is valid, let us attempt to prove it using the rules from Chapter 5. Our goal is to derive

$$\vdash ((p \to q) \to p) \to p.$$

Applying **Ded**, we can reduce this to proving

$$((p \to q) \to p) \vdash p.$$

Using **MP** and **Triv** we can reduce this to proving

$$((p \to q) \to p) \vdash p \to q.$$

Applying **Ded**, this reduces to deriving

$$((p \to q) \to p), p \vdash q.$$

So far we have constructed the following partial proof tree:

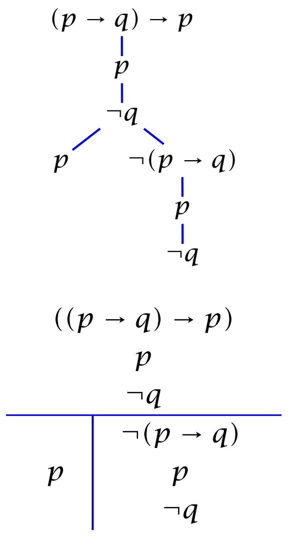


Figure 6.2: Tableau for sequent (6.1) in two formats

$$\dfrac{\dfrac{\dfrac{}{(p \to q) \to p \vdash (p \to q) \to p}\,\textbf{Triv} \qquad \dfrac{(p \to q) \to p, p \vdash q}{(p \to q) \to p \vdash p \to q}\,\textbf{Ded}}{(p \to q) \to p \vdash p}\,\textbf{MP}}{\vdash ((p \to q) \to p) \to p}\,\textbf{Ded}$$

It remains to derive the sequent

$$((p \to q) \to p), p \dot{\vdash} q. \tag{6.1}$$

It is not clear what we can do to make progress towards deriving (6.1). Instead of applying more rules, let us consider why the sequent (6.1) might be valid. We proceed exactly as we did for Peirce's Law. Assume we have an intepretation $\mathcal{I}$ witnessing that (6.1) is *not* valid. That is, $\mathcal{I} \vDash (p \to q) \to p$, $\mathcal{I} \vDash p$ and $\mathcal{I} \vDash \neg q$. Since $\mathcal{I} \vDash (p \to q) \to p$, we must have either $\mathcal{I} \vDash p$ or $\mathcal{I} \vDash \neg(p \to q)$. In the second case we must have $\mathcal{I} \vDash p$ and $\mathcal{I} \vDash \neg q$. Figure 6.2 represents these steps as a tableau. We did not reach any contradiction. In fact, if we take $\mathcal{I}$ to be a logical interpretation where $\mathcal{I}p = 1$ and $\mathcal{I}q = 0$, then all of the formulas in the tree are satisfied by $\mathcal{I}$. In this case we have shown that the sequent

$$((p \to q) \to p), p \dot{\vdash} q$$

is *not* valid. As a consequence of soundness, we know there is *no proof tree* ending with $((p \to q) \to p), p \dot{\vdash} q$.

What happened? We were easily able to argue (using interpretations) that Peirce's Law was valid, but we got lost when trying to prove it. We were also

$$\textbf{Closed}\ \frac{}{A, s, \neg s \vdash \bot} \qquad \textbf{Imp}\ \frac{A, t \vdash \bot \qquad A, \neg s \vdash \bot}{A \vdash \bot}\ s \to t \in A,\ t \notin A,\ \neg s \notin A$$

$$\textbf{NegImp}\ \frac{A, s, \neg t \vdash \bot}{A \vdash \bot}\ \neg(s \to t) \in A,\ \{s, \neg t\} \not\subseteq A$$

Figure 6.3: Some derivable refutation rules

easily able to show we were “lost” by arguing (again using interpretations) that a subgoal was not valid. In the latter case, the argument gave us an interpretation $\mathcal{I}$ showing the subgoal is not valid.

Can the argument for validity of Peirce’s Law (as depicted in Figure 6.1) serve as a guide to proving Peirce’s Law? The answer is *yes*. In fact, if we have the derived rules in Figure 6.3, then Figure 6.1 directly corresponds to the proof tree

$$\cfrac{\cfrac{\cfrac{\cfrac{}{A, \neg p, p \vdash \bot}\,\textbf{Closed} \qquad \cfrac{\cfrac{}{A, \neg p, \neg(p \to q), p, \neg q \vdash \bot}\,\textbf{Closed}}{A, \neg p, \neg(p \to q) \vdash \bot}\,\textbf{NegImp}}{\neg(((p \to q) \to p) \to p), (p \to q) \to p, \neg p \vdash \bot}\,\textbf{Imp}}{\neg(((p \to q) \to p) \to p) \vdash \bot}\,\textbf{NegImp}}{\vdash ((p \to q) \to p) \to p}\,\textbf{Contr}$$

where $A = \{\neg(((p \to q) \to p) \to p), (p \to q) \to p\}$.

Exercise 6.1.1 Prove the following rules are derivable. You may use any of the basic or derived rules from Chapter 5.

a)

$$\textbf{Contra}^{-}\ \frac{A \vdash s}{A, \neg s \vdash \bot}$$

b)

$$\mathbf{I}\neg\ \frac{A, s \vdash \bot}{A \vdash \neg s}$$

c)

$$\mathbf{I}\neg^{-}\ \frac{A \vdash \neg s}{A, s \vdash \bot}$$

d) Prove the rules in Figure 6.3 are derivable.

6.2 Refutations

A **refutation step** is a proof step of the form

$$\frac{A_1 \vdash \bot \qquad \ldots \qquad A_n \vdash \bot}{A \vdash \bot}$$

where $n \geq 0$ and $A \subset A_i$ (i.e., $A \subseteq A_i$ and $A \neq A_i$) for each $i \in \{1, \ldots, n\}$. A **refutation rule** is a schema of refutation steps. We will introduce a number of refutation rules throughout this chapter by giving a diagram

$$\frac{A_1 \vdash \bot \qquad \ldots \qquad A_n \vdash \bot}{A \vdash \bot}\ \phi$$

We restrict instances of refutation rules to be refutation steps. We have already seen three refutation rules in Figure 6.3. Note that

$$\frac{p \to q, p, q \vdash \bot \qquad p \to q, p, \neg p \vdash \bot}{p \to q, p \vdash \bot}$$

is an instance of the **Imp** rule. On the other hand,

$$\frac{p \to q, \neg p, q \vdash \bot \qquad p \to q, \neg p \vdash \bot}{p \to q, \neg p \vdash \bot}$$

is not an instance of the **Imp** rule since it is not a refutation step.

Exercise 6.2.1 Which of the following are instances of the refutation rule **NegImp**.

a)

$$\frac{\neg p, p, \neg q \vdash \bot}{\neg p, \neg(p \to q) \vdash \bot}$$

b)

$$\frac{\neg p, \neg(p \to q), p, \neg q \vdash \bot}{\neg p, \neg(p \to q) \vdash \bot}$$

c)

$$\frac{\neg(p \to q), p, \neg q \vdash \bot}{\neg(p \to q), p \vdash \bot}$$

d)

$$\frac{\neg(p \to q), p, \neg q \vdash \bot}{\neg(p \to q), p, \neg q \vdash \bot}$$

A **refutation of A** is a proof tree for $A \dot{\vdash} \bot$ where each step is a refutation step. Note that every sequent in a refutation must be of the form $A \dot{\vdash} \bot$.

Each of the refutation rules we give in this chapter will be derivable in the basic proof system **B**. Consequently, we will always be assured that if we have a refutation of A using the refutation rules presented in this chapter, then we also have $A \vdash \bot$.

Note that for any sequent $A \dot{\vdash} s$, we can reduce the problem of proving $A \dot{\vdash} s$ to the problem of finding a refutation of $A, \neg s$ by using the **Contra** rule.

Also, if we are trying to prove $A \dot{\vdash} \bot$ and apply a refutation rule, then each new subgoal will be $A_i \dot{\vdash} \bot$ with $A \subset A_i$. Note that any interpretation satisfying A_i will necessarily satisfy A. We can never end up with a subgoal that is not valid unless $A \dot{\vdash} \bot$ was not valid to begin with. This is in contrast to the **ND** system. We say in the case of Peirce's Law that applying **ND** rules such as **MP** can yield subgoals that are not valid even if the conclusion is valid. When we search with **ND** rules, we may need to backtrack. When we search with refutation rules, we never need to backtrack.

6.3 Tableaux

Just as the refutation of Peirce's Law can be viewed as the tree in Figure 6.1, any refutation can be viewed as such a tree. Suppose we have a refutation rule of the form

$$\frac{A, P_1 \vdash \bot \qquad \ldots \qquad A, P_n \vdash \bot}{A \vdash \bot}\; P \subseteq A, \;\ldots$$

Note that each refutation rule (read from bottom to top) allows us to expand the set of assumptions. Suppose $P = \{s_1, \ldots, s_m\}$ and $P_i = \{t^i_1, \ldots, t^i_{k_i}\}$ for each $i \in \{1, \ldots, n\}$. According to the refutation rule, if we have $s_1, \ldots, s_m$ in our set of assumptions, then we can split the goal into n cases. In the case i^{th} case (where $i \in \{1, \ldots, n\}$), we can add $t^i_1, \ldots, t^i_{k_i}$ to our assumptions. As in Figure 6.1, we include assumptions by extending the relevant branch of the tree and we distinguish cases by splitting the relevant branch into several branches. Given this convention, a tableau view of such a rule is shown in Figure 6.4. Figure 6.4 should be read operationally: if a branch contains all the formulas above the line (and any side condition holds with respect to the branch), then we split the branch into n branches where we add $\{t^i_1, \ldots, t^i_{k_i}\}$ to the i^{th} branch. We can also depict this operation as shown in Figure 6.5: if a branch on the current tableau contains all the formulas on the solid line, then applying the refutation rule allows us to extend the tableau by adding the dotted portions (so long as any side condition holds with respect to the branch).

$$\frac{s_1,\ldots,s_m}{t^1_1,\ldots,t^1_{k_1} \mid \cdots \mid t^n_1,\ldots,t^n_{k_n}}$$

Figure 6.4: Tableau view of a refutation rule

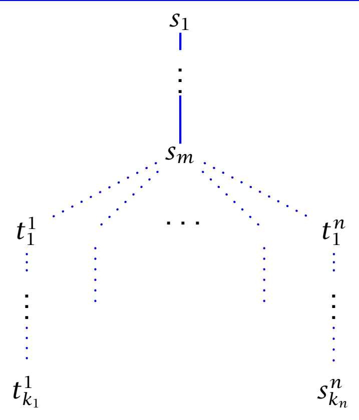


Figure 6.5: Refutation rule as an operation on a tableau

How do tableaux correspond to refutations? For each branch of a tableau there is a set A of formulas that occur on the branch. If the tableau has n branches with corresponding sets of formulas $A^1,\ldots,A^n$, then we have n sequents to prove, namely

$$A^1 \dot{\vdash} \bot \qquad\qquad \cdots \qquad\qquad A^n \dot{\vdash} \bot$$

For some of the branches, we may be able to apply a refutation rule with no premisses. One example of such a rule is the **Closed** rule in Figure 6.3. This rule is given along with four other refutation rules with no premisses in Figure 6.6. If one of these rules applies, then we say the branch is **closed**. In other words, a branch with a set of formulas A is closed if one of the following holds:

1. $\{s,\neg s\} \subseteq A$,
2. $\bot \in A$,
3. $\neg\top \in A$,
4. $\neg(s = s) \in A$, or
5. $(s = t), \neg(t = s) \in A$.

Otherwise, we say the other branches are **open**. At any stage of the proof, our goal is to prove the sequent $A \dot{\vdash} \bot$ for each open branch with formulas A.

We think of the formulas on a branch conjunctively. We think of the collection

$$\textbf{Closed}\ \frac{}{A, \bot \vdash \bot} \qquad \textbf{Closed}\ \frac{}{A, \neg\top \vdash \bot} \qquad \textbf{Closed}\ \frac{}{A, s, \neg s \vdash \bot}$$

$$\textbf{Closed}\neg = \ \frac{}{A, \neg(s = s) \vdash \bot} \qquad \textbf{ClosedSym}\ \frac{}{A, (s = t), \neg(t = s) \vdash \bot}$$

Figure 6.6: Derivable refutation rules with no premisses

of branches disjunctively. For a logical interpretation $\mathcal{I}$, we say $\mathcal{I}$ **satisfies a branch** if it satisfies every formula on the branch. We say a branch is **satisfiable** if there is some logical interpretation $\mathcal{I}$ that satisfies the branch.

Suppose a branch with a set of formulas A is closed. Then $A \dot{\vdash} \bot$ is derivable (using one of the derived rules in Figure 6.6. By soundness, A is unsatisfiable. In other words, the branch is unsatisfiable. If every branch of a tableau is closed, then every branch is unsatisfiable.

Suppose C is a set of formulas such that every formula $s \in C$ is on every branch in the tableau. In this case, if some branch is satisfiable, then C is satisfiable.

6.4 Refutation Rules for Propositional Logic

We have already seen refutation rules for $\rightarrow$ in Figure 6.3. Also, there are two refutation rules in Figure 6.6 for $\top$ and $\bot$. In this section we will derivable refutation rules corresponding to the logical constants $\neg$, $\wedge$, $\vee$ and $=_B$. We also show the tableau views of the rules. These rules will be enough to prove any valid formula of propositional logic. That is, we will prove our first completeness results.

Refutation rules for propositional constants (including the two rules for $\rightarrow$) are given in Figure 6.7. In each of the rules we implicitly assume that A is a proper subset of the left hand side of the sequent in each premiss. The corresponding tableau views are shown in Figure 6.8. The rules are depicted as operations on tableaux in Figure 6.9. We say a set of formulas A has a **propositional refutation** if there is a refutation of A using only the rules in Figures 6.6 and 6.7.

We can now state the completeness result we want to prove.

Theorem 6.4.1 (Propositional Completeness)

1. If A is an unsatisfiable finite set of propositional formulas, then there is a propositional refutation of A.

$$\textbf{DNeg}\ \frac{A, s \vdash \bot}{A \vdash \bot}\ \neg\neg s \in A \qquad \textbf{DeMorgan}\ \frac{A, \neg s \wedge \neg t \vdash \bot}{A \vdash \bot}\ \neg(s \vee t) \in A$$

$$\textbf{DeMorgan}\ \frac{A, \neg s \vee \neg t \vdash \bot}{A \vdash \bot}\ \neg(s \wedge t) \in A \qquad \textbf{And}\ \frac{A, s, t \vdash \bot}{A \vdash \bot}\ s \wedge t \in A$$

$$\textbf{Or}\ \frac{A, s \vdash \bot \qquad A, t \vdash \bot}{A \vdash \bot}\ s \vee t \in A \qquad \textbf{Imp}\ \frac{A, t \vdash \bot \qquad A, \neg s \vdash \bot}{A \vdash \bot}\ s \to t \in A$$

$$\textbf{NegImp}\ \frac{A, s, \neg t \vdash \bot}{A \vdash \bot}\ \neg(s \to t) \in A$$

$$\textbf{Boolean}{=}\ \frac{A, s, t \vdash \bot \qquad A, \neg s, \neg t \vdash \bot}{A \vdash \bot}\ s \equiv t \in A$$

$$\textbf{Boolean}{\neq}\ \frac{A, s, \neg t \vdash \bot \qquad A, t, \neg s \vdash \bot}{A \vdash \bot}\ \neg(s \equiv t) \in A$$

Figure 6.7: Derivable refutation rules for propositional constants

$$\textbf{DNeg}\frac{\neg\neg s}{s} \qquad \textbf{DeMorgan}\frac{\neg(s \vee t)}{\neg s \wedge \neg t} \qquad \textbf{DeMorgan}\frac{\neg(s \wedge t)}{\neg s \vee \neg t}$$

$$\textbf{And}\frac{s \wedge t}{s, t} \qquad \textbf{Or}\frac{s \vee t}{s \mid t} \qquad \textbf{Imp}\frac{s \to t}{t \mid \neg s} \qquad \textbf{NegImp}\frac{\neg(s \to t)}{s, \neg t}$$

$$\textbf{Boolean}{=}\frac{s \equiv t}{s, t \mid \neg s, \neg t} \qquad \textbf{Boolean}{\neq}\frac{\neg(s \equiv t)}{s, \neg t \mid t, \neg s}$$

Figure 6.8: Tableau views of refutation rules for propositional constants

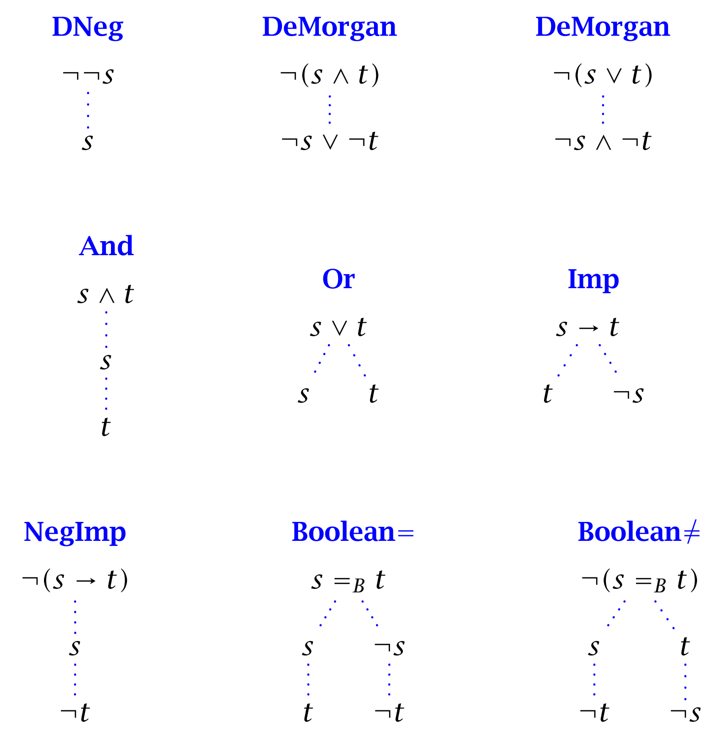


Figure 6.9: Propositional refutation rules as operations on tableaux

2. If A is a finite set of propositional formulas, s is a propositional formula and $A \vDash s$, then $A \vdash s$.

Since the **Contra** rule is derived and all the rules in Figures 6.6 and 6.7 are derived, the second part of Theorem 6.4.1 follows from the first part by considering the set $A, \neg s$. That is, we only need to prove the first part of Theorem 6.4.1. We will eventually prove that either A has a refutation or A is satisfiable. We first prove that if we reach a proof state with subgoal $A \vdash \bot$ but none of the refutation rules apply, then A must be satisfiable.

What does it mean to say none of the refutation rules applies? We mean that no refutation step which is an instance of one of the refutation rules has conclusion $A \vdash \bot$.

Suppose we are trying to prove $\vdash (q \to p) \to p$. This reduces to refuting $\{\neg((q \to p) \to p)\}$. Applying the refutation rules we can construct the following tableau with two branches:

$$\begin{array}{c}\neg((q \to p) \to p)\\ q \to p\\ \neg p\\ \hline \begin{array}{c|c} p & \neg q\end{array}\end{array}$$

The left branch is closed since both p and $\neg p$ occur on the branch. The right branch is open and corresponds to the subgoal $A \vdash \bot$ where

$$A = \{\neg((q \to p) \to p), (q \to p), \neg p, \neg q\}.$$

Can we apply any of the refutation rules in Figures 6.6 and 6.7 to reduce $A \vdash \bot$ to new subgoals? In other words, does any instance of the refutation rules in Figures 6.6 and 6.7 have conclusion $A \vdash \bot$? Inspecting each of the rules, we find that no instance of the rules has conclusion $A \vdash \bot$. The only rules that might apply are **NegImp** since $\neg((q \to p) \to p) \in A$ and **Imp** since $(q \to p) \in A$. However, **NegImp** does not apply since $\{(q \to p), (\neg p)\} \subseteq A$ and **Imp** does not apply since $\neg q \in A$.

Clearly we cannot use the refutation rules in Figures 6.6 and 6.7 to refute

$$A = \{\neg((q \to p) \to p), (q \to p), \neg p, \neg q\}.$$

Is A satisfiable? In other words, is there a logical interpretation $\mathcal{I}$ such that $\mathcal{I} \models A$? Yes, there is such a logical interpretation and we define one now. We will also make use of this interpretation (appropriately modified) to prove that whenever no instance of the refutation rules in Figures 6.6 and 6.7 has conclusion $A \vdash \bot$, then A is satisfiable.

We define a **default interpretation** $\mathcal{I}^{\mathsf{d}}$ as follows. On the sort B, we take $\mathcal{I}^{\mathsf{d}}B = \{0, 1\}$. On every other sort α, we take $\mathcal{I}^{\mathsf{d}}\alpha$ to be the singleton set $\{\emptyset\}$. On function types $\tau\sigma$, $\mathcal{I}^{\mathsf{d}}(\tau\sigma)$ is defined to be the set of functions from $\mathcal{I}^{\mathsf{d}}\tau$ to $\mathcal{I}^{\mathsf{d}}\sigma$. We can define a default element d^τ of each $\mathcal{I}^{\mathsf{d}}\tau$ as follows:

$$d^B = 0 \in \mathcal{I}^{\mathsf{d}}B,$$

$$d^\alpha = \emptyset \in \mathcal{I}^{\mathsf{d}}\alpha \text{ for other sorts } \alpha,$$

and

$$d^{\tau\sigma} = (\lambda x \in \mathcal{I}^{\mathsf{d}}\tau . d^\sigma) \in \mathcal{I}^{\mathsf{d}}(\tau\sigma).$$

Note that the use of λ above is to describe a function, not a term. We have defined $d^{\tau\sigma}$ as the function in $\mathcal{I}^{\mathsf{d}}(\tau\sigma)$ which takes each element of $\mathcal{I}^{\mathsf{d}}\tau$ to $d^\sigma \in \mathcal{I}^{\mathsf{d}}\sigma$. For each logical constant c of type σ, we define $\mathcal{I}^{\mathsf{d}}c$ to be the unique element of $\mathcal{I}^{\mathsf{d}}\sigma$ which has the property corresponding to the constant. For variables x of type σ, we define $\mathcal{I}^{\mathsf{d}}x = d^\sigma$. That is, all variables are interpreted as the default element of the given type.

One can easily see that $\mathcal{I}^{\mathsf{d}} \vDash A$ since $\mathcal{I}^{\mathsf{d}}p = 0$ and $\mathcal{I}^{\mathsf{d}}q = 0$. We can use a similar idea to prove the following general result.

Proposition 6.4.2 Let A be a finite set of propositional formulas. If no instance of the refutation rules in Figures 6.6 and 6.7 has conclusion $A \vdash \bot$, then A is satisfiable.

For any set A of propositional formulas we can construct a logical interpretation $\mathcal{I}^A$ such that

$$\mathcal{I}^A p = \begin{cases} 1 & \text{if } p \in A \\ 0 & \text{if } p \notin A \end{cases}$$

We define $\mathcal{I}^A$ as follows:

$$\mathcal{I}^A \sigma = \mathcal{I}^{\mathsf{d}} \sigma$$

for all types σ,

$$\mathcal{I}^A p = 1$$

if $p \in A$ (where p is a variable), and

$$\mathcal{I}^A a = \mathcal{I}^{\mathsf{d}} a$$

for every other name a. Clearly $\mathcal{I}^A \vDash p$ for names $p \in A$. We cannot in general conclude $\mathcal{I}^A \vDash A$ (i.e., $\mathcal{I}^A \vDash s$ for all $s \in A$). However, we will soon define a property $\nabla_{\mathbf{Prop}}$ of sets A of formulas which will guarantee $\mathcal{I}^A \vDash A$. Once we define $\nabla_{\mathbf{Prop}}$ we will be able to prove two lemmas. Proposition 6.4.2 will follow from these two lemmas.

Lemma 6.4.3 Let A be a finite set of formulas. If no instance of the refutation rules in Figures 6.6 and 6.7 has conclusion $A \vdash \bot$, then A satisfies $\nabla_{\mathbf{Prop}}$.

Lemma 6.4.4 Let A be a set of formulas satisfying $\nabla_{\mathbf{Prop}}$. For all propositional formulas s, we have the following:

1. If $s \in A$, then $\hat{\mathcal{I}^A} s = 1$.
2. If $\neg s \in A$, then $\hat{\mathcal{I}^A} s = 0$.

To see that Proposition 6.4.1 follows from the two lemmas, we argue as follows. Suppose A is a finite set of propositional formulas and no instance of the refutation rules in Figures 6.6 and 6.7 has conclusion $A \vdash \bot$. By Lemma 6.4.3 A satisfies $\nabla_{\mathbf{Prop}}$. By the first property in Lemma 6.4.4, $\mathcal{I}^A \vDash A$.

The property $\nabla_{\mathbf{Prop}}$ will be the conjunction of several properties. We will call these properties **Hintikka properties**. In order to simplify the presentation, let us first consider the fragment of propositional logic with negation and implication. Let $\textbf{Prop}_{\rightarrow}$ be the set of formulas determined by the grammar

$$s ::= p \mid \neg s \mid s \rightarrow s$$
$$p : B \quad \text{is a variable}$$

We define the following **Hintikka properties** of a set A of formulas:

∇_c For all formulas s, $s \notin A$ or $\neg s \notin A$.

$\nabla_{\neg\neg}$ If $\neg\neg s \in A$, then $s \in A$.

$\nabla_{\rightarrow}$ If $s \rightarrow t \in A$, then $\neg s \in A$ or $t \in A$.

$\nabla_{\neg\rightarrow}$ If $\neg(s \rightarrow t) \in A$, then $s \in A$ and $\neg t \in A$.

Let $\nabla_{\textbf{Prop}_{\rightarrow}}$ be the conjunction of these four Hintikka properties.

We now prove the following simplified versions of Lemmas 6.4.3 and 6.4.4.

Lemma 6.4.5 (Simplified version of Lemma 6.4.3) Let A be a finite set of formulas. If no instance of the refutation rules in Figures 6.6 and 6.7 has conclusion $A \vdash \bot$, then A satisfies $\nabla_{\textbf{Prop}_{\rightarrow}}$.

Lemma 6.4.6 (Simplified version of Lemma 6.4.4) Let A be a set of formulas satisfying $\nabla_{\textbf{Prop}_{\rightarrow}}$. For all $s \in \textbf{Prop}_{\rightarrow}$, we have the following:

1. If $s \in A$, then $\hat{\mathcal{I}}^A s = 1$.
2. If $\neg s \in A$, then $\hat{\mathcal{I}}^A s = 0$.

Proof (Proof of Lemma 6.4.5) We verify each of the four properties. The careful reader will note that we can prove the result even if all we know is that no instance of the rules **Closed**, **DNeg**, **Imp** or **NegImp** has conclusion $A \vdash \bot$.

∇_c Assume ∇_c does not hold. Then $\{s, \neg s\} \subseteq A$ for some formula s. In this case an instance of the rule **Closed** has conclusion $A \vdash \bot$, contradicting our assumption.

$\nabla_{\neg\neg}$ Assume $\nabla_{\neg\neg}$ does not hold. We must have $\neg\neg s \in A$ and $s \notin A$. In this case an instance of **DNeg** has conclusion $A \vdash \bot$ (and premiss $A, s \vdash \bot$).

$\nabla_{\rightarrow}$ Assume $\nabla_{\rightarrow}$ does not hold. We must have $s \rightarrow t \in A$, $\neg s \notin A$ and $t \notin A$. In this case an instance of **Imp** has conclusion $A \vdash \bot$ (and premisses $A, \neg s \vdash \bot$ and $A, t \vdash \bot$).

$\nabla_{\neg\rightarrow}$ Assume $\nabla_{\neg\rightarrow}$ does not hold. We must have $\neg(s \rightarrow t) \in A$ and $\{s, \neg t\} \not\subseteq A$. In this case an instance of **NegImp** has conclusion $A \vdash \bot$ (and premiss $A, s, \neg t \vdash \bot$). ■

Proof (Proof of Lemma 6.4.6) The proof is by induction on the structure of $s \in \textbf{Prop}_{\rightarrow}$. We need to prove for every $s \in \textbf{Prop}_{\rightarrow}$ the two properties hold:

(1^s) If $s \in A$, then $\mathcal{I}^A s = 1$.

(2^s) If $\neg s \in A$, then $\mathcal{I}^A s = 0$.

There are three kinds of formulas in $\mathbf{Prop}_{\to}$: variables p, negations $\neg s$ and implications $s \to t$. We consider each case.

For the base case of the induction we consider a variable $p \in \mathbf{Prop}_{\to}$.

(1^p): Assume $p \in A$. We must prove $\mathcal{I}^A p = 1$. This is trivial since $\mathcal{I}^A p$ was defined to be 1.

(2^p): Assume $\neg p \in A$. We must prove $\mathcal{I}^A p = 0$. By ∇_c we know $p \notin A$. By definition $\mathcal{I}^A p = 0$.

Next we consider $\neg s \in \mathbf{Prop}_{\to}$. Our inductive hypothesis is that both (1^s) and (2^s) hold.

($1^{\neg s}$): Assume $\neg s \in A$. We must prove $\hat{\mathcal{I}}^A(\neg s) = 1$. By ($2^s$) we know $\hat{\mathcal{I}}^A s = 0$ and so $\hat{\mathcal{I}}^A(\neg s) = 1$.

($2^{\neg s}$): Assume $\neg\neg s \in A$. We must prove $\hat{\mathcal{I}}^A(\neg s) = 0$. By $\nabla_{\neg\neg}$ we have $s \in A$. By (1^s) we know $\hat{\mathcal{I}}^A s = 1$ and so $\hat{\mathcal{I}}^A(\neg s) = 0$.

Finally we consider $s \to t \in \mathbf{Prop}_{\to}$. Our inductive hypothesis is that (1^s), (2^s), (1^t) and (2^t) hold.

($1^{s\to t}$): Assume $s \to t \in A$. We must prove $\hat{\mathcal{I}}^A(s \to t) = 1$. By $\nabla_{\to}$ either $\neg s \in A$ or $t \in A$. If $\neg s \in A$ then $\hat{\mathcal{I}}^A s = 0$ by (2^s) and so $\hat{\mathcal{I}}^A(s \to t) = 1$. If $t \in A$ then $\hat{\mathcal{I}}^A t = 1$ by (1^t) and so $\hat{\mathcal{I}}^A(s \to t) = 1$. In either case we have the desired result.

($2^{s\to t}$) Assume $\neg(s \to t) \in A$. We must prove $\hat{\mathcal{I}}^A(s \to t) = 0$. By $\nabla_{\neg\to}$ both $s \in A$ and $\neg t \in A$. By (1^s) and (2^t) we have $\hat{\mathcal{I}}^A s = 1$ and $\hat{\mathcal{I}}^A t = 0$. Hence $\hat{\mathcal{I}}^A(s \to t) = 0$. ■

We now define $\nabla_{\mathbf{Prop}}$ as the conjunction of the following 12 properties of a set A of formulas:

∇_c For all formulas s, $s \notin A$ or $\neg s \notin A$.

$\nabla_\bot$ $\bot \notin A$.

$\nabla_{\neg\top}$ $\neg\top \notin A$.

$\nabla_{\neg\neg}$ If $\neg\neg s \in A$, then $s \in A$.

$\nabla_{\to}$ If $s \to t \in A$, then $\neg s \in A$ or $t \in A$.

$\nabla_{\neg\to}$ If $\neg(s \to t) \in A$, then $s \in A$ and $\neg t \in A$.

$\nabla_\vee$ If $s \vee t \in A$, then $s \in A$ or $t \in A$.

$\nabla_{\neg\vee}$ If $\neg(s \vee t) \in A$, then $\neg s \in A$ and $\neg t \in A$.

$\nabla_\wedge$ If $s \wedge t \in A$, then $s \in A$ and $t \in A$.

$\nabla_{\neg\wedge}$ If $\neg(s \wedge t) \in A$, then $\neg s \in A$ or $\neg t \in A$.

$\nabla_\equiv$ If $s \equiv t \in A$, then $\{s, t\} \subseteq A$ or $\{\neg s, \neg t\} \subseteq A$.

$\nabla_{\not\equiv}$ If $\neg(s \equiv t) \in A$, then $\{s, \neg t\} \subseteq A$ or $\{\neg s, t\} \subseteq A$.

Note that this includes the four properties we used to define $\nabla_{\mathbf{Prop}\to}$. We can prove Lemmas 6.4.3 and 6.4.4 using the same techniques as Lemmas 6.4.5 and 6.4.6. We sketch the proofs and leave the reader to check the details.

Proof (Proof of Lemma 6.4.3) ∇_c,$\nabla_\bot$,$\nabla_{\neg\top}$ since **Closed** does not apply.

$\nabla_{\neg\neg}$ since **DNeg** does not apply.

$\nabla_\to$ since **Imp** does not apply.

$\nabla_{\neg\to}$ since **NegImp** does not apply.

$\nabla_\vee$ since **Or** does not apply.

$\nabla_{\neg\vee}$ since otherwise either **DeMorgan** or **And** would apply.

$\nabla_\wedge$ since **And** does not apply.

$\nabla_{\neg\wedge}$ since otherwise either **DeMorgan** or **Or** would apply.

$\nabla_\equiv$ since **Boolean=** does not apply.

$\nabla_{\not\equiv}$ since **Boolean≠** does not apply. ∎

Proof (Proof of Lemma 6.4.4) The proof is by induction on the propositional formula s:

- For variables p use ∇_c.
- For $\top$ use $\nabla_{\neg\top}$.
- For $\bot$ use $\nabla_\bot$.
- For $\neg s$ use $\nabla_{\neg\neg}$.
- For $s \to t$ use $\nabla_\to$ and $\nabla_{\neg\to}$.
- For $s \vee t$ use $\nabla_\vee$ and $\nabla_{\neg\vee}$.
- For $s \wedge t$ use $\nabla_\wedge$ and $\nabla_{\neg\wedge}$.
- For $s \equiv t$ use $\nabla_\equiv$ and $\nabla_{\neg\equiv}$. ∎

Finally, we can prove propositional completeness.

Proof (Proof of Theorem 6.4.1) Suppose A is an unsatisfiable finite set of propositional formulas and A has no propositional refutation. We will prove a contradiction.

First we define the height $h(s)$ of a propositional formula s by induction as follows:

$$\begin{array}{rr} h(p) = 1 & h(\top) = 1 \\ h(\bot) = 1 & h(\neg s) = 1 + h(s) \\ h(s \to t) = 1 + max(h(s), h(t)) & h(s \wedge t) = 1 + max(h(s), h(t)) \\ h(s \vee t) = 1 + max(h(s), h(t)) & h(s \equiv t) = 1 + max(h(s), h(t)) \end{array}$$

Let H be the maximum height of the propositional formulas in A. There are only finitely many names in $\mathcal{N}A$ and there are only finitely many propositional formulas s with maximum height H such that $\mathcal{N}s \subseteq \mathcal{N}A$. Let F be this finite set. Clearly $A \subseteq F$.

By Proposition 6.4.2 there is some instance of a rule from Figures 6.6 and 6.7 of the form

$$\frac{A_1 \vdash \bot \qquad \dots \qquad A_n \vdash \bot}{A \vdash \bot}$$

Since A has no propositional refutation, at least one A_i must have no propositional refutation. Examining the rules, we also see that $A_i \subseteq F$ since the premisses never have greater height than any formula in the conclusion and every name occurring free in the premisses must have occurred free in the conclusion. Also, since A is unsatisfiable and A is a (proper) subset of A_i, A_i is also unsatisfiable. Let C^0 be A and C^1 be A. We will define an infinite sequence of sets C^n recursively.

Suppose we have an unsatisfiable set $C^n \subseteq F$ which has no propositional refutation. As above, there must be an instance of one of the rules with conclusion $C^n \vdash \bot$ and a premiss $C^{n+1} \vdash \bot$ where C^{n+1} has no propositional refutation. Again, C^n is a proper subset of C^{n+1} and $C^{n+1} \subseteq F$. By induction we have an infinite chain

$$C^0 \subset C^1 \subset C^2 \subset \cdots \subset C^n \subset \cdots$$

Since each inclusion is proper, $\bigcup_n C^n$ must be infinite. On the other hand $\bigcup_n C^n$ is a subset of the finite set F since each $C^n \subseteq F$. This is a contradiction. ■

Implicit in the proof above is the fact that any search for a proof will either terminate with a refutation or with a satisfiable set of propositional formulas extending the set we intended to refute. Consequently, we have a decision procedure for propositional formulas.

6.5 Quantifiers

We now consider refutation rules for quantifiers and the corresponding tableaux. The refutation rules are shown in Figure 6.10 and corresponding tableau views are shown in Figure 6.11. As with the propositional case, we include **DeMorgan** rules to exchange negation with a quantifier. To apply the **Forall** rule we must give a term t of the same type as the bound variable. Discovering the correct t to use in a given proof often requires creativity. To apply the **Exists** rule we must give a variable y of the same type as the bound variable. The only requirement

$$\textbf{DeMorgan}\ \frac{A, \exists x.\neg s \vdash \bot}{A \vdash \bot}\ \neg(\forall x.s) \in A$$

$$\textbf{DeMorgan}\ \frac{A, \forall x.\neg s \vdash \bot}{A \vdash \bot}\ \neg(\exists x.s) \in A \qquad \textbf{Forall}\ \frac{A, s^x_t \vdash \bot}{A \vdash \bot}\ \forall x.s \in A$$

$$\textbf{Exists}\ \frac{A, s^x_y \vdash \bot}{A \vdash \bot}\ \exists x.s \in A,\ y \notin \mathcal{N}A$$

Figure 6.10: Derivable refutation rules for quantifiers

$$\textbf{DeMorgan}\frac{\neg\forall x.s}{\exists x.\neg s} \qquad \textbf{DeMorgan}\frac{\neg\exists x.s}{\forall x.\neg s} \qquad \textbf{Forall}\frac{\forall x.s}{s^x_t}$$

$$\textbf{Exists}\frac{\exists x.s}{s^x_y}\ \text{where } y \text{ is fresh}$$

Figure 6.11: Tableau views for quantifier rules

is that y is not free in the conclusion sequent. Note that if the bound variable x does not occur in the conclusion sequent then we can use x for y. For the tableau view of the **Exists** rule in Figure 6.11 we simply say **y is fresh** to mean y is not free in any formula on the branch. Again, if x is not free in any formula on the branch, then we can simply use the same name x.

Combining these rules with the propositional rules will be enough to prove any first-order sequent. We say a set A has a **first-order refutation** if A is first-order and there is a refutation of A using only the rules in Figures 6.6, 6.7 and 6.10.

Unlike the propositional case, applying the rules will not terminate in general. This is because the **Forall** rule can be applied infinitely many times (by choosing different terms t). Consequently, the rules will not give a decision procedure. Indeed, the set of provable first-order sequents is not decidable. However, we will be able to argue completeness in a similar way as the propositional case.

Suppose we have a finite set of first-order formulas A for which there is no first-order refutation. Applying the same procedure as in the proof of propositional completeness (Theorem 6.4.1) we can argue that either A is satisfiable or some rule must apply. The instance of the rule must have a premiss for which there is also no first-order refutation. Continuing this process we obtain a chain of finite sets of formulas

$$A \subset C^1 \subset \cdots \subset C^n \subset \cdots .$$

This time we will prove that the set $\bigcup_n C^n$ is satisfiable.

Before embarking on another completeness result we consider a few examples.

We first consider two small examples. Let α be a sort different from B and let $p : \alpha B$ and $x, a, b : \alpha$ be names. We will give a tableau proof of

$$(pa \vee pb) \to \exists x.px$$

We first negate the statement (applying **Contra**) and then apply **NegImp** to obtain

$$\begin{array}{c} \neg((pa \vee pb) \to \exists x.px) \\ pa \vee pb \\ \neg\exists x.px \end{array}$$

After applying **DeMorgan** we have $\forall x.\neg px$ on the branch and after applying **Or** we have the following tableau:

$$\begin{array}{c} \neg((pa \vee pb) \to \exists x.px) \\ pa \vee pb \\ \neg\exists x.px \\ \forall x.\neg px \\ \hline \begin{array}{c|c} pa & pb \end{array} \end{array}$$

On the left branch we apply **Forall** with the term a and this branch is then closed. On the right branch we apply **Forall** with the term b and then both branches are closed. The final tableau is shown in Figure 6.12. The reader is encouraged to consider other possible tableau proofs by reordering the application of the rules.

By a similar process we can construct a tableau proof of

$$(\forall x.px) \to (pa \wedge pb)$$

such as the one shown in Figure 6.13.

Note that in both of these examples we have used the **Forall** rule, in spite of the fact that the first example contained $\exists$ and the second example contained $\forall$. We have not yet had an example using the **Exists** rule.

Consider the following simple example with a name $r : \alpha\alpha B$ and names $x, y, z : \alpha$ where α is a sort other than B:

$$(\exists x.rxx) \to \exists yz.ryz$$

$$\begin{array}{c|c}
\multicolumn{2}{c}{\neg((pa \vee pb) \to \exists x.px)} \\
\multicolumn{2}{c}{pa \vee pb} \\
\multicolumn{2}{c}{\neg\exists x.px} \\
\multicolumn{2}{c}{\forall x.\neg px} \\
\hline
pa & pb \\
\neg pa & \neg pb
\end{array}$$

Figure 6.12: Tableau example with ∃

$$\begin{array}{c|c}
\multicolumn{2}{c}{\neg((\forall x.px) \to pa \wedge pb)} \\
\multicolumn{2}{c}{\forall x.px} \\
\multicolumn{2}{c}{\neg(pa \wedge pb)} \\
\multicolumn{2}{c}{\neg pa \vee \neg pb} \\
\hline
\neg pa & \neg pb \\
pa & pb
\end{array}$$

Figure 6.13: Tableau example with ∀

After negating this and applying **NegImp** we have the tableau

$$\begin{array}{c}
\neg((\exists x.rxx) \to \exists yz.ryz) \\
\exists x.rxx \\
\neg\exists yz.ryz
\end{array}$$

We can now apply **Exists** to $\exists x.rxx$ with any name of type α that is not free in the branch. Let us simply use x. This adds rxx to the branch. The idea now is to apply **DeMorgan** and **Forall** to $\neg\exists yz.ryz$ until we obtain a contradiction. Each time we apply the **Forall** rule we use the term x. In the end we obtain the tableau shown in Figure 6.14 in which the single branch is closed.

We now turn to a larger example. Let α and β be sorts different from B, let $r : \alpha\beta B$, $x : \alpha$ and $y : \beta$ be names of the given types. Consider the following formula:

$$(\exists y.\forall x.rxy) \to \forall x.\exists y.rxy \tag{6.2}$$

Is this formula valid? It may be helpful to translate the formula into everyday language. Suppose α represents the students taking this class and β represents various reading materials. Suppose rxy means student x reads y. The left side of the implication

$$\exists y.\forall x.rxy$$

$$\begin{array}{c}
\neg((\exists x.rxx) \to \exists yz.ryz) \\
\exists x.rxx \\
\neg\exists yz.ryz \\
rxx \\
\forall y.\neg\exists z.ryz \\
\neg\exists z.rxz \\
\forall z.\neg rxz \\
\neg rxx
\end{array}$$

Figure 6.14: Tableau example using **Exists**

means there is some reading material y that every student x reads. This is, of course, true, because all of you are reading these lecture notes. The right hand side of the implication

$$\forall x.\exists y.rxy$$

means that every student x reads some material y. This is, of course, a weaker statement. It only asserts that every students reads something, not that all students are reading the same thing. The right hand side of the implication would still be true even if some student x reads the newspaper but does not read these lecture notes. This interpretation suggests that (6.2) is a valid statement, but that the converse implication

$$(\forall x.\exists y.rxy) \to \exists y.\forall x.rxy \tag{6.3}$$

is not valid.

We construct a tableau proof of (6.2) in a step by step fashion. We first negate the statement (applying **Contra**) and then apply **NegImp** to obtain

$$\begin{array}{c}
\neg((\exists y.\forall x.rxy) \to \forall x.\exists y.rxy) \\
\exists y.\forall x.rxy \\
\neg(\forall x.\exists y.rxy)
\end{array}$$

According the only branch of this tableau, there is some y that every x reads but it is not the case that every x reads some y. Since there is some reading material y that every x reads, we can choose some name to stand for such reading material. Let us use the name $n : \beta$ (short for "lecture **n**otes"). Applying **Exists** with name n we extend the branch to obtain

$$\begin{array}{c}
\neg((\exists y.\forall x.rxy) \to \forall x.\exists y.rxy) \\
\exists y.\forall x.rxy \\
\neg(\forall x.\exists y.rxy) \\
\forall x.rxn
\end{array}$$

Now we apply **DeMorgan** to add $\exists x.\neg(\exists y.rxy)$ to the branch indicating that some student x does not read anything. We can apply **Exists** to this rule by giving a name of type α. We use the name $u : \alpha$ (short for "**u**nfortunate") to stand for this student. Following the use of **Exists** by **DeMorgan** we obtain

$$\begin{array}{c}
\neg((\exists y.\forall x.rxy) \to \forall x.\exists y.rxy) \\
\exists y.\forall x.rxy \\
\neg(\forall x.\exists y.rxy) \\
\forall x.rxn \\
\exists x.\neg(\exists y.rxy) \\
\neg(\exists y.ruy) \\
\forall y.\neg ruy
\end{array}$$

After a careful look at this tableau, you should recognize that the two formulas $\forall x.rxn$ and $\forall y.\neg ruy$ cannot both be true. If every student is reading the lecture notes, then there cannot by an unfortunate student who reads nothing. To make the conflict explicit we apply the **Forall** rule to $\forall y.\neg ruy$ with term n and obtain

$$\begin{array}{c}
\neg((\exists y.\forall x.rxy) \to \forall x.\exists y.rxy) \\
\exists y.\forall x.rxy \\
\neg(\forall x.\exists y.rxy) \\
\forall x.rxn \\
\exists x.\neg(\exists y.rxy) \\
\neg(\exists y.ruy) \\
\forall y.\neg ruy \\
\neg run
\end{array}$$

We have now concluded that the unfortunate student is not reading the lecture notes. But every student reads the lecture notes. To make this explicit we apply **Forall** to $\forall x.rxn$ with term u and obtain the tableau where the only branch is now closed:

$$\begin{array}{c}
\neg((\exists y.\forall x.rxy) \to \forall x.\exists y.rxy) \\
\exists y.\forall x.rxy \\
\neg(\forall x.\exists y.rxy) \\
\forall x.rxn \\
\exists x.\neg(\exists y.rxy) \\
\neg(\exists y.ruy) \\
\forall y.\neg ruy \\
\neg run \\
run
\end{array}$$

We have given a first-order refutation of the negation of (6.2) and hence a (tableau) proof of (6.2). In particular, (6.2) is valid.

It is worthwhile to consider the converse implication (6.3) as well. As we have argued informally above, (6.3) is not valid. If we negate (6.3) and apply the obvious rules we obtain a tableau

$$\begin{gathered}\neg((\forall x.\exists y.rxy) \to \exists y.\forall x.rxy)\\ \forall x.\exists y.rxy\\ \neg(\exists y.\forall x.rxy)\\ \forall y.\neg\forall x.rxy\end{gathered}$$

To make progress we must apply the **Forall** rule to either $\forall x.\exists y.rxy$ with some term of type α or $\forall y.\neg\forall x.rxy$ with some term of type β. There are no reasonable terms of either type to use here. Suppose we apply the **Forall** rule to each of the two using the names $u : \alpha$ (the unfortunate student) and $n : \beta$ (the lecture notes). After applying **Forall** twice and **DeMorgan** we obtain

$$\begin{gathered}\neg((\forall x.\exists y.rxy) \to \exists y.\forall x.rxy)\\ \forall x.\exists y.rxy\\ \neg(\exists y.\forall x.rxy)\\ \forall y.\neg\forall x.rxy\\ \exists y.ruy\\ \neg\forall x.rxn\\ \exists x.\neg rxn\end{gathered}$$

We may be tempted to apply **Exists** to $\exists y.ruy$ using name n and $\exists x.\neg rxn$ using name u. If we did this, then the branch would be closed. That would be a disaster since we would have proven formula (6.3) which was not valid! What prevents this disaster? Neither application of **Exists** would be legal since both n and u occur in the branch. (Neither are **fresh**.) If we apply **Exists** using fresh names n' and u', then we obtain

$$\begin{gathered}\neg((\forall x.\exists y.rxy) \to \exists y.\forall x.rxy)\\ \forall x.\exists y.rxy\\ \neg(\exists y.\forall x.rxy)\\ \forall y.\neg\forall x.rxy\\ \exists y.ruy\\ \neg\forall x.rxn\\ \exists x.\neg rxn\\ run'\\ \neg ru'n\end{gathered}$$

which is not closed. Note that we could continue trying to build this tableau by applying **Forall** to $\forall x.\exists y.rxy$ and $\forall y.\neg\forall x.rxy$ with other terms (e.g., u' and n'). The attempt to close the branch can go on forever but the branch will never close.

Now we turn to another example. Let N be a sort other than B and let $s : NNB$ and $x, y, z : N$ be names. We can think of sxy as meaning "x has successor y," or "y is the successor of x." We can express the idea that every element has a successor with the formula

$$\forall x.\exists y.sxy.$$

If we assume every element has a successor, then we should be able to prove

$$\forall x.\exists yz.sxy \wedge syz.$$

We formulate the statement as the single formula

$$(\forall x.\exists y.sxy) \to \forall x.\exists yz.sxy \wedge syz.$$

We construct a tableau proof of this. We first negate the statement (applying **Contra**) and then apply **NegImp** and **DeMorgan** to obtain

$$\begin{array}{c} \neg((\forall x.\exists y.sxy) \to \forall x.\exists yz.sxy \wedge syz) \\ \forall x.\exists y.sxy \\ \neg\forall x.\exists yz.sxy \wedge syz \\ \exists x.\neg\exists yz.sxy \wedge syz \end{array}$$

We can apply **Exists** using the name x (since it does not occur free in the branch) and include

$$\neg\exists yz.sxy \wedge syz$$

on the branch. This new formula says there is no successor y of x which has a successor z. We can obtain a successor of x using by applying **Forall** to $\forall x.\exists y.sxy$ with the term x. When we do this we have

$$\exists y.sxy$$

on the branch. Since y does not occur free on the branch we can apply **Exists** with y and include

$$sxy$$

on the branch. We now want to use this y. We do this by applying **DeMorgan** to $\neg\exists yz.sxy \wedge syz$ and then **Forall** to $\forall y.\neg\exists z.sxy \wedge syz$ to obtain

$$\begin{array}{c} \neg((\forall x.\exists y.sxy) \to \forall x.\exists yz.sxy \wedge syz) \\ \forall x.\exists y.sxy \\ \neg\forall x.\exists yz.sxy \wedge syz \\ \exists x.\neg\exists yz.sxy \wedge syz \\ \neg\exists yz.sxy \wedge syz \\ \exists y.sxy \\ sxy \\ \forall y.\neg\exists z.sxy \wedge syz \\ \neg\exists z.sxy \wedge syz \end{array}$$

Since y is a successor of x we know

$$\neg\exists z.sxy \wedge syz$$

means there is no successor z of y. We also know we can obtain a successor of anything using $\forall x.\exists y.sxy$. We apply **Forall** to this formula with y. Note that when we substitute, the bound y will be renamed to another variable y' to avoid capture. Hence we add

$$\exists y'.syy'$$

to the branch. We can now apply **Exists** with any variable of type N that does not occur on the branch. We cannot use x or y. We can use y' (unless y' happens to be x), but a reasonable choice in this context is z. We use z and add

$$syz$$

to the branch. We now apply **DeMorgan** to $\neg\exists z.sxy \wedge syz$, then **Forall** with this z, then **DeMorgan** and **Or** to obtain

$$\begin{array}{c}
\neg((\forall x.\exists y.sxy) \to \forall x.\exists yz.sxy \wedge syz)\\
\forall x.\exists y.sxy\\
\neg\forall x.\exists yz.sxy \wedge syz\\
\exists x.\neg\exists yz.sxy \wedge syz\\
\neg\exists yz.sxy \wedge syz\\
\exists y.sxy\\
sxy\\
\forall y.\neg\exists z.sxy \wedge syz\\
\neg\exists z.sxy \wedge syz\exists y'.syy'\\
syz\\
\forall z.\neg(sxy \wedge syz)\\
\neg(sxy \wedge syz)\\
\neg sxy \vee \neg syz\\
\hline
\begin{array}{c|c} \neg sxy & \neg syz \end{array}
\end{array}$$

Note that both branches are closed.

Figure 6.15 shows the tableau views of the refutation rules we have introduced so far (except for **Closed**$\neg =$ and **ClosedSym**). We will also refer to the tableau views of such refutation rules as **tableau rules**.

6.6 Termination and Completeness with Restrictions

We define a restricted set of formulas. Let I be a sort other than B. There are infinitely many names of type I. Let $\mathcal{V}$ and $\mathcal{P}$ be two disjoint, infinite sets of

$$\textbf{Closed}\,\frac{s, \neg s}{\quad} \qquad \textbf{Closed}\,\frac{\bot}{\quad} \qquad \textbf{Closed}\,\frac{\neg\top}{\quad} \qquad \textbf{DNeg}\,\frac{\neg\neg s}{s}$$

$$\textbf{DeMorgan}\,\frac{\neg(s \vee t)}{\neg s \wedge \neg t} \qquad \textbf{DeMorgan}\,\frac{\neg(s \wedge t)}{\neg s \vee \neg t} \qquad \textbf{And}\,\frac{s \wedge t}{s, t} \qquad \textbf{Or}\,\frac{s \vee t}{s \mid t}$$

$$\textbf{Imp}\,\frac{s \to t}{t \mid \neg s} \qquad \textbf{NegImp}\,\frac{\neg(s \to t)}{s, \neg t} \qquad \textbf{Boolean=}\,\frac{s \equiv t}{s, t \mid \neg s, \neg t}$$

$$\textbf{Boolean}{\neq}\,\frac{\neg(s \equiv t)}{s, \neg t \mid t, \neg s} \qquad \textbf{DeMorgan}\,\frac{\neg\forall x.s}{\exists x.\neg s} \qquad \textbf{DeMorgan}\,\frac{\neg\exists x.s}{\forall x.\neg s}$$

$$\textbf{Forall}\,\frac{\forall x.s}{s^x_t} \qquad \textbf{Exists}\,\frac{\exists x.s}{s^x_y}\ \text{where } y \text{ is fresh}$$

Figure 6.15: Tableau rules so far

names of type I. In this section we will use the word **variable** to mean a member of $\mathcal{V}$ and the word **parameter** to mean a member of $\mathcal{P}$. Disjointness means $\mathcal{V} \cap \mathcal{P} = \emptyset$. In words, a name cannot be both a variable and a parameter.

- We use x, y, z for variables.
- We use a, b, c for parameters. We fix one particular parameter $a^0 \in \mathcal{P}$ which we will use for a special purpose.
- We use w to range over $\mathcal{V} \cup \mathcal{P}$.
- For terms s we define

  $$\mathcal{V}s = \mathcal{N}s \cap \mathcal{V} \text{ (the variables free in } s\text{)}$$

  and for sets A of terms we define

  $$\mathcal{V}A = \mathcal{N}A \cap \mathcal{V} \text{ (the variables free in } A\text{).}$$

- For terms s we define

  $$\mathcal{P}s = \mathcal{N}s \cap \mathcal{P} \text{ (the parameters free in } s\text{)}$$

  and for sets A of terms we define

  $$\mathcal{P}A = \mathcal{N}A \cap \mathcal{P} \text{ (the parameters free in } A\text{).}$$

$$\textbf{Closed}\,\frac{s,\neg s}{\quad} \qquad \textbf{And}\,\frac{s\wedge t}{s,t} \qquad \textbf{Or}\,\frac{s\vee t}{s\mid t}$$

$$\textbf{Forall}\,\frac{\forall x.s}{s^x_a}\ \text{where } a\in\mathcal{P} \text{ is a parameter on the branch or } a^0$$

$$\textbf{Exists}\,\frac{\exists x.s}{s^x_a}\ \text{where } a\in\mathcal{P} \text{ is fresh and there is no}$$

$$b\in\mathcal{P} \text{ such that } s^x_b \text{ is on the branch.}$$

Figure 6.16: Restricted tableau calculus $\mathcal{T}_{\mathbf{PLN}}$

We define the set of **PLN**-formulas s by the grammar

$$pw\cdots w \mid \neg pw\cdots w \mid s\vee s \mid s\wedge s \mid \forall x.s \mid \exists x.s$$

where p is a name of type $I\cdots IB$. Notice that the bound names in **PLN**-formulas are all variables (members of $\mathcal{V}$). Note also that each **PLN**-formula is a first-order formula (see Section 5.7). Here we are most interested in **closed PLN-formulas**:

$$\mathbf{PLN}_c = \{s \mid s \text{ is a } \mathbf{PLN}\text{-formula and } \mathcal{V}s=\emptyset\}.$$

Along with the restricted set of formulas we also define a restricted calculus. The restricted tableau calculus $\mathcal{T}_{\mathbf{PLN}}$ is given by the rules in Figure 6.16.

Suppose $A\subseteq\mathbf{PLN}_c$ is the finite set of closed **PLN**-formulas occurring on an open branch of some tableau. If none of the $\mathcal{T}_{\mathbf{PLN}}$-rules apply to the branch, then the following **Hintikka properties** hold for A:

∇_c For all formulas s, $s\notin A$ or $\neg s\notin A$.

$\nabla_\vee$ If $s\vee t\in A$, then $s\in A$ or $t\in A$.

$\nabla_\wedge$ If $s\wedge t\in A$, then $s\in A$ and $t\in A$.

$\nabla_\forall$ If $\forall x.s\in A$, then $s^x_a\in A$ for all $a\in\mathcal{P}A\cup\{a^0\}$.

$\nabla_\exists$ If $\exists x.s\in A$, then $s^x_a\in A$ for some $a\in\mathcal{P}$.

For any set $A\subseteq\mathbf{PLN}_c$, if A satisfies these Hintikka properties, then A is satisfiable. We record this fact in the following model existence theorem.

Theorem 6.6.1 (Model Existence Theorem) Suppose $A\subseteq\mathbf{PLN}_c$ and A satisfies the Hintikka properties ∇_c, $\nabla_\vee$, $\nabla_\wedge$, $\nabla_\forall$ and $\nabla_\exists$. There is a logical interpretation $\mathcal{I}$ such that $\mathcal{I}\vDash s$ for all $s\in A$.

Proof We define $\mathcal{I}$ on types as follows:

$$\begin{array}{rcl} \mathcal{I}B & = & \{0,1\} \\ \mathcal{I}I & = & \mathcal{P}A \cup \{a^0\} \\ \mathcal{I}\alpha & = & \{\emptyset\} \text{ for sorts } \alpha \notin \{B,I\} \\ \mathcal{I}(\sigma\tau) & = & \text{all functions from } \mathcal{I}\sigma \text{ to } \mathcal{I}\tau. \end{array}$$

For each type σ we define a default element $d^\sigma \in \mathcal{I}\sigma$ as follows:

$$\begin{array}{rcl} d^B & = & 0 \\ d^I & = & a^0 \\ d^\alpha & = & \emptyset \text{ for sorts } \alpha \notin \{B,I\} \\ d^{\sigma\tau} & = & \text{the constant function } (\lambda x \in \mathcal{I}\sigma . d^\tau). \end{array}$$

For each logical constant $c : \sigma$ we choose $\mathcal{I}c$ to be the unique element of $\mathcal{I}\sigma$ with the property corresponding to the constant. For each name $a \in \mathcal{P}A \cup \{a^0\}$ we take

$$\mathcal{I}a = a.$$

For each name $p : I \cdots IB$ such that $p \in \mathcal{N}A$, we define $p \in \mathcal{I}(I \cdots IB)$ such that for $b^1, \ldots, b^n \in \mathcal{I}I$ we have

$$(\mathcal{I}p)b^1 \cdots b^n = \begin{cases} 1 & \text{if } (pb^1 \cdots b^n) \in A \\ 0 & \text{otherwise.} \end{cases}$$

For every other name $u : \sigma$ we take

$$\mathcal{I}u = d^\sigma.$$

$\mathcal{I}$ is clearly a logical interpretation.

It remains to prove that $\mathcal{I} \models s$ for every $s \in A$. Rephrased, we prove (by induction on PLN-formulas) for every $s \in$ PLN, if $s \in A$, then $\hat{\mathcal{I}}s = 1$. We consider each case:

- Assume $(pb^1 \cdots b^n) \in A$. By definition $\hat{\mathcal{I}}(pb^1 \cdots b^n) = 1$.
- Assume $(\neg pb^1 \cdots b^n) \in A$. By ∇_c we know $(pb^1 \cdots b^n) \notin A$. Hence $\hat{\mathcal{I}}(pb^1 \cdots b^n) = 0$ and so $\hat{\mathcal{I}}(\neg pb^1 \cdots b^n) = 1$.
- Assume $(s^1 \vee s^2) \in A$. By $\nabla_\vee$ we know $s^1 \in A$ or $s^2 \in A$. By inductive hypothesis either $\hat{\mathcal{I}}s^1 = 1$ or $\hat{\mathcal{I}}s^2 = 1$. Hence $\hat{\mathcal{I}}(s^1 \vee s^2) = 1$.
- Assume $(s^1 \wedge s^2) \in A$. By $\nabla_\wedge$ we know $s^1 \in A$ and $s^2 \in A$. By inductive hypothesis $\hat{\mathcal{I}}s^1 = 1$ and $\hat{\mathcal{I}}s^2 = 1$. Hence $\hat{\mathcal{I}}(s^1 \wedge s^2) = 1$.
- Assume $(\forall x.s) \in A$. By $\nabla_\forall$ we know $s^x_a \in A$ for all $a \in \mathcal{P}A \cup \{a^0\}$. By inductive hypothesis $\hat{\mathcal{I}}s^x_a = 1$ for all $a \in \mathcal{I}I$. Using Lemma 4.1.3 we know $\widehat{\mathcal{I}_{x,a}}s = 1$ for all $a \in \mathcal{I}I$. Hence $\hat{\mathcal{I}}(\forall x.s) = 1$.

- Assume $(\exists x.s) \in A$. By $\nabla_\exists$ we know $s^x_a \in A$ for some $a \in \mathcal{P}$. By inductive hypothesis $\hat{\mathcal{I}}s^x_a = 1$. Using Lemma 4.1.3 we know $\widehat{\mathcal{I}_{x,a}}s = 1$. Hence $\hat{\mathcal{I}}(\exists x.s) = 1$. ∎

Given any restricted tableau calculus $\mathcal{T}$ (e.g., $\mathcal{T}_{\mathbf{PLN}}$) and set of formulas $\mathcal{F}$ (e.g., $\mathbf{PLN}_c$) there are two important issues we will consider: termination and completeness.

1. Does $\mathcal{T}$ **terminate** on $\mathcal{F}$? Informally, if we start with a finite set $A \subseteq \mathcal{F}$, are we guaranteed we cannot apply the rules $\mathcal{T}$ infinitely often?
2. Is $\mathcal{T}$ **complete** for $\mathcal{F}$? That is, are there $\mathcal{T}$-refutations for finite $A \subseteq \mathcal{F}$ whenever A is unsatisfiable?

We already have the notion of a terminating relation from Section 3.5. To apply this notion here, we define a relation $\xrightarrow{\mathcal{T}}$ on finite subsets of $\mathcal{F}$ by saying $A \xrightarrow{\mathcal{T}} A'$ if there is a refutation step which is an instance of a rule from $\mathcal{T}$ where $A \vdash \bot$ is the conclusion of the step and $A' \vdash \bot$ is one of the premisses. Stated in terms of tableaux, $A \xrightarrow{\mathcal{T}} A'$ if we can extend a branch with formulas A to a branch with formulas A' (possibly creating other branches as well) using a rule from $\mathcal{T}$. Note that if $A \xrightarrow{\mathcal{T}} A'$ then we know A is a proper subset of A' (i.e., $A \subset A'$).

Completeness means that we are guaranteed that if a finite set $A \subseteq \mathcal{F}$ is unsatisfiable, then there is a refutation of A using the rules in $\mathcal{T}$. Another way to say "there is a refutation of A using the rules in $\mathcal{T}$" is to say that we can construct a complete tableau using the rules of $\mathcal{T}$ starting from a single branch containing only formulas from A. When we prove a completeness result, we will actually prove the contrapositive. We will assume there is no refutation of A using the rules of $\mathcal{T}$ and then construct a logical interpretation $\mathcal{I}$ satisfying all the formulas in A. Now that we have the Model Existence Theorem above (Theorem 6.6.1) we can prove completeness in an even simpler way (assuming $A \subseteq \mathbf{PLN}_c$). Instead of constructing an interpretation of such an A, we need only find a set $H \subseteq \mathbf{PLN}_c$ such that $A \subseteq H$ and H satisfies the Hintikka properties ∇_c, $\nabla_\vee$, $\nabla_\wedge$, $\nabla_\forall$ and $\nabla_\exists$.

Suppose we have a set $\mathcal{F}$ of formulas such that $\mathcal{F} \subseteq \mathbf{PLN}_c$. Suppose further that for any finite $A \subseteq \mathcal{F}$, if $A \xrightarrow{\mathcal{T}_{\mathbf{PLN}}} A'$, then $A' \subseteq \mathcal{F}$. (This condition certainly holds for the set $\mathbf{PLN}_c$.) Then we have completeness whenever we have termination.

Theorem 6.6.2 (Termination implies Completeness) Let $\mathcal{F}$ be a set of formulas such that $\mathcal{F} \subseteq \mathbf{PLN}_c$. Suppose for any finite $A \subseteq \mathcal{F}$, if $A \xrightarrow{\mathcal{T}_{\mathbf{PLN}}} A'$, then $A' \subseteq \mathcal{F}$. If $\xrightarrow{\mathcal{T}_{\mathbf{PLN}}}$ is terminating on $\mathcal{F}$, then $\xrightarrow{\mathcal{T}_{\mathbf{PLN}}}$ is complete for $\mathcal{F}$.

Proof Suppose A has no refutation using the $\mathcal{T}_{\mathbf{PLN}}$-rules. Then there must be some finite set $H \subseteq \mathbf{PLN}_c$ such that $A \xrightarrow{\mathcal{T}_{\mathbf{PLN}}}{}^{*} H$, H is open (that is, there is no s

such that $s \in H$ and $\neg s \in H$) and there is no A' such that $H \xrightarrow{\mathcal{T}_{\mathbf{PLN}}} A'$. Informally, H is an open branch such that none of the $\mathcal{T}_{\mathbf{PLN}}$-rules applies to H. Since H is open, ∇_c holds for H. Since none of the $\mathcal{T}_{\mathbf{PLN}}$-rules applies to H, the remaining Hintikka conditions $\nabla_\vee$, $\nabla_\wedge$, $\nabla_\forall$ and $\nabla_\exists$ all hold for H. By the Model Existence Theorem (Theorem 6.6.1) there is a logical interpretation $\mathcal{I}$ such that $\mathcal{I} \vDash s$ for all $s \in H$. In particular, $\mathcal{I} \vDash s$ for all $s \in A$. ■

The restricted tableau calculus $\mathcal{T}_{\mathbf{PLN}}$ does not terminate on $\mathbf{PLN}_c$. Consider the formula $\forall x \exists y.rxy$. This is clearly in $\mathbf{PLN}_c$. We can easily apply the restricted **Forall** and **Exists** rules forever as indicated:

$$\begin{array}{c} \forall x \exists y.rxy \\ \exists y.ra^0y \\ ra^0a^1 \\ \exists y.ra^1y \\ ra^1a^2 \\ \exists y.ra^2y \\ ra^2a^3 \\ \vdots \end{array}$$

On the other hand, $\mathcal{T}_{\mathbf{PLN}}$ will terminate on some subsets of $\mathbf{PLN}_c$. For any s, let $\mathrm{Sub}(s)$ be the set of subterms of s of type B. We will call these subterms **subformulas** of s. For a set A of formulas, let $\mathrm{Sub}(A)$ be $\bigcup_{s \in A} \mathrm{Sub}(s)$.

We will prove termination for (hence completeness of) the following three subsets of $\mathbf{PLN}_c$ relative to $\mathcal{T}_{\mathbf{PLN}}$.

$$\mathbf{PLN}_c^{no\exists} = \{s \in \mathbf{PLN}_c \mid \text{there is no } \exists y.t \in \mathrm{Sub}(s)\}$$

$$\mathbf{PLN}_c^{cl\exists} = \{s \in \mathbf{PLN}_c \mid \text{for every } \exists y.t \in \mathrm{Sub}(s) \text{ we have } \mathcal{V}(\exists y.t) = \emptyset\}$$

$$\mathbf{PLN}_c^{\exists *} = \{s \in \mathbf{PLN}_c \mid \text{for every } \exists y.t \in \mathrm{Sub}(s), x \in \mathcal{V}(\exists y.t), x \text{ is bound by an } \exists\}$$

The phrase "x is bound by an $\exists$" is informal and possibly misleading. (Recall that only λ can bind a variable.) When we speak of $\exists$ or $\forall$ binding a variable we actually mean the relevant λ binder occurs as $\exists(\lambda x.[])$ or $\forall(\lambda x.[])$. We could make the phrase "x is bound by an $\exists$" precise using contexts. The important fact that we will use is the following: If $\forall x.s \in \mathbf{PLN}_c^{\exists *}$, $a \in \mathcal{P}$ and $\exists y.t \in \mathrm{Sub}(s_a^x)$, then $\exists y.t \in \mathrm{Sub}(\forall x.s)$.

Clearly we have

$$\mathbf{PLN}_c^{no\exists} \subseteq \mathbf{PLN}_c^{cl\exists} \subseteq \mathbf{PLN}_c^{\exists *} \subseteq \mathbf{PLN}_c.$$

It is enough to prove termination of $\mathbf{PLN}_c^{\exists *}$, since termination will follow for the smaller three. However, termination becomes more difficult to prove each time we include more existential quantifiers.

For any set A of formulas we define $\text{Stock}(A)$ to be the set

$$\begin{aligned}\{\theta s| \quad & s \in \text{Sub}(A), \theta : \text{Nam} \to \text{Ter},\\ & (\forall x \in \mathcal{V}(s).\theta(x) \in \mathcal{P}(A) \cup \{a^0\}),\\ & (\forall u \in \text{Nam} \setminus \mathcal{V}(s).\theta(u) = u)\}\end{aligned}$$

Note that if A is finite, then $\text{Sub}(A)$ is finite, $\mathcal{V}(s)$ is finite for each $s \in \text{Sub}(A)$, and $\mathcal{P}(A) \cup \{a^0\}$ is finite. Hence if A is finite, then $\text{Stock}(A)$ is finite. For finite A, we define $\text{Slack}(A)$ to be the number of elements in $\text{Stock}(A) \setminus A$. That is,

$$\text{Slack}(A) = |\text{Stock}(A) \setminus A|.$$

To prove $\mathcal{T}_{\textbf{PLN}}$ terminates on $\textbf{PLN}_c^{no\exists}$ it suffices to prove that if $A \xrightarrow{\mathcal{T}_{\textbf{PLN}}} A'$, then $\text{Slack}(A) > \text{Slack}(A')$. Suppose $A \xrightarrow{\mathcal{T}_{\textbf{PLN}}} A'$. We consider four cases:

- Assume we applied **And** with $s^1 \wedge s^2 \in A$. Here $A' = A \cup \{s^1, s^2\}$. Note that $\text{Sub}(A) = \text{Sub}(A')$ and so $\text{Stock}(A) = \text{Stock}(A')$. Thus $\text{Slack}(A) > \text{Slack}(A')$.
- Assume we applied **Or** with $s^1 \vee s^2 \in A$. Here $A' = A \cup \{s^1\}$ or $A' = A \cup \{s^2\}$. In either case $\text{Sub}(A) = \text{Sub}(A')$ and so $\text{Stock}(A) = \text{Stock}(A')$. Thus $\text{Slack}(A) > \text{Slack}(A')$.
- Assume we applied **Forall** with $\forall x.s \in A$. Here $A' = A \cup \{s_a^x\}$. In this case we do not know $\text{Sub}(A) = \text{Sub}(A')$. However, we can still conclude $\text{Stock}(A) = \text{Stock}(A')$. To see this, note that the only new subformulas in $\text{Sub}(A')$ are of the form t_a^x where $t \in \text{Sub}(A)$. Any $\theta(t_a^x) \in \text{Stock}(A')$ was already of the form $\theta' t \in \text{Stock}(A)$ with $\theta' = \theta[x := a]$.
- Assume we applied **Exists**. This is impossible because we cannot have $\exists x.s \in A \subseteq \textbf{PLN}_c^{no\exists}$.

Therefore, $\mathcal{T}_{\textbf{PLN}}$ terminates on $\textbf{PLN}_c^{no\exists}$.

Let us next consider termination of $\mathcal{T}_{\textbf{PLN}}$ on $\textbf{PLN}_c^{cl\exists}$. If $A \xrightarrow{\mathcal{T}_{\textbf{PLN}}} A'$ using any rule except **Exists**, then $\text{Slack}(A) > \text{Slack}(A')$. The proof of this fact proceeds in the same way as above. On the other hand, the **Exists** rule introduces a fresh parameter. In this case $\text{Stock}(A')$ may be much larger than $\text{Stock}(A)$. Hence $\text{Slack}(A')$ may be much bigger than $\text{Slack}(A)$ after applying **Exists**.

We can still make use of $\text{Slack}(A)$ to justify termination by making it part of a lexical ordering on two natural numbers. The lexical ordering on pairs of natural numbers is defined by

$$(i^1, j^1) > (i^2, j^2) \text{ iff either } i^1 > i^2 \text{ or both } i^1 = i^2 \text{ and } j^1 > j^2$$

Note that if we know $i^1 > i^2$, then we clearly have $(i^1, j^1) > (i^2, j^2)$. Also if we know $i^1 \geq i^2$ and $j^1 > j^2$, then we have $(i^1, j^1) > (i^2, j^2)$. $\text{Slack}(A)$ will play the role of the j in our termination argument. We need only find a value to play the role of the i. This value must decrease when we apply **Exists** and not increase when we apply the other rules. For any A we define the set $\exists(A)$ to be

$$\{\exists x.s \in \mathrm{Sub}(A) | \forall a \in \mathcal{P}.s^x_a \notin A\}.$$

We can take $|\exists(A)|$ (the number of elements in $\exists(A)$) to play the role of i. Clearly if $A \xrightarrow{\mathcal{T}_{\mathrm{PLN}}} A'$ via the **Exists** rule, then $|\exists(A)| > |\exists(A')|$. It is also easy to check that if $A \xrightarrow{\mathcal{T}_{\mathrm{PLN}}} A'$ via the **And** or **Or** rule, then $|\exists(A)| \geq |\exists(A')|$. Suppose $A \xrightarrow{\mathcal{T}_{\mathrm{PLN}}} A'$ via the **Forall** rule with $\forall x.s \in A$. Here A' is of the form A, s^x_a. In principle we may have introduced a new $\exists y.t^x_a \in \mathrm{Sub}(A')$. However, since $\forall x.s \in A \subseteq \mathbf{PLN}^{cl\exists}_c$, we know $\mathcal{V}(\exists y.t) = \emptyset$. Hence $\exists y.t^x_a$ is the same as $\exists y.t$ which was already in $\mathrm{Sub}(A')$. We conclude that $|\exists(A)| \geq |\exists(A')|$ in this case as well.

We have now proven that whenever $A \xrightarrow{\mathcal{T}_{\mathrm{PLN}}} A'$ one of two things happen:

1. $|\exists(A)| > |\exists(A')|$, or
2. $|\exists(A)| \geq |\exists(A')|$ and $\mathrm{Slack}(A) > \mathrm{Slack}(A')$.

Using the lexical ordering we conclude that $\xrightarrow{\mathcal{T}_{\mathrm{PLN}}}$ terminates on $\mathbf{PLN}^{cl\exists}_c$.

We leave the proof of termination of $\xrightarrow{\mathcal{T}_{\mathrm{PLN}}}$ on $\mathbf{PLN}^{\exists *}_c$ as an exercise. A similar argument with a lexical ordering will work in this case as well.

Exercise 6.6.3 (Termination of $\mathcal{T}_{\mathrm{PLN}}$ on $\mathbf{PLN}^{\exists *}_c$)

a) Define a natural number $\mathrm{Power}^{\exists}(A)$ for finite sets $A \subseteq \mathbf{PLN}^{\exists *}_c$ making use of the set $\exists(A)$.

b) Prove that if $A \xrightarrow{\mathcal{T}_{\mathrm{PLN}}} A'$ via the **Exists** rule, then

$$\mathrm{Power}^{\exists}(A) > \mathrm{Power}^{\exists}(A')$$

c) Prove that if $A \xrightarrow{\mathcal{T}_{\mathrm{PLN}}} A'$ via the **And** or **Or** rule, then

$$\mathrm{Power}^{\exists}(A) \geq \mathrm{Power}^{\exists}(A')$$

d) Prove that if $A \xrightarrow{\mathcal{T}_{\mathrm{PLN}}} A'$ via the **Forall** rule, then

$$\mathrm{Power}^{\exists}(A) \geq \mathrm{Power}^{\exists}(A')$$

e) Conclude that $\xrightarrow{\mathcal{T}_{\mathrm{PLN}}}$ terminates on $\mathbf{PLN}^{\exists *}_c$.

We now know termination and completeness for each of the fragments $\mathbf{PLN}^{no\exists}_c$, $\mathbf{PLN}^{cl\exists}_c$ and $\mathbf{PLN}^{\exists *}_c$. We also know $\mathcal{T}_{\mathrm{PLN}}$ does not terminate on $\mathbf{PLN}_c$. The only question remaining is whether $\mathcal{T}_{\mathrm{PLN}}$ is complete for $\mathbf{PLN}_c$. The answer is **yes** and we will spend the rest of this section proving this completeness result. The information is summarized in Table 6.1.

We say a set C of sets of formulas is a **consistency class** if it satisfies the following five properties:

$\mathcal{T}_{\mathbf{PLN}}$	terminates?	complete?
$\mathbf{PLN}_c^{no\exists}$	Yes	Yes
$\mathbf{PLN}_c^{cl\exists}$	Yes	Yes
$\mathbf{PLN}_c^{\exists *}$	Yes	Yes
$\mathbf{PLN}_c$	No	Yes

Table 6.1: Termination and completeness results for $\mathcal{T}_{\mathbf{PLN}}$

δ_c If $A \in C$, then $s \notin A$ or $\neg s \notin A$.

$\delta_\wedge$ If $A \in C$ and $s^1 \wedge s^2 \in A$, then $A, s^1, s^2 \in C$.

$\delta_\vee$ If $A \in C$ and $s^1 \vee s^2 \in A$, then $A, s^1 \in C$ or $A, s^2 \in C$.

$\delta_\forall$ If $A \in C$ and $\forall x.s \in A$, then $A, s^x_a \in C$ for all $a \in \mathcal{P}(A) \cup \{a^0\}$.

$\delta_\exists$ If $A \in C$ and $\exists x.s \in A$, then $A, s^x_a \in C$ for some $a \in \mathcal{P}$.

We say C is **subset closed** if $A \in C$ whenever $A \subseteq A'$ and $A' \in C$.

Lemma 6.6.4 (Extension Lemma) Let C be a subset closed consistency class. For any $A \in C$ with $A \subseteq \mathbf{PLN}_c$ there is a set $H \subseteq \mathbf{PLN}_c$ such that $A \subseteq H$ and H satisfies the Hintikka properties ∇_c, $\nabla_\vee$, $\nabla_\wedge$, $\nabla_\forall$ and $\nabla_\exists$.

Proof Suppose $A \in C$ and $A \subseteq \mathbf{PLN}_c$. Enumerate all formulas in $\mathbf{PLN}_c$:

$$t_0, t_1, t_2, t_3, \ldots$$

Let $A_0 = A \in C$. For each n we define $A_{n+1} \in C$ as follows:

1. If $A_n, t_n \notin C$, then let $A_{n+1} = A_n$.
2. If $A_n, t_n \in C$ and t_n is not of the form $\exists x.s$, then let $A_{n+1} = A_n, t_n$.
3. If $A_n, t_n \in C$ and t_n is $\exists x.s$, then by $\delta_\exists$ there is some $a \in \mathcal{P}$ such that $A_n, t_n, s^x_a \in C$. In this case, let $A_{n+1} = A_n, t_n, s^x_a$.

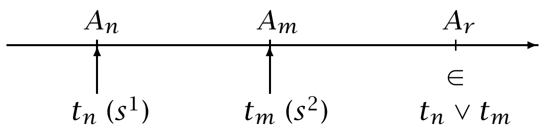
We have ensured that $A_n \in C$ and $A_n \subseteq \mathbf{PLN}_c$ for each n and that we have the inclusions

$$A_0 \subseteq A_1 \subseteq A_2 \subseteq \cdots \subseteq A_n \subseteq \cdots$$

Let $H = \bigcup_n A_n \subseteq \mathbf{PLN}_c$. Clearly $A \subseteq H$. We now easily verify that H satisfies the Hintikka properties ∇_c, $\nabla_\vee$, $\nabla_\wedge$, $\nabla_\forall$ and $\nabla_\exists$.

∇_c Suppose $s, \neg s \in H$. There must be some n such that $s, \neg s \in A_n$ contradicting $A_n \in C$ and δ_c.

$\nabla_\vee$ Suppose $s^1 \vee s^2 \in H$. There exist n, m such that $s^1 = t_n$ and $s^2 = t_m$. Let $r > \max(n, m)$ be big enough that $t_n \vee t_m \in A_r$. We can visualize the situtation as follows:



By $\delta_\vee$ either $A_r, t_n \in C$ or $A_r, t_m \in C$. By subset closure either $A_n, t_n \in C$ or $A_m, t_m \in C$. Thus $t_n \in A_{n+1} \subseteq H$ or $t_m \in A_{m+1} \subseteq H$.

$\nabla_\wedge$ Suppose $t_n \wedge t_m \in H$. Let $r > \max(n, m)$ be such that $t_n \wedge t_m \in A_r$. By $\delta_\wedge$ we know $A_r, t_n, t_m \in C$. By subset closure $A_n, t_n \in C$ and $A_m, t_m \in C$. Hence $t_n \in H$ and $t_m \in H$.

$\nabla_\forall$ Suppose $\forall x.s \in H$ and $a \in \mathcal{P}(H) \cup \{a^0\}$. There is some n such that $t_n = s^x_a$. Let $r > n$ be big enough that $\forall x.s \in A_r$ and $a \in \mathcal{P}(A_r) \cup \{a^0\}$. By $\delta_\forall$ we know $A_r, s^x_a \in C$. By subset closure $A_n, t_n \in C$. We conclude that $t_n \in H$. That is, $s^x_a \in H$.

$\nabla_\exists$ Suppose $\exists x.s \in H$. There is some n such that $t_n = \exists x.s$. Let $r > n$ be big enough that $\exists x.s \in A_r$. By subset closure $A_n, t_n \in C$. By definition of A_{n+1} there is a parameter $a \in \mathcal{P}$ such that $s^x_a \in A_{n+1} \subseteq H$. ∎

Any consistency class can be extended to a subset closed consistency class. We leave this as an exercise.

Exercise 6.6.5 (Subset Closure) Let C be a consistency class. Define

$$C^+ = \{A | \exists A' \in C. A \subseteq A'\}.$$

Prove C^+ is a subset closed consistency class.

The set $C_{\mathbf{PLN}}$ of all A that cannot be refuted using $\mathcal{T}_{\mathbf{PLN}}$ form a consistency class. We define $C_{\mathbf{PLN}}$ to be such that $A \in C_{\mathbf{PLN}}$ iff A is a finite set with $A \subseteq \mathbf{PLN}_c$ such that there is no refutation $A \vdash \perp$ using $\mathcal{T}_{\mathbf{PLN}}$-rules. In terms of tableaux $A \in C_{\mathbf{PLN}}$ if it is not possible to construct a complete tableau starting with a single branch with formulas from A. We leave the verification that $C_{\mathbf{PLN}}$ is a consistency class as an exercise.

Exercise 6.6.6 Prove $C_{\mathbf{PLN}}$ is a consistency class.

Using the exercise above we have a subset closed consistency class $C^+_{\mathbf{PLN}}$ such that $C_{\mathbf{PLN}} \subseteq C^+_{\mathbf{PLN}}$. We can now prove completeness.

Theorem 6.6.7 (Completeness) $\mathcal{T}_{\mathbf{PLN}}$ is complete for $\mathbf{PLN}_c$.

Proof Suppose there is no $\mathcal{T}_{\mathbf{PLN}}$-refutation of A. Then $A \in C_{\mathbf{PLN}} \subseteq C^+_{\mathbf{PLN}}$. By Lemma 6.6.4 there is a set $H \subseteq \mathbf{PLN}_c$ with $A \subseteq H$ such that H satisfies the Hintikka properties ∇_c, $\nabla_\vee$, $\nabla_\wedge$, $\nabla_\forall$ and $\nabla_\exists$. By the Model Existence Theorem (Theorem 6.6.1) there is a logical interpretation $\mathcal{I}$ such that $\mathcal{I} \models s$ for all $s \in H$. In particular, $\mathcal{I} \models s$ for all $s \in A$. ∎

6.7 Cantor's Theorem

Cantor's Theorem states that a set X is always smaller than its power set $\wp(X)$. The powerset $\wp(X)$ is the set of all subsets of X.

How can we express the idea that X is smaller than another set Y? Clearly X and Y are the same size if we can find a bijective function from X to Y. This would mean the elements of X and Y are in one-to-one correspondence with each other. There are two ways there could fail to be a bijection from X to Y. Either Y does not have enough elements and so any function $g : X \to Y$ must eventually start repeating values in Y (i.e., g will not be an injection) or Y has too many elements and any function $g : X \to Y$ must miss some elements of Y. We can express the idea that X is smaller than Y by saying there is no surjection from X to Y.

How can we express the idea that there is no surjection from X to Y in the language? If we have types σ, τ such that $\mathcal{I}\sigma = X$ and $\mathcal{I}\tau = Y$, then we can write

$$\neg\exists g.\forall v.\exists u.gu = v$$

where g is a name of type $\sigma\tau$, v is a name of type τ and u is a name of type σ.

For Cantor's Theorem, if $\mathcal{I}\sigma = X$, then $\mathcal{I}(\sigma B)$ is isomorphic to the power set $\wp(X)$. Therefore, Cantor's Theorem is equivalent to saying there is no surjection from $\mathcal{I}\sigma$ onto $\mathcal{I}(\sigma B)$. This motivates the following version of Cantor's Theorem (for each type σ):

$$\neg\exists g.\forall f.\exists u.gu = f$$

where g is a name of type $\sigma\sigma B$, f is a name of type σB and u is a name of type σ. That is, there is no surjection from σ onto σB.

Informally, we can prove Cantor's Theorem as follows. Assume there is a surjection $G : X \to \wp(X)$. Consider the set

$$D = \{a \in X | a \notin (Ga)\}.$$

Since $D \in \wp(X)$ and G is surjective, there must be some $d \in X$ such that $G(d) = D$. Is $d \in D$? If $d \in D$, then $d \in G(d)$ and so $d \notin D$ (by the definition of D). On the other hand, if $d \notin D$, then $d \notin G(d)$ and so $d \in D$ (by the definition of D). That is, $d \in D$ iff $d \notin D$. This is clearly impossible. Therefore, no such surjection G can exist.

Can we prove this using the tableau rules introduced so far? We can start forming a tableau for Cantor's theorem by writing the negation of the theorem, then applying **DNeg** and **Exists**.

$$\begin{array}{c} \neg\neg\exists g.\forall f.\exists u.gu = f \\ \exists g.\forall f.\exists u.gu = f \\ \forall f.\exists u.gu = f \end{array}$$

$$\neg\neg\exists g.\forall f.\exists u.gu = f$$
$$\exists g.\forall f.\exists u.gu = f$$
$$\forall f.\exists u.gu = f$$
$$\exists u.gu = \lambda x.\neg gxx$$
$$gd = \lambda x.\neg gxx$$

Figure 6.17: Partial tableau proof for Cantor

The next apparent step is to use **Forall** with a term of type σB. What is a reasonable term to use? The key step in proving the set theory version of Cantor's Theorem was introducing the set

$$D = \{a \in X | a \notin (ga)\}.$$

In the current context, we would have $a \in \mathcal{I}\sigma$, $g \in \mathcal{I}(\sigma\sigma B)$, and $ga \in \mathcal{I}(\sigma B)$. Let us use g both as a name of type $\sigma\sigma B$ and its intepretation $\mathcal{I}g$ for the moment. Instead of writing $a \notin (ga)$, we need to write $\neg(gaa)$. Also, instead of forming a set, we form a function in $\mathcal{I}(\sigma B)$. The relevant function is the interpretation of

$$\lambda x.\neg gxx.$$

We apply **Forall** with this term to add

$$\exists u.gu = \lambda x.\neg gxx$$

to the branch. We can now apply **Exists** with a name d of type σ to add

$$gd = \lambda x.\neg gxx$$

to the branch.

Using the tableau rules given so far, we have formed the tableau shown in Figure 6.17. Is there any way to make progress? No. We need new rules to make progress.

The first rule is **Functional**$=$ which will allow us to treat a functional equation $f =_{\sigma\tau} g$ as $\forall x.fx =_\tau gx$. Another rule **Functional**$\neq$ will allow us to treat functional disequation $f \neq_{\sigma\tau} g$ as $\exists x.fx \neq_\tau gx$. These are shown in Figure 6.18.

We can make progress by applying **Functional**$=$ with some term t to add

$$gdt =_B (\lambda x.\neg gxx)t$$

to the branch. Recall that in the proof of Cantor's Theorem we asked whether $d \in D$. In the current context, this corresponds to asking whether or not

$$(\lambda x.\neg gxx)d$$

$$\textbf{Functional=}\,\frac{s =_{\sigma\tau} u}{st = ut} \text{ for any term } t \text{ of type } \sigma$$

$$\textbf{Functional}{\neq}\,\frac{s \neq_{\sigma\tau} u}{sa \neq ua} \text{ where } a \text{ is a fresh name of type } \sigma$$

Figure 6.18: Rules for equality at function types

$$\begin{array}{c}
\neg\neg\exists g.\forall f.\exists u.gu = f \\
\exists g.\forall f.\exists u.gu = f \\
\forall f.\exists u.gu = f \\
\exists u.gu = \lambda x.\neg gxx \\
gd = \lambda x.\neg gxx \\
gdd = (\lambda x.\neg gxx)d \\
\hline
\begin{array}{c|c}
gdd & \neg gdd \\
(\lambda x.\neg gxx)d & \neg(\lambda x.\neg gxx)d
\end{array}
\end{array}$$

Figure 6.19: Second partial tableau proof for Cantor

$$\textbf{Lambda}\,\frac{s}{s'} \text{ where } s \sim_\lambda s'$$

Figure 6.20: Lambda tableau rule

is true. This motivates applying **Functional=** with the term d to add

$$gdd =_B (\lambda x.\neg gxx)d$$

to the branch. Since we have an equation at type B, we can apply **Boolean=** and split the branches into two branches giving the tableau in Figure 6.19.

Neither of the two branches is closed, but both of the branches would be closed if we could simply β-reduce $(\lambda x.\neg gxx)d$ to be $\neg gdd$. This motivates adding the **Lambda** rule shown in Figure 6.20.

Using the **Lambda** rule we can close the two branches and we have the tableau proof for Cantor's Theorem shown in Figure 6.21.

$$
\begin{array}{c}
\neg\neg\exists g.\forall f.\exists u.gu = f \\
\exists g.\forall f.\exists u.gu = f \\
\forall f.\exists u.gu = f \\
\exists u.gu = \lambda x.\neg gxx \\
gd = \lambda x.\neg gxx \\
gdd = (\lambda x.\neg gxx)d \\
\hline
\begin{array}{c|c}
gdd & \neg gdd \\
(\lambda x.\neg gxx)d & \neg(\lambda x.\neg gxx)d \\
\neg gdd & \neg\neg gdd
\end{array}
\end{array}
$$

Figure 6.21: Tableau proof for Cantor

6.7.1 Russell's Law/Turing's Law

How would you answer the following questions?

1. On a small island, can there be a barber who shaves everyone who doesn't shave himself?
2. Does there exist a Turing machine that halts on the representation of a Turing machine x if and only if x does not halt on the representation of x?
3. Does there exist a set that contains a set x as element if and only if $x \notin x$?

The answer to all 3 questions is no, and the reason is purely logical. It is the same idea underlying the proof of Cantor's theorem.

Russell's or Turing's Law: Let $f : IIB$ be a name and $x, y : I$ be names. We can prove $\neg\exists f.\exists x.\forall y.fxy \equiv \neg fyy$.

$$
\begin{array}{c}
\neg\neg(\exists f.\exists x.\forall y.fxy \equiv \neg fyy) \\
\exists f.\exists x.\forall y.fxy \equiv \neg fyy \\
\exists x.\forall y.fxy \equiv \neg fyy \\
\forall y.fxy \equiv \neg fyy \\
fxx \equiv \neg fxx \\
\hline
\begin{array}{c|c}
fxx & \neg fxx \\
\neg fxx & \neg\neg fxx
\end{array}
\end{array}
$$

6.7.2 More about the Lambda rule

Adding the **Lambda** rule opens many new possibilities. A very simple example we can now prove $\exists f.\forall x.fx = x$ where $f : II$ is a name and $x : I$ is name. The following tableau proves this formula:

$$\begin{array}{c} \neg\exists f.\forall x.fx = x \\ \forall f.\neg\forall x.fx = x \\ \neg\forall x.(\lambda x.x)x = x \\ \neg\forall x.x = x \\ \exists x.x \neq x \\ x \neq x \end{array}$$

6.8 Equality

We are still missing a few rules. In particular we need rewriting rules so that we can use equations to replace subterms. Suppose we are attempting to refute

$$A = \{a = b, b = c, a \neq c\}$$

where a, b, c are names of the same sort $\alpha \neq B$. At the moment no rule can be applied to extend the tableau

$$\begin{array}{c} a = b \\ b = c \\ a \neq c \end{array}$$

We remedy this gap by adding the two tableau rules in Figure 6.22 for rewriting. For convenience we allow these rules to be applied when the equation is preceded by a list of universal quantifiers, and we allow the variables to be instantiated before the replacement is made. Using the rewriting rules it is easy to complete the tableau above in several different ways:

1. We could use **Apply=** on $a = b$ and $a \neq c$ with context $[] \neq c$ to add $b \neq c$ to the branch.
2. We could use **Apply=** on $b = c$ and $a \neq c$ with context $a \neq []$ to add $a \neq b$ to the branch.
3. We could use **Apply=** on $a = b$ and $b = c$ with context $[] = c$ to add $a \neq c$ to the branch.
4. We could use **Apply=** on $b = c$ and $a = b$ with context $a = []$ to add $a \neq c$ to the branch.

Any of these applications will close the branch.

To see an easy example in which we instantiate quantified variables, consider the formula

$$(\forall xy.fxy = gxy) \to fy = gy$$

$$\textbf{Apply=}\frac{\forall\overline{x^n}.s = t, C[\theta s]}{C[\theta t]} \qquad \textbf{Apply=}\frac{\forall\overline{x^n}.s = t, C[\theta t]}{C[\theta s]}$$

where for both versions of the rule $\theta\ y\ =\ y$ if $y \notin\ \overline{x^n}$

and $C[]$ is admissible for $\{\forall\overline{x^n}.s = t\}$

Figure 6.22: Tableau rewriting rules

where $x, y : I$ are names and $f, g : III$ are names. A complete tableau for this is given by

$$\begin{array}{c} \neg(\forall xy.fxy = gxy) \to fy = gy \\ (\forall xy.fxy = gxy) \\ fy \neq gy \\ (\lambda z.fyz) \neq \lambda z.gyz \\ (\lambda z.fyz) \neq \lambda z.fyz \end{array}$$

Note that in one step we used **Lambda** to η-expand $\neg fy = gy$ into $\neg(\lambda z.fyz) = \lambda z.gyz$. In another step we have used **Apply=** with context $(\lambda z.fyz) \neq \lambda z.[]$ and θ such that $\theta(x) = y$ and $\theta(y) = z$.

6.9 Excluded Middle as a Rule

Suppose we are trying to refute the set

$$A = \{pa, \neg pb, a, b\}$$

where $p : BB$ is a name and $a, b : B$ are names. Is this set satisfiable? No, it is not. We are forced to interpret a and b such that $\mathcal{I}a = 1 = \mathcal{I}b$. This means $\mathcal{I}(pa)$ must equal $\mathcal{I}(pb)$. On the other hand, none of the tableau rules given so far can be applied to

$$\begin{array}{c} pa \\ \neg pb \\ a \\ b \end{array}$$

We can remedy this problem by adding the refutation rules (both called **XM**) in Figure 6.23. The corresponding tableau views are shown in Figure 6.24.

The **XM** rule allows us to introduce an arbitrary disjunction of the form $s \vee \neg s$ or $\neg s \vee s$ onto a branch. Combining this with the **Or** rule, we can split any branch

$$\textbf{XM}\ \frac{A, s \lor \neg s \vdash \bot}{A \vdash \bot} \qquad \textbf{XM}\ \frac{A, \neg s \lor s \vdash \bot}{A \vdash \bot}$$

Figure 6.23: Extra refutation rules

$$\textbf{XM}\frac{\qquad}{s \lor \neg s} \qquad \textbf{XM}\frac{\qquad}{\neg s \lor s}$$

Figure 6.24: Tableau views of extra refutation rules

into two branches where in one branch we know s holds and in the other we know $\neg s$ holds. Often this is useful if we know s must be true on the branch (so that the branch with $\neg s$ will be easy to close) and we believe s will be useful.

Using the **XM** rule, we can obtain the following complete tableau refuting the set A above.

$$\begin{array}{c}
pa \\
\neg pb \\
a \\
b \\
a = b \lor a \neq b \\
\hline
\begin{array}{c|c}
\begin{array}{c} a = b \\ pb \end{array} &
\begin{array}{c} a \neq b \\ \hline \begin{array}{c|c} a & b \\ \neg b & \neg a \end{array} \end{array}
\end{array}
\end{array}$$

On the left branch we have used **Apply=** and on the right branch we have used **Boolean≠** to split into two (closed) branches.

Note that this rule is fundamentally different from the tableau rules we have shown so far. The new formula $s \lor \neg s$ need not have any relationship to a formula already on the branch.

6.10 Examples

6.10.1 Expressing Universal Quantification in terms of Existential

We know we can express the universal quantifier in terms of the existential quantifier using De Morgan's law. Let all be $\lambda f.\neg\exists x.\neg f x$. We can easily prove

$$\forall f.\mathsf{all} f = \forall x.f x$$

as a tableau. (Note that the steps where we expand the definition of all are actually applications of **Lambda**.)

$$\begin{array}{c|c}
\multicolumn{2}{c}{\neg\forall f.\mathsf{all}f = \forall x.fx}\\
\multicolumn{2}{c}{\exists f.\mathsf{all}f \neq \forall x.fx}\\
\multicolumn{2}{c}{\mathsf{all}f \neq \forall x.fx}\\
\hline
 & \forall x.fx\\
\mathsf{all}f & \neg\mathsf{all}f\\
\neg\forall x.fx & \neg\neg\exists x.\neg fx\\
\exists x.\neg fx & \exists x.\neg fx\\
\neg\exists x.\neg fx & \neg fx\\
 & fx
\end{array}$$

6.10.2 Expressing Equality using Higher-Order Quantifiers

We have seen that equality can be expressed using Leibniz' Law:

$$\lambda xy.\forall f.fx \to fy$$

Here we assume $x, y : \sigma$ and $f : \sigma B$ are variables. We can use the same idea to express disequality. Let neq be $\lambda xy.\exists f.fx \wedge \neg fy$. We can easily prove

$$\forall xy.\mathsf{neq}xy \equiv x \neq y$$

with the following tableau.

$$\begin{array}{c|c}
\multicolumn{2}{c}{\neg\forall xy.\mathsf{neq}xy \equiv x \neq y)}\\
\multicolumn{2}{c}{\exists x.\neg\forall y.\mathsf{neq}xy \equiv x \neq y}\\
\multicolumn{2}{c}{\neg\forall y.\mathsf{neq}xy \equiv x \neq y)}\\
\multicolumn{2}{c}{\exists y.\neg(\mathsf{neq}xy \equiv x \neq y)}\\
\multicolumn{2}{c}{\neg(\mathsf{neq}xy \equiv x \neq y)}\\
\hline
\mathsf{neq}xy & x \neq y\\
\neg x \neq y & \neg\mathsf{neq}xy\\
x = y & \neg\exists f.fx \wedge \neg fy\\
\exists f.fx \wedge \neg fy & \forall f.\neg(fx \wedge \neg fy)\\
fx \wedge \neg fy & \neg((\lambda y.x = y)x \wedge \neg(\lambda y.x = y)y)\\
fx & \neg(x = x \wedge x \neq y)\\
\neg fy & x \neq x \vee \neg x \neq y\\
\neg fx & \begin{array}{c|c}\hline x \neq x & \neg x \neq y\end{array}
\end{array}$$

We can also express equality as the least reflexive relation. Let eq be $\lambda xy.\forall r.(\forall z.rzz) \to rxy$ where $r : \sigma\sigma B$ is a variable. We prove

$$\forall xy.\mathsf{eq}xy = (x = y)$$

with a tableau.

$$\begin{array}{c}
\neg\forall xy.\mathsf{eq}xy = (x = y) \\
\exists x.\neg\forall y.\mathsf{eq}xy = (x = y) \\
\neg\forall y.\mathsf{eq}xy = (x = y) \\
\exists y.\mathsf{eq}xy \neq (x = y) \\
\mathsf{eq}xy \neq (x = y) \\
\hline
\begin{array}{c|c}
\begin{array}{c}
\mathsf{eq}xy \\
x \neq y \\
\forall r.(\forall z.rzz) \to rxy \\
(\forall z.(\lambda xy.x = y)zz) \to (\lambda xy.x = y)xy \\
(\forall z.z = z) \to x = y \\
\hline
\begin{array}{c|c}
x = y &
\begin{array}{c}
\neg\forall z.z = z \\
\exists z.z \neq z \\
z \neq z
\end{array}
\end{array}
\end{array}
&
\begin{array}{c}
x = y \\
\neg\mathsf{eq}xy \\
\neg\forall r.(\forall z.rzz) \to rxy \\
\exists r.\neg((\forall z.rzz) \to rxy) \\
\neg((\forall z.rzz) \to rxy) \\
\forall z.rzz \\
\neg rxy \\
rxx \\
\neg rxx
\end{array}
\end{array}
\end{array}$$

6.10.3 Solving Easy Equations

Sometimes existential theorems can be proven by taking the body as giving a definition. It is important to recognize when theorems are this easy to prove! Here are two examples:

We can think of the type σB as the type of sets of elements of type σ. The membership relation should have type $\sigma(\sigma B)B$ in this context. Let $\mathsf{in} : \sigma(\sigma B)B$, $x : \sigma$ and $f : \sigma B$ be variables. We can prove the existential theorem

$$\exists \mathsf{in}.\forall xf.\mathsf{in}xf = fx$$

by using the term

$$\lambda xf.fx$$

when we apply the **Forall** rule to the quantifier binding in in the following tableau.

$$\begin{array}{c}
\neg\exists \mathsf{in}.\forall xf.\mathsf{in}xf = fx \\
\forall \mathsf{in}.\neg\forall xf.\mathsf{in}xf = fx \\
\neg\forall xf.(\lambda xf.fx)xf = fx \\
\neg\forall xf.fx = fx \\
\exists x.\neg\forall f.fx = fx \\
\neg\forall f.fx = fx \\
\exists f.fx \neq fx \\
fx \neq fx
\end{array}$$

We can similarly prove the existence of a set difference function. Let $\mathsf{diff} : (\sigma B)(\sigma B)\sigma B$, $f, g : \sigma B$ and $x : \sigma$ be variables. We prove

$$\exists \mathsf{diff} : (XB)(XB)XB.\forall fgx.\mathsf{diff}\,fgx = (fx \land \neg gx)$$

with the following tableau.

$$\begin{array}{c}
\neg\exists \mathsf{diff}.\forall fgx.\mathsf{diff}\,fgx = (fx \land \neg gx) \\
\forall \mathsf{diff}.\neg\forall fgx.\mathsf{diff}\,fgx = (fx \land \neg gx) \\
\neg\forall fgx.(\lambda fgx.fx \land \neg gx)fgx = (fx \land \neg gx) \\
\neg\forall fgx.(fx \land \neg gx) = (fx \land \neg gx) \\
\exists f.\neg\forall gx.(fx \land \neg gx) = (fx \land \neg gx) \\
\neg\forall gx.(fx \land \neg gx) = (fx \land \neg gx) \\
\exists g.\neg\forall x.(fx \land \neg gx) = (fx \land \neg gx) \\
\neg\forall x.(fx \land \neg gx) = (fx \land \neg gx) \\
\exists x.(fx \land \neg gx) \neq (fx \land \neg gx) \\
(fx \land \neg gx) \neq (fx \land \neg gx)
\end{array}$$

6.10.4 Expressing Conjunction with → and ⊥

Let $x, y : B$ be variables. Let $\mathsf{neg} : BB$ be $\lambda x.x \to \bot$. Let $\mathsf{and} : BBB$ be $\lambda xy.\mathsf{neg}(x \to \mathsf{neg}\,y)$. We prove $\forall xy.\mathsf{and}\,xy = (x \land y)$.

$$\begin{array}{c}
\neg\forall xy.\mathsf{and}\,xy = (x \land y) \\
\exists x.\neg\forall y.\mathsf{and}\,xy = (x \land y) \\
\neg\forall y.\mathsf{and}\,xy = (x \land y) \\
\exists y.\mathsf{and}\,xy \neq (x \land y) \\
\mathsf{and}\,xy \neq (x \land y) \\
\hline
\begin{array}{c|c}
\begin{array}{c}
\mathsf{and}\,xy \\
\neg(x \land y) \\
\neg x \lor \neg y \\
\mathsf{neg}(x \to \mathsf{neg}\,y) \\
(x \to \mathsf{neg}\,y) \to \bot \\
\hline
\begin{array}{c|c}
\bot &
\begin{array}{c}
\neg(x \to \mathsf{neg}\,y) \\
x \\
\neg\mathsf{neg}\,y \\
\hline
\begin{array}{c|c}
\neg x &
\begin{array}{c}
\neg y \\
\neg(y \to \bot) \\
y \\
\neg\bot
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
&
\begin{array}{c}
x \land y \\
\neg\mathsf{and}\,xy \\
x \\
y \\
\neg\mathsf{neg}(x \to \mathsf{neg}\,y) \\
\neg((x \to \mathsf{neg}\,y) \to \bot) \\
x \to \mathsf{neg}\,y \\
\neg\bot \\
\hline
\begin{array}{c|c}
\begin{array}{c}
\mathsf{neg}\,y \\
y \to \bot \\
\hline
\begin{array}{c|c}
\bot & \neg y
\end{array}
\end{array}
& \neg x
\end{array}
\end{array}
\end{array}
\end{array}$$

6.10.5 Expressing Conjunction with Higher-Order Quantification

Let $x, y : B$ and $g : BBB$ be variables. Let $\mathsf{and} : BBB$ be $\lambda xy.\forall g.gxy = g\top\top$.

We will prove $\forall xy.\mathsf{and}xy = (x \wedge y)$. Proving this with one big tableau is possible, but let's split it into two lemmas.

First we prove $\forall xy.\mathsf{and}xy \to x \wedge y$.

$$\begin{array}{c}
\neg\forall xy.\mathsf{and}xy \to x \wedge y \\
\exists x.\neg\forall y.\mathsf{and}xy \to x \wedge y \\
\neg\forall y.\mathsf{and}xy \to x \wedge y \\
\exists y.\neg(\mathsf{and}xy \to x \wedge y) \\
\neg(\mathsf{and}xy \to x \wedge y) \\
\mathsf{and}xy \\
\neg(x \wedge y) \\
\neg x \vee \neg y \\
\forall g.gxy = g\top\top \\
(\lambda xy.x \wedge y)xy = (\lambda xy.x \wedge y)\top\top \\
(x \wedge y) = (\top \wedge \top) \\
\hline
\begin{array}{c|c}
 & \neg(x \wedge y) \\
x \wedge y & \neg(\top \wedge \top) \\
\top \wedge \top & \neg\top \vee \neg\top \\
\cline{2-2}
 & \begin{array}{c|c} \neg\top & \neg\top \end{array}
\end{array}
\end{array}$$

Next we prove $\forall xy.x \land y \rightarrow \mathsf{and}xy$.

$$\begin{array}{c}
\neg\forall xy.x \land y \rightarrow \mathsf{and}xy \\
\exists x.\neg\forall y.x \land y \rightarrow \mathsf{and}xy \\
\neg\forall y.x \land y \rightarrow \mathsf{and}xy \\
\exists y.\neg(x \land y \rightarrow \mathsf{and}xy) \\
\neg(x \land y \rightarrow \mathsf{and}xy) \\
x \land y \\
\neg\mathsf{and}xy \\
\neg\forall g.gxy = g\top\top \\
\exists g.gxy \neq g\top\top \\
gxy \neq g\top\top \\
x \\
y \\
x = \top \lor x \neq \top \\
\hline
\begin{array}{c|c}
\begin{array}{c}
x = \top \\
g\top y \neq g\top\top \\
y = \top \lor y \neq \top \\
\hline
\begin{array}{c|c}
\begin{array}{c} y = \top \\ g\top\top \neq g\top\top \end{array} &
\begin{array}{c} y \neq \top \\ \hline \begin{array}{c|c} y & \top \\ \neg\top & \neg y \end{array} \end{array}
\end{array}
\end{array} &
\begin{array}{c}
x \neq \top \\
\hline
\begin{array}{c|c} x & \top \\ \neg\top & \neg x \end{array}
\end{array}
\end{array}
\end{array}$$

Finally we prove $\forall xy.\mathsf{and}xy = (x \land y)$ using these two lemmas.

$$\begin{array}{c}
\neg\forall xy.\mathsf{and}xy = (x \land y) \\
\forall xy.\mathsf{and}xy \rightarrow x \land y \\
\forall xy.x \land y \rightarrow \mathsf{and}xy \\
\exists x.\neg\forall y.\mathsf{and}xy = (x \land y) \\
\neg\forall y.\mathsf{and}xy = (x \land y) \\
\exists y.\mathsf{and}xy \neq (x \land y) \\
\mathsf{and}xy \neq (x \land y) \\
\hline
\begin{array}{c|c}
\begin{array}{c}
\mathsf{and}xy \\
\neg(x \land y) \\
\forall y.\mathsf{and}xy \rightarrow x \land y \\
\mathsf{and}xy \rightarrow x \land y \\
\hline
\begin{array}{c|c} x \land y & \neg\mathsf{and}xy \end{array}
\end{array} &
\begin{array}{c}
x \land y \\
\neg\mathsf{and}xy \\
\forall y.x \land y \rightarrow \mathsf{and}xy \\
x \land y \rightarrow \mathsf{and}xy \\
\hline
\begin{array}{c|c} \mathsf{and}xy & \neg(x \land y) \end{array}
\end{array}
\end{array}
\end{array}$$

6.10.6 Kaminski Equation

Let $f : BB$ and $x : B$ be names. We can prove the following equation

$$f(f(fx)) = fx$$

which we call the Kaminski Equation. A special case of the Kaminski Equation is

$$f(f(f\bot)) = f\bot$$

This can be proven by considering whether certain subterms are equal to $\top$ or $\bot$. For instance, what if $f\bot = \bot$? In this case we clearly have $f(f(f\bot)) = \bot = f\bot$. What if $f\bot = \top$? In this case we have $f(f(f\bot)) = f(f\top)$, but does $f(f\top) = \top$? A further case analysis on the value of $f\top$ (assuming $f\bot = \top$) can be used to prove $f(f\top) = \top$, verifying the equation.

Many proofs of the Kaminski Equation are possible.

Exercise 6.10.1 Prove the Kaminski Equation informally and then give a tableau proof based on your informal proof.

6.10.7 Choice

If C is a choice function at σ, then $C(\lambda y.x = y) = x$ for all x of type σ.

$$\begin{array}{c}
\forall p.(\exists x.px) \to p(Cp) \\
\neg\forall x.C(\lambda y.x = y) = x \\
\exists x.C(\lambda y.x = y) \neq x \\
C(\lambda y.x = y) \neq x \\
(\exists z.(\lambda y.x = y)z) \to (\lambda y.x = y)(C\lambda y.x = y) \\
(\exists z.x = z) \to x = C\lambda y.x = y \\
\hline
\begin{array}{c|c}
x = C\lambda y.x = y & \begin{array}{c} \neg\exists z.x = z \\ \forall z.x \neq z \\ x \neq x \end{array}
\end{array}
\end{array}$$

If we assume C is a choice function at type τ, then we can prove the Skolem principle

$$((\forall x.\exists y.rxy) \to \exists f.\forall x.rx(fx))$$

where r has type $\sigma\tau B$.

$$\begin{array}{c}
\forall p.(\exists x.px) \to p(Cp) \\
\neg((\forall x.\exists y.rxy) \to \exists f.\forall x.rx(fx)) \\
\forall x.\exists y.rxy \\
\neg\exists f.\forall x.rx(fx) \\
\forall f.\neg\forall x.rx(fx) \\
\neg\forall x.rx((\lambda x.C(rx))x) \\
\neg\forall x.rx(C(rx)) \\
\exists x.\neg rx(C(rx)) \\
\neg rx(C(rx)) \\
(\exists z.rxz) \to rx(C(rx)) \\
\hline
\begin{array}{c|c}
 & \neg\exists z.rxz \\
rx(C(rx)) & \exists y.rxy \\
 & \exists z.rxz
\end{array}
\end{array}$$

If we have the Skolem principle

$$((\forall x.\exists y.rxy) \to \exists f.\forall x.rx(fx))$$

where r has type $(\sigma B)\sigma B$, then we can prove the existence of a choice function at type σ. We first prove the following lemma:

$$\forall p.\exists y.(\exists y.py) \to py$$

$$\begin{array}{c}
\neg\forall p.\exists y.(\exists y.py) \to py \\
\exists p.\neg\exists y.(\exists y.py) \to py \\
\neg\exists y.(\exists y.py) \to py \\
\forall y.\neg((\exists y.py) \to py) \\
\neg((\exists y.py) \to pz) \\
\exists y.py \\
\neg pz \\
py \\
\neg((\exists y.py) \to py) \\
\exists y.py \\
\neg py
\end{array}$$

Now we use the lemma to prove the main result.

$$\begin{array}{c}
\forall p.\exists y.(\exists y.py) \to py \\
\neg\exists C.\forall p.(\exists y.py) \to p(Cp) \\
\forall r.(\forall x.\exists y.rxy) \to \exists f.\forall x.rx(fx) \\
(\forall x.\exists y.(\lambda xy.(\exists y.xy) \to xy)xy) \to \exists f.\forall x.(\lambda xy.(\exists y.xy) \to xy)x(fx) \\
(\forall x.\exists y.(\exists y.xy) \to xy) \to \exists f.\forall x.(\exists y.xy) \to x(fx) \\
\hline
\begin{array}{c|c}
\exists f.\forall x.(\exists y.xy) \to x(fx) & \neg\forall x.\exists y.(\exists y.xy) \to xy \\
\exists C.\forall p.(\exists y.py) \to p(Cp) & \neg\forall p.\exists y.(\exists y.py) \to py
\end{array}
\end{array}$$

Note that the last step on each branch is an application of **Lambda** to α-convert the next to the last formula on the branch.

7 A Taste of Set Theory

Earlier we saw how to specify the natural numbers by giving a sort N, names such as $O : N$ and $S : NN$, and formulas to ensure that the interpretation that the sort and names have the intended interpretation. We can repeat a similar process for sets.

We still assume the same situation as the last section. We have the sort I and the disjoint infinite sets $\mathcal{V}$ and $\mathcal{P}$ of names of sort I.

Let us suppose $\mathcal{I}$ is a logical interpretation and $\mathcal{I}I$ some a collection of sets. Let us use $X, Y, Z, A, B, C, D, a, b, c, d$ to range over sets in $\mathcal{I}I$.

The basic relation of set theory is membership. Assume we have a name $\in: IIB$ which we write in infix notation. We write $s \in t$ for the term $\in st$. We will also write $s \notin t$ as notation for the term $\neg(\in st)$. Assume $\in$ and $\notin$ have the same priority as the infix operator $=$.

Another important relation in set theory is the subset relation. Assume we have a term $\subseteq: IIB$ which we also write in infix notation. Assume $\subseteq$ also has the same priority as the infix operator $=$. We can express subset in terms of membership as follows:

$$\forall xy.x \subseteq y \equiv \forall z.z \in x \to z \in y$$

We can also write this as the closed PLN-formula

$$\forall xy.(\neg x \subseteq y \vee \forall z.\neg z \in x \vee z \in y) \wedge (x \subseteq y \vee \exists z.z \in x \wedge \neg z \in y)$$

though this is more difficult to read. From now on we will not restrict ourselves to PLN-formulas, but will make the point when formulas can be expanded into equivalent PLN-formulas. At this point, we could take $\subseteq: IIB$ to be a name and assume

$$\forall xy.x \subseteq y \equiv \forall z.z \in x \to z \in y$$

as an axiom. Since we are not restricting ourselves to PLN-formulas, it also makes sense to define $\subseteq: IIB$ to simply be notation for the term

$$\lambda xy.\forall z.z \in x \to z \in y.$$

We take this option.

Which sets would we like to ensure are in $\mathcal{I}I$? The first set we consider is the **empty set**, denoted $\emptyset$. The empty set has no elements. How can we ensure the empty set is in $\mathcal{I}I$? In fact we can do this with a simple PLN-formula:

- **Empty Set Exists**: $\exists y.\forall z.z \notin y$.

  Next, any time we have a set $X \in \mathcal{U}$ we can form the **power set of X**, denoted $\wp(X)$. The power set of X contains exactly the subsets of X. That is, $Y \in \wp(X)$ if and only if $Y \subseteq X$. We can ensure the existence of power sets as follows:

- **Power Sets Exist**: $\forall x.\exists y.\forall z.z \in y \equiv z \subseteq x$.

This is not a **PLN**-formula, but could be easily expanded into one.

We are now assured $\mathcal{U}$ contains at least $\emptyset$, $\wp(\emptyset)$, $\wp(\wp(\emptyset))$, $\wp(\wp(\wp(\emptyset)))$, etc. Let us think about these sets. How many subsets of $\emptyset$ are there? Clearly, there is only one: $\emptyset$. Hence

$$\wp(\emptyset) = \{\emptyset\}.$$

Now how many subsets of $\{\emptyset\}$ are there? There are two: $\emptyset$ and $\{\emptyset\}$, so

$$\wp(\wp(\emptyset)) = \{\emptyset, \{\emptyset\}\}.$$

Now how many subsets of $\{\emptyset, \{\emptyset\}\}$ are there? There are four:

$$\wp(\wp(\wp(\emptyset))) = \{\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \{\emptyset, \{\emptyset\}\}\}.$$

In general, if a finite set X has n elements, then $\wp(X)$ will have 2^n elements.

Notice that for each X above, $X \subseteq \wp(X)$. Is this always true? No. We say a set X is called **transitive** if for every $Z \in A$ and $A \in X$, we have $Z \in X$. It is an easy exercise to prove $X \subseteq \wp(X)$ if and only if X is transitive. It is also easy to prove that if X is transitive, then $\wp(X)$ is transitive. **Warning:** This use of the word "transitive" is distinct from the notation of a "transitive relation." We are not in any way suggesting the relation $\in$ is transitive!

Exercise 7.0.2 Find a set X such that $X \nsubseteq \wp(X)$.

Exercise 7.0.3 Prove the following three statements are equivalent:

a) X is transitive.

b) For every $A \in X$ we know $A \subseteq X$.

c) $X \subseteq \wp(X)$.

Hint: Prove (a) implies (b); Prove (b) implies (c); Prove (c) implies (a).

Exercise 7.0.4 If X is transitive, then $\wp(X)$ is transitive.

Combining these facts, we conclude the following inclusions:

$$\emptyset \subseteq \wp(\emptyset) \subseteq \wp(\wp(\emptyset)) \subseteq \cdots \subseteq \wp^n(\emptyset) \subseteq \cdots$$

We recursively define sets $V_0 = \emptyset$ and $V_{n+1} = \wp(V_n)$. Since we know the size of V_n grows (quickly!) as n grows we have the more suggestive picture of a "big V" in Figure 7.1.

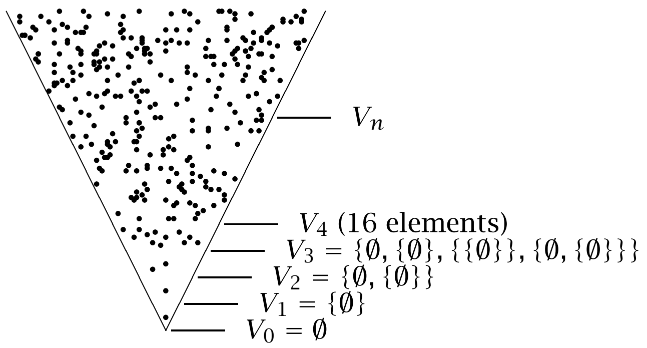

V_n
V_4 (16 elements)
$V_3 = \{\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \{\emptyset, \{\emptyset\}\}\}$
$V_2 = \{\emptyset, \{\emptyset\}\}$
$V_1 = \{\emptyset\}$
$V_0 = \emptyset$

Figure 7.1: Sets as a big V

How could we prove $\wp(\emptyset)$ exists? The first question is how we express the claim that $\wp(\emptyset)$ exists. A first try might be

$$\exists y.\forall z.z \in y \equiv z \subseteq \emptyset$$

but $\emptyset$ is a set, not a name of type I! We can express the conjecture without using a name for the empty set. Here is an example how:

$$\exists y.\forall z.z \in y \equiv \forall w.(\forall u.u \notin w) \to z \subseteq w$$

We will *not* do this! We want names for our objects and constructors!

Suppose $\emptyset : I$ is a name of type I. (Note the overloading. Some occurrences of $\emptyset$ will be a name of type I and others will be the empty set. We already used such overloading for $\in$ and $\subseteq$, without saying so.) To avoid any potential complications, assume $\emptyset \notin \mathcal{V}$. Now we *can* state existence of $\wp(\emptyset)$ as

$$\exists y.\forall z.z \in y \equiv z \subseteq \emptyset$$

but have no hope of proving it. Why not? We only assumed an empty set exists, not that the name $\emptyset$ is the empty set. We remedy this:

- **Empty Set**: $\forall z.y \notin \emptyset$.

This axiom will guarantee that $\mathcal{I}\emptyset = \emptyset$. (That is, the interpretation of the name $\emptyset$ is the empty set.)

We now have a name $\emptyset$ for the empty set. Why do we not have a term of type I corresponding to $\wp(\emptyset)$? The problem is the formulation of the **Power Sets Exist** axiom. In order to make our lives easier, we assume we have a name $\mathsf{Pow} : II$ where $\mathcal{I}\mathsf{Pow}X$ is (or at least should be) the power set $\wp(X)$. We now assume

- **Power Sets**: $\forall xz.z \in (\mathsf{Pow}x) \equiv z \subseteq x$.

Now the fact that $\wp(\emptyset)$ exists is trivial (as it should be!) since $\mathsf{Pow}\emptyset$ is a term such that $\mathcal{I}(\mathsf{Pow}\emptyset) = \wp(\emptyset)$.

We noted that if we had a set X with n elements, $\wp(X)$ had 2^n elements. Clearly n is smaller than 2^n. Is there some way to express the fact that X is a smaller set than $\wp(X)$ within the language? After we express it, can we prove it?

Unfortunately, we are not in such a simple situation. In our situation X and Y do not correspond to types, but to members of $\mathcal{I}I$. We can give a formulation that is almost as simple by taking g to be a name of type II, x, y, u and v to be names of type I, and writing

$$\begin{array}{rl} \neg\exists g. & (\forall u.u \in x \rightarrow gu \in y) \\ \wedge & \forall v.v \in y \rightarrow \exists u.u \in x \wedge gu = v \end{array}$$

If $\mathcal{I}x = X$ and $\mathcal{I}y = Y$, then the formula above will be true iff there is a function from all of $\mathcal{I}I$ to $\mathcal{I}I$ which restricts to be a surjection from X to Y. We take $\mathsf{surj} : II(II)B$ to be notation for the term

$$\begin{array}{rl} \lambda xyg. & (\forall u.u \in x \rightarrow gu \in y) \\ \wedge & \forall v.v \in y \rightarrow \exists u.u \in x \wedge gu = v \end{array}$$

Note that we have made use of a higher-order quantifier ($\exists g$ where g has type II). As soon as we quantify over a type other than I, then we have certainly abandoned **PLN**-formulas. With some work, we can express the idea that there is no surjection without quantifying over types other than I. We will not do this here, but we will sketch the steps.

1. Consider a function as a set of ordered pairs.
2. Define ordered pairs $(a, b) \in \mathcal{I}I$ of $a \in \mathcal{I}I$ and $b \in \mathcal{I}I$ such that

$$(a, b) = (a', b') \Rightarrow a = a' \text{ and } b = b'.$$

   A common choice is to use the **Kuratowski pair** (a, b) of a and b given by

$$\{\{a\}, \{a, b\}\}.$$

3. Define $Z \in \mathcal{I}I$ to be a surjection from X to Y if four conditions hold:
   a) Every element of Z is of the form (a, b) for some $a \in X$ and $b \in Y$.
   b) For every $a \in X$, there is some $b \in Y$ such that $(a, b) \in Z$.
   c) If $(a, b) \in Z$ and $(a, b') \in Z$, then $b = b'$.
   d) For every $b \in Y$, there is some $a \in X$ such that $(a, b) \in Z$.
4. Write these conditions as a formula.

We now return to the concept we wanted to define: when a set x is smaller than y. Now that we have surj, we take $\prec$ (written in infix, with the same precedence as $=$) to be notation for the term

$$\lambda xy.\neg\exists g.\mathsf{surj}xyg$$

of type IIB. We can now easily express the result about the size of the power set as

$$\forall x.x \prec (\mathsf{Pow}x).$$

We will delay thinking about *proving* this result.

So far we have only constructed finite sets. We started with the finite set $\emptyset$. At each stage, we had a finite set V_n of finite sets and formed $V_{n+1} = \wp(V_n)$. Since any subset of a finite set is finite, V_{n+1} only contains finite sets. Also, since a finite set has only finitely many subsets, V_{n+1} is itself finite. How can we form an infinite set?

We can easily define a sequence of sets which resembles the natural numbers. Let 0 be $\emptyset$, 1 be $\{0\}$, 2 be $\{0,1\}$, and so on. In general, we can define $n+1 = \{0,\ldots,n\}$. An easy induction on n proves that $n \notin V_n$, $n \subseteq V_n$ and $n \in V_{n+1}$. Consider the set ω defined to be

$$\{0,1,2,\ldots,n\ldots\}.$$

Clearly ω is an infinite set and is not in any V_n.

We can continue the construction of our set theoretical universe by taking

$$V_\omega = \bigcup_{n\in\omega} V_n.$$

This V_ω is an infinite set. Is $\omega \in V_\omega$? It cannot be. Every member of V_ω is a member of V_n for some n. Hence V_ω only contains finite sets. On the other hand, $\omega \subseteq V_\omega$ and so $\omega \in \wp(V_\omega)$. This motivates continuing our power set construction: For each n, we take $V_{\omega+n+1}$ to be $\wp(V_{\omega+n})$. Clearly $\omega \in V_{\omega+1}$ (just as each n was in V_{n+1}). Note that we still have a chain

$$V_0 \subseteq V_1 \subseteq V_2 \subseteq \cdots \subseteq V_n \subseteq \cdots \subseteq V_\omega \subseteq V_{\omega+1} \subseteq \cdots \subseteq V_{\omega+n} \subseteq \cdots$$

and the even bigger "big V" in Figure 7.2.

Notice that for any $n, m \in \omega$, we have $n < m$ (as integers) iff $n \in m$ (as sets).

Now that we have an infinite set ω, we can ask an interesting question. How does the size of $\wp(\omega)$ compare with the size of ω? Clearly ω is countable. In the case of finite sets we had $X \prec \wp(X)$. Is this also true of infinite sets? Yes. In fact, we gave the informal proof in Chapter 6. We repeat it here.

Theorem 7.0.5 (Cantor's Theorem) For any set X, $X \prec \wp(X)$.

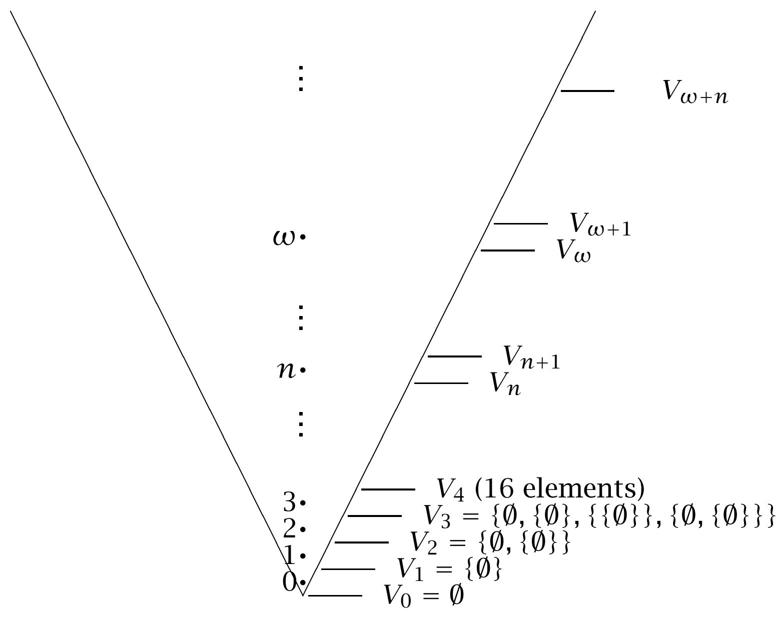


Figure 7.2: Sets, including some infinite sets, as a big V

Proof Assume we have a set X such that $X \prec \wp(X)$ does *not* hold. Then there must be a surjection $G : X \to \wp(X)$. Consider the set

$$D = \{a \in X | a \notin (Ga)\}.$$

Since $D \in \wp(X)$ and G is surjective, there must be some $d \in X$ such that $G(d) = D$. Is $d \in D$? If $d \in D$, then $d \in G(d)$ and so $d \notin D$ (by the definition of D). On the other hand, if $d \notin D$, then $d \notin G(d)$ and so $d \in D$ (by the definition of D). That is, $d \in D$ iff $d \notin D$. This is clearly impossible. Therefore, no such surjection G can exist. ■

The key idea of the proof of Cantor's theorem is the choice of D. In order to mimic the informal proof above in a formal tableau proof, we must be able to make use of the set D. We can construct a partial tableau of the form

$$\neg(x \prec (\mathsf{Pow}x))$$
$$\vdots$$
$$\forall v.v \in (\mathsf{Pow}x) \to \exists u.u \in x \land (gu) = v$$

We then need to somehow apply **Forall** to instantiate v with a term corresponding to the set D. Examine the set D:

$$D = \{a \in X | \neg(a \in (Ga))\}$$

We can imagine writing the body as

$$\neg(y \in (gy))$$

with a variable y, and this body has type B as expected. We can λ-abstract the y and obtain the term

$$\lambda y.\neg(y \in (gy))$$

but this term has type IB, where we need to instantiate v with a term of type I. With the axioms and constructors introduced so far, we cannot represent the set D as a term. What we need to be able to do is take a term of type I representing a set X and a term of type IB representing a property P of sets and form a term of type I corresponding to $\{a \in X | Pa = 1\}$.

Just as we introduced names $\emptyset$ and Pow, we introduce a name sep of type $I(IB)I$ such that for all $X \in \mathcal{I}I$ and $P \in \mathcal{I}(IB)$ we have

$$\mathcal{I}\mathsf{sep}XP = \{a \in X | Pa = 1\}.$$

The corresponding axiom is the **Axiom of Separation**:

- **Separation**: $\forall xpz.z \in (\mathsf{sep}xp) \equiv z \in x \wedge px$

Remark: In first-order set theory, the separation axiom is restricted to properties p defined by first-order formulas.

Cantor's theorem implies that ω is a smaller infinity that $\wp(\omega)$. In fact we have an infinite chain of infinities

$$\omega \prec \wp(\omega) \prec \wp(\wp(\omega)) \prec \cdots.$$

A natural question one could ask is whether there are infinities in between any of these inequalities. In particular, is there a set X such that $\omega \prec X$ and $X \prec \wp(\omega)$? Cantor's Continuum Hypothesis states that there are no such sets X. The Continuum Hypothesis is often stated in the following form:

Continuum Hypothesis (CH) Every infinite subset X of ω is either countable or has the same cardinality as $\wp(\omega)$.

Is this true? Is it false? Is it provable? Is its negation provable?

In 1900 a famous German mathematician and logician named David Hilbert made a list of 23 important open problems in mathematics. In an address at the International Congress of Mathematicians in Paris in 1900 he talked about the first 10 of these problems. The continuum hypothesis was the first of these problems, so it is also known as **Hilbert's first problem**.

CH led to the following interesting results:

- The fact that the negation of CH is not provable using the usual axioms of set theory was proven by Gödel in 1940.
- The fact that CH is not provable using the usual axioms of set theory was proven by Paul Cohen in 1962.

How can we prove something is not provable? The question only becomes precise once we specify the deductive system. If we have a deductive system, a notion of a logical interpretation, and soundness result, then we can prove a formula s is unprovable by constructing a logical interpretation such that $\mathcal{I}s = 0$. In the case of the results above, the deductive system is first-order logic with the **Zermelo-Fraenkel** of set theory with the axiom of choice, which is commonly referred to as **ZFC**.

There is much more we could say about set theory, but this is not a course on set theory! If you want to read more about set theory, try the first chapters of [26] or [25].

8 Decision Trees

In this section we investgate equivalence of propositional formulas. Two formulas s, t are equivalent if the equation $s = t$ is deducible. We identify a class of canonical propositional formulas such that every propositional formula is equivalent to exactly one canonical formula. The canonical form is based on so-called decision trees. We obtain an efficient algorithm for equivalence checking by employing a dynamic programming technique on a minimal graph representation of decision trees known as binary decision diagram (BDD). The data structures and algorithms presented in this section have many applications, including computer-aided design of computer hardware and verification of finite state systems (i.e., model checking). The essential ideas are due to Randal Bryant [33].

8.1 Boolean Functions

An important building block of the hardware of computers are functional circuits. A functional circuit maps some inputs $x_1, \dots, x_m$ to some outputs $y_1, \dots, y_n$. Inputs and outputs are two-valued. Every output is determined as a function of the inputs. This leads to the notion of a Boolean function, an abstraction that is essential for hardware design.

A **Boolean variable** is a variable of type B. Let X be a nonempty and finite set of Boolean variables. The following definitions are parameterized with X.

An **assignment** (σ) is a function $X \to B$. A **Boolean function** (φ) is a function $(X \to B) \to B$. Seen from the circuit perspective, an assignment σ provides values σx for the inputs $x \in X$, and a Boolean function decribes how an output is obtained from the inputs. Some authors call Boolean functions switching functions. Sometimes it is helpful to see a Boolean function as a set of assignments.

In order to design a functional circuit, one must know which Boolean functions (one per output) it ought to compute. So how can electrical engineers specify Boolean functions? A simple-minded approach are **truth tables**. For instance, given $x, y \in X$, we can see the truth table

x	y	$\varphi\sigma$
0	0	0
0	1	1
1	0	1
1	1	0

as a specification of the Boolean function that returns 1 if and only if its inputs x and y are different. A more compact specification of this Boolean function would be the propositional formula $x \not\equiv y$. Clearly, if there are many relevant inputs, the specification of a Boolean function with a formula can be more compact than the specification with a truth table.

Let PF be the set of all propositional formulas containing only variables from X. We define a function $\mathcal{F} \in \mathrm{PF} \to (X \to \mathbb{B}) \to \mathbb{B}$ such that $\mathcal{F}s$ is the Boolean function specified by the formula s:

$$\begin{aligned}
\mathcal{F}\bot\sigma &= 0 \\
\mathcal{F}\top\sigma &= 1 \\
\mathcal{F}x\sigma &= \sigma x \qquad \text{if } x \in X \\
\mathcal{F}(\neg s)\sigma &= 1 - \mathcal{F}s\sigma \\
\mathcal{F}(s \wedge t)\sigma &= \min\{\mathcal{F}s\sigma, \mathcal{F}t\sigma\} \\
\mathcal{F}(s \vee t)\sigma &= \max\{\mathcal{F}s\sigma, \mathcal{F}t\sigma\} \\
\mathcal{F}(s \to t)\sigma &= \max\{1 - \mathcal{F}s\sigma, \mathcal{F}t\sigma\} \\
\mathcal{F}(s \equiv t)\sigma &= (\mathcal{F}s\sigma = \mathcal{F}t\sigma)
\end{aligned}$$

Proposition 8.1.1 Let $s \in \mathrm{PF}$, $\mathcal{I}$ be a logical interpretation, and σ be the unique assignment such that $\sigma \subseteq \mathcal{I}$. Then $\mathcal{I}s = \mathcal{F}s\sigma$.

Proof By induction on s. ■

From the proposition it's clear that assignments can play the role of logical interpretations for propositional formulas. While logical interpretations come with redundant and irrelevant information, assignments only contain the information that is necessary for the evaluation of propositional formulas.

Often it is necessary to decide whether two formulas $s, t \in \mathrm{PF}$ represent the same Boolean function. For instance, s might be the specification of a Boolean function and t may describe the implementation of this function in terms of more primitive functions. Then the implementation is correct if and only if $\mathcal{F}s = \mathcal{F}t$.

Proposition 8.1.2 Let $s, t \in \mathrm{PF}$. Then $\mathcal{F}s = \mathcal{F}t$ iff $s \equiv t$ is a tautology.

Proof $\mathcal{F}s = \mathcal{F}t$ holds if s, t evaluate to the same value with every assignment, and $s = t$ is a tautology if s, t evaluate to the same value with every logical interpretation. Thus the claim follows with Proposition 8.1.1. ∎

Given Proposition 8.1.2, we can say that the function $\mathcal{F}$ constitutes a semantics for propositional formulas. Since there are only finitely many assignments, the semantics provided by $\mathcal{F}$ is effective in that it gives us a naive algorithm for deciding whether a propositional formula is a tautology. The algorithm may even be practical if a formula is not a tautology and we implement it with heuristics that find a falsifying assignment quickly. On the other hand, if we want to show that a formula is a tautology, the tableau method seems more promising.

We say that two terms s, t are **equivalent** if they have the same type and the equation $s = t$ is deducible.

Proposition 8.1.3 Let $s, t \in \mathrm{PF}$. Then the following statements are equivalent:

1. s and t are equivalent.
2. $s = t$ is a tautology.
3. $\mathcal{F}s = \mathcal{F}t$.

Proof Follows with Theorem 6.4.1 and Proposition 8.1.2. ∎

$\mathcal{F}$ illustrates an important semantic idea that we have not seen so far. The interesting thing about $\mathcal{F}$ is that it gives us a *denotational characterization* of propositional equivalence: Two propositional formulas are equivalent if and only if they denote the same same semantic object (i.e., a Boolean function).

Let's return to Boolean functions. Can every Boolean function be represented by a propositional formula? The answer is yes.

Proposition 8.1.4 Let φ be a Boolean function. Then:

$$\varphi \;=\; \mathcal{F}\left(\bigvee_{\substack{\sigma\in X\to\mathbb{B}\\ \varphi\sigma=1}} \; \bigwedge_{x\in X} \text{ if } \sigma x = 1 \text{ then } x \text{ else } \neg x \right)$$

Proof Let σ be an assignment. It suffices to show that the left hand side yields 1 for sigma if and only if the right hand side does. This is easy to verify. Remark: if the index set of the disjunction is empty, it represents $\perp$. ∎

Exercise 8.1.5 We require that X is finite so that every Boolean function can be represented by a formula. Suppose X is infinite. How can we obtain a Boolean function that cannot be represented by a propositional formula?

8.2 Decision Trees and Prime Trees

Every Boolean function can be represented by infinitely many propositional formulas. Given a Boolean function, formulas can represent it more or less explicitly. For instance $\top$ and $\neg\neg x \to x$ both represent the same Boolean function. Of course, in general it is not clear what more or less explicit means. However, the following question is meaningful: Can we find a class of canonical propositional formulas such that every Boolean function can be represented by one and only one canonical formula, and such that canonical formulas are informative representations of Boolean functions?

In the following we will study a system of canonical propositional formulas that is based on the notion of a decision tree. We start with the notation

$$(s,t,u) \;:=\; \neg s \wedge t \vee s \wedge u$$

and call formulas of this form **conditionals**. The following proposition explains why we use this name.

Proposition 8.2.1 The following formulas are tautologies:

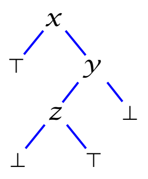
1. $(\bot, x, y) = x$
2. $(\top, x, y) = y$
3. $(x, y, y) = y$

Exercise 8.2.2 Find tableau proofs for the tautologies in Proposition 8.2.1.

Decision trees are defined recursively:

1. $\bot$ and $\top$ are decision trees.
2. (x, s, t) is a decision tree if x is a Boolean variable and s and t are decision trees.

As the name suggests, decision trees can be thought of as trees. For instance, $(x, \top, (y, (z, \bot, \top), \bot))$ may be seen as the tree



To compute $\mathcal{J}s\sigma$ for a decision tree s, we start at the root of s and follow the path determined by σ ($\sigma x = 0$ means "go left", and $\sigma x = 1$ means "go right"). If we reach a leaf, the result is determined by the label of the leaf (0 for $\bot$ and 1 for $\top$).

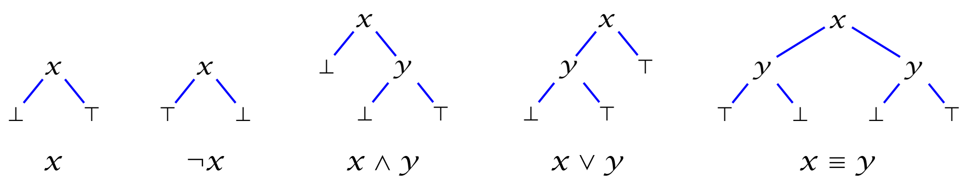


Figure 8.1: Prime trees for some simple formulas

Proposition 8.2.3 Let $(x, s_0, s_1) \in \mathrm{PF}$. Then:
$\mathcal{F}(x, s_0, s_1)\sigma = \mathit{if}\ \sigma x{=}0\ \mathit{then}\ \mathcal{F}s_0\sigma\ \mathit{else}\ \mathcal{F}s_1\sigma$

Proposition 8.2.4 (Coincidence)
Let $s \in \mathrm{PF}$ and $\sigma x = \sigma' x$ for all $x \in \mathcal{V}s$. Then $\mathcal{F}s\sigma = \mathcal{F}s\sigma'$.

Given decision trees, it is straightforward to define a canonical subclass. A decision tree is **reduced** if none of its subtrees has the form (x, s, s). We assume a linear order on the set of all Boolean variables and write $x < y$ if x is smaller than y. A decision tree is **ordered** if the variables get larger as one goes down on a path from the root to a leaf. The example tree shown above is ordered if and only if $x < y < z$. A **prime tree** is a reduced and ordered decision tree. Formally, we define prime trees recursively:

1. $\perp$ and $\top$ are prime trees.
2. (x, s, t) is prime tree if s and t are different prime trees (i.e., $s \neq t$) and x is a Boolean variable that is smaller than every variable that occurs in s or t.

We will show that every propositional formula is equivalent to exactly one prime tree. Figure 8.1 shows the prime trees for some simple formulas.

8.3 Existence of Prime Trees

First we outline an algorithm that computes for a propositional formula an equivalent prime tree. The algorithm is based on the following proposition.[1]

Proposition 8.3.1 (Shannon Expansion)
For every formula s and every Boolean variable x: $\vdash s = (x,\, s^x_\perp,\, s^x_\top)$.

Proof Straightforward with BCAR and Proposition 8.2.1. ■

[1] Claude Shannon showed in his famous 1937 master's thesis done at MIT that the arrangement of the electromechanical relays then used in telephone routing switches could be analyzed with Boolean algebra.

The algorithm works by recursion on the number of Boolean variables occurring in s. If s contains no Boolean variables, s can be evaluated to $\bot$ or $\top$. Otherwise, the algorithm determines the least Boolean variable x occurring in s and obtains prime trees s_0, s_1 for $s^x_\bot$ and $s^x_\top$ by recursion. If $s_0 \neq s_1$, we obtain the prime tree (x, s_0, s_1), otherwise s_0 does the job.

In the following, we will show the correctness of the algorithm by stating and proving some essential properties.

We define the **free variables of a term** s as $\mathcal{V}s := \{\, x \in \mathcal{N}s \mid x \text{ is a variable} \,\}$. A term s is **variable-free** if $\mathcal{V}s = \emptyset$.

Proposition 8.3.2 (Evaluation) Let s be a variable-free propositional formula. Then $\vdash s{=}\bot$ or $\vdash s{=}\top$.

Proof By induction on s. Straightforward. ■

Lemma 8.3.3 Let s be a propositional formula and x be the least variable that occurs in s. Let s_0 and s_1 be prime trees that are equivalent to $s^x_\bot$ and $s^x_\top$, respectively. Moreover, let $\mathcal{V}s_0 \subseteq \mathcal{V}s^x_\bot$ and $\mathcal{V}s_1 \subseteq \mathcal{V}s^x_\top$. Then:

1. If $s_0 = s_1$, then $s^x_\bot$ is a prime tree that is equivalent to s.
2. If $s_0 \neq s_1$, then (x, s_0, s_1) is a prime tree that is equivalent to s.

Proof Follows with Proposition 8.3.1 and Proposition 8.2.1. ■

Proposition 8.3.4
For every propositional formula s there exists an equivalent prime tree t such that $\mathcal{V}t \subseteq \mathcal{V}s$.

Proof Follows with Lemma 8.3.3 by induction on the number of variables occurring in s. ■

Exercise 8.3.5 Draw all prime trees containing no other variables but x and y. Assume $x < y$. For each tree give an equivalent propositional formula that is as simple as possible.

Exercise 8.3.6 Let s be the propositional formula $x = (y = z)$. Assume $x < y < z$. Draw the prime trees for the following formulas: s, $\neg s$, $s \wedge s$, $s \rightarrow s$.

8.4 Example: Diet Rules

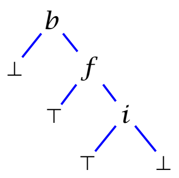
On a TV show a centenarian is asked for the secret of his long life. Oh, he says, my long life is due to a special diet that I started 60 years ago and follow by every day. The presenter gets all exited and asks for the diet. Oh, that's easy, says the old gentleman, there are only 3 rules you have to follow:

1. If you don't take beer, you must have fish.
2. If you have both beer and fish, don't have ice cream.
3. If you have ice cream or don't have beer, then don't have fish.

Let's look at the diet rules from a logical perspective. Obviously, the diet is only concerned with three Boolean properties of a meal: having beer, having fish, and having ice cream . We can model these properties with three Boolean variables b, f, i and describe the diet with a propositional formula that evaluates to 1 if the diet is satisfied by a meal:

$$(\neg b \to f) \;\wedge\; (b \wedge f \to \neg i) \;\wedge\; (i \vee \neg b \to \neg f)$$

The formula is one one possible description of the diet. More abstractly, the diet is represented by the Boolean function decribed by the formula. A related representation of the diet is the prime tree that is equivalent to the initial formula:



Now we know that the diet is observed if and only if the following rules are observed:

1. Always drink beer.
2. Do not have both fish and ice cream.

Clearly, the prime tree represents the diet much more explicitly than the initial formula obtained form the rules given by the gentleman.

Exercise 8.4.1 Four girls agree on some rules for a party:

i) Whoever dances which Richard must also dance with Peter and Michael.

ii) Whoever does not dance with Richard is not allowed to dance with Peter and must dance with Christophe.

iii) Whoever does not dance with Peter is not allowed to dance with Christophe.

Express these rules as simply as possible.

a) Describe each rule with a propositional formula. Do only use the variables c (Christophe), p (Peter), m (Michael), r (Richard).

b) Give the prime tree that is equivalent to the conjunction of the rules. Use the order $c < p < m < r$.

8.5 Uniqueness of Prime Trees

We use σ_b^x to denote the assignment that is like σ except that it maps x to b.

Lemma 8.5.1 If $s, t \in \mathrm{PF}$ are different prime trees, then $\mathcal{F}s$ and $\mathcal{F}t$ are different Boolean functions.

Proof By induction on $|s| + |t|$. Let s, t be different prime trees. We show that there is an assignment σ such that $\mathcal{F}s\sigma \neq \mathcal{F}t\sigma$.

Case $s, t \in \{\bot, \top\}$. Every σ does the job.

Case $s = (x, s_0, s_1)$ and $x \notin \mathcal{N}t$. By induction we have an assignment σ such that $\mathcal{F}s_0\sigma \neq \mathcal{F}s_1\sigma$. Since x occurs neither in s_0 nor s_1, we have $\mathcal{F}s\sigma_0^x \neq \mathcal{F}s\sigma_1^x$ since $\mathcal{F}s\sigma_0^x = \mathcal{F}s_0\sigma_0^x = \mathcal{F}s_0\sigma \neq \mathcal{F}s_1\sigma = \mathcal{F}s_1\sigma_1^x = \mathcal{F}s\sigma_1^x$. But $\mathcal{F}t\sigma_0^x = \mathcal{F}t\sigma_1^x$ since x does not occur in t. Hence $\mathcal{F}s\sigma_0^x \neq \mathcal{F}t\sigma_0^x$ or $\mathcal{F}s\sigma_1^x \neq \mathcal{F}t\sigma_1^x$.

Case $t = (x, t_0, t_1)$ and $x \notin \mathcal{N}s$. Analogous to previous case.

Case $s = (x, s_0, s_1)$ and $t = (x, t_0, t_1)$. Then $s_0 \neq t_0$ or $s_1 \neq t_1$. By induction there exists an assignment σ such that $\mathcal{F}s_0\sigma \neq \mathcal{F}t_0\sigma$ or $\mathcal{F}s_1\sigma \neq \mathcal{F}t_1\sigma$. By coincidence $\mathcal{F}s_0\sigma_0^x \neq \mathcal{F}t_0\sigma_0^x$ or $\mathcal{F}s_1\sigma_1^x \neq \mathcal{F}t_1\sigma_1^x$. Hence $\mathcal{F}s\sigma_0^x \neq \mathcal{F}t\sigma_0^x$ or $\mathcal{F}s\sigma_1^x \neq \mathcal{F}t\sigma_1^x$.

To see that the case analysis is exhaustive, consider the case where both s and t are non-atomic trees with the root variables x and y. If $x < y$, then x does not occur in t since all variables in t are greater or equal than y and hence are greater that x. If $y < x$, then y does not occur in s since all variables in s are greater or equal than x and hence are greater than y. ■

Theorem 8.5.2 (Prime Tree) For every propositional formula there exists exactly one equivalent prime tree.

Proof The existence follows with Proposition 8.3.4. To show the uniqueness, assume that s is a propositional formula and t_1, t_2 are different prime trees that are equivalent to s. Without loss of generality we can assume that $t_1, t_2 \in \mathrm{PF}$ (because we can choose $X = \mathcal{V}t_1 \cup \mathcal{V}t_2$). Hence $\mathcal{F}t_1 \neq \mathcal{F}t_2$ by Lemma 8.5.1. Hence t_1, t_2 are not equivalent by Proposition 8.1.3. Contradiction since t_1, t_2 are both equivalent to s. ■

8.6 Properties of Prime Trees

For every propositional formula s we denote the unique prime tree equivalent to s with πs. We call πs the **prime tree for s**.

Proposition 8.6.1 Let s and t be propositional formulas.

1. s is equivalent to πs.
2. $\mathcal{V}(\pi s) \subseteq \mathcal{V}s$.

3. s, t are equivalent if and only if $\pi s = \pi t$.
4. s is a tautology if and only if $\pi s = \top$.

Proof Claim (1) follows by definition of πs. Claim (2) follows with Proposition 8.3.4 and Theorem 8.5.2. Claim (3) follows with (1) and Theorem 8.5.2. For Claim (4) first note that s is a tautology iff $s = \top$ is a tautology. By Proposition 8.1.3 this is the case iff s and $\top$ are equivalent. By Claim (3) this is the case iff $\pi s = \pi \top$. Now we are done since $\pi \top = \top$. ∎

The **significant variables** of a propositional formula are the variables occurring in its prime tree. The following lemma says that the significant variables of a prime tree s are significant variables of the Boolean function $\mathcal{F}s$.

Lemma 8.6.2 Let $s \in \mathrm{PF}$ be a prime tree and $x \in \mathcal{V}s$. Then there exists an assignment σ such that $\mathcal{F}s\sigma^x_\perp \neq \mathcal{F}s\sigma^x_\top$.

Proof By contradiction. Assume $\mathcal{F}s\sigma^x_\perp = \mathcal{F}s\sigma^x_\top$ for all assignments σ. Then $\mathcal{F}s\sigma = \mathcal{F}s\sigma^x_\perp$ for all σ. Hence s and $s^x_\perp$ are equivalent. Thus $\pi s = \pi(s^x_\perp)$ by Proposition 8.6.1 (3). Since s is a prime tree, we have $s = \pi s = \pi(s^x_\perp)$. This is a contradiction since $x \in \mathcal{V}s = \mathcal{V}(\pi(s^x_\perp)) \subseteq \mathcal{V}(s^x_\perp)$ by Proposition 8.6.1 (2) and $x \notin \mathcal{V}(s^x_\perp)$. ∎

Proposition 8.6.3 For every propositional formula s:

1. If x is a significant variable of s, then x occurs in s.
2. A Boolean variable x is significant for s if and only if there exists an assignment σ such that $\mathcal{F}s\sigma^x_\perp \neq \mathcal{F}s\sigma^x_\top$.

Proof Claim (1) follows with Proposition 8.6.1. For Claim (2) we assume without loss of generality that s is a prime tree (recall that $\mathcal{F}s = \mathcal{F}(\pi s)$ holds for all propositional formulas). The left-to-right direction follows with Lemma 8.6.2. To see the other direction, let $\mathcal{F}s\sigma^x_\perp \neq \mathcal{F}s\sigma^x_\top$. By Coincidence we have $x \in \mathcal{V}s$. Since s is a prime tree, x is a significant variable of s. ∎

A variable $x \in X$ is **significant** for a Boolean function φ if there exists an assignment σ such that $\varphi\sigma^x_\perp \neq \varphi\sigma^x_\top$. By the the proposition just stated we know that the significant variables of a formula $s \in \mathrm{PF}$ are exactly the significant variables of the Boolean function $\mathcal{F}s$.

Boolean functions and $\mathcal{F}$ are defined with respect to a finite set of variables X. In contrast, the definition of the prime tree representation πs and of the significant variables of s does not depend on X. In principle, it is possible to fix X as the set of all Boolean variables, with the consequence that not every Boolean function can be described by a propositional formula. In this case, prime trees are a perfect representation for the finitary Boolean functions.

Prime trees are a canonical representation for propositional formulas. Given a set S of syntactic objects and an equivalence relation on these objects, a canonical representation is a set $C \subseteq S$ such that for every object in S there is exactly one equivalent object in C. In § 3.7 we have seen a canonical representation for simply typed terms: Every term is λ-equivalent to exactly one λ-normal term (Corollary 3.7.3).

Exercise 8.6.4

a) Find a propositional formula s that contains the variables x, y, z and has x as its only significant variable.

b) Determine the significant variables of the formula $(x \to y) \wedge (x \vee y) \wedge (y \vee z)$.

8.7 Prime Tree Algorithms

Given two prime trees s and t, how can we efficiently compute the prime trees for $\neg s$, $s \wedge t$, $s \vee t$ and so on? It turns out that there are elegant algorithms that perform well in practice. Here we will develop the algorithms for negation and conjunction. The algorithms for the other operations can be obtained along the same lines.

To develop th algorithms for negation and conjunction, we first define the functions to be computed by the algorithms.

$$\begin{array}{ll} \mathit{not} \in \mathrm{PT} \to \mathrm{PT} & \qquad \mathit{and} \in \mathrm{PT} \to \mathrm{PT} \to \mathrm{PT} \\ \mathit{not}\, s = \pi(\neg s) & \qquad \mathit{and}\, s\, t = \pi(s \wedge t) \end{array}$$

We base the algorithm for negation on the tautologies (verify!)

$$\begin{aligned} \neg\bot &= \top \\ \neg\top &= \bot \\ \neg(x, y, z) &= (x, \neg y, \neg z) \end{aligned}$$

With the tautologies one can verify the equations

$$\begin{aligned} \pi(\neg\bot) &= \top \\ \pi(\neg\top) &= \bot \\ \pi(\neg(x, s, t)) &= (x, \pi(\neg s), \pi(\neg t)) \qquad \text{if } (x, s, t) \text{ is a prime tree} \end{aligned}$$

The correctness of the first two equations is obvious. For the corectness of the last equation we show 2 things:

1. The formula on the left is equivalent to the formula on the right. To verify this, we can erase all applications of π since π always yields equivalent formulas. Now we are left with an instance of the third tautology.
2. The formula on the right is a prime tree. Let (x, s, t) be a prime tree. Clearly, the formula on the right is a decision tree. We need to show that it is ordered and reduced. Since $\pi(\neg s)$ and $\pi(\neg t)$ contain only variables that are in s and t and (x, s, t) is a ordered, $(x, \pi(\neg s), \pi(\neg t))$ is ordered. Since (x, s, t) is a prime tree, s and t are not equivalent. Hence $\neg s$ and $\neg t$ are not equivalent (since $x{=}y \equiv \neg x{=}\neg y$ is a tautology). Hence $\pi(\neg s)$ and $\pi(\neg t)$ are different prime trees.

Together, the two properties yield the correctness of the equation since for every formula there is only one equivalent prime tree.

Now we have the following procedure:

$$\begin{aligned} & \mathit{not} : \mathrm{PT} \to \mathrm{PT} \\ & \qquad \mathit{not}\ \bot = \top \\ & \qquad \mathit{not}\ \top = \bot \\ & \mathit{not}(x, s, t) = (x,\ \mathit{not}\ s,\ \mathit{not}\ t) \end{aligned}$$

The equations are exhaustive and terminating. Moreover, they are valid for the function *not* (they reduce to the above equations with π by unfolding the definition of the function not). Hence the procedure *not* computes the function *not*.

Next we devise an algorithm for conjunction. This time we employ the following tautologies (verify!):

$$\begin{aligned} \bot \wedge y &= \bot \\ \top \wedge y &= y \\ (x, y, z) \wedge (x, y', z') &= (x, y \wedge y', z \wedge z') \\ (x, y, z) \wedge u &= (x, y \wedge u, z \wedge u) \end{aligned}$$

Moreover, we exploit the commutativity of $\wedge$. We also use an auxiliary function

$$\begin{aligned} & \mathit{red} \in \mathrm{DT} \to \mathrm{DT} \\ & \mathit{red}\ \bot = \bot \\ & \mathit{red}\ \top = \top \\ & \mathit{red}(x, s, t) = \text{if } s = t \text{ then } s \text{ else } (x, s, t) \end{aligned}$$

Now we can verify the following equations:

$$\begin{aligned}
\pi(\bot \wedge t) &= \bot \\
\pi(\top \wedge t) &= t \\
\pi((x, s_0, s_1) \wedge (x, t_0, t_1)) &= \mathit{red}(x, \pi(s_0 \wedge t_0), \pi(s_1 \wedge t_1)) \\
\pi((x, s_0, s_1) \wedge t) &= \mathit{red}(x, \pi(s_0 \wedge t), \pi(s_1 \wedge t)) \\
&\qquad \text{if } t = (y, t_0, t_1) \text{ and } x < y
\end{aligned}$$

As for negation, the correctness of the equations is established in 2 steps. First one verifies that for each equations the formula on the left is equivalent to the formula on the right. Since π and *red* yield equivalent formulas, we can erase their applications. Now we are left with instances of the above tautologies. For the second step we show that the formulas on the right are prime trees, provided the arguments on the left hand side are prime trees. This is easy and explains the condition $x < y$ coming with the last equation.

Now we have the following procedure:

$$\begin{aligned}
&\mathit{and} : \mathrm{PT} \to \mathrm{PT} \to \mathrm{PT} \\
\mathit{and}\ \bot\ t &= \bot \\
\mathit{and}\ \top\ t &= t \\
\mathit{and}\ s\ \bot &= \bot \\
\mathit{and}\ s\ \top &= s \\
\mathit{and}\ (x, s_0, s_1)\ (x, t_0, t_1) &= \mathit{red}(x,\ \mathit{and}\ s_0\ t_0,\ \mathit{and}\ s_1\ t_1) \\
\mathit{and}\ (x, s_0, s_1)\ t &= \mathit{red}(x,\ \mathit{and}\ s_0\ t,\ \mathit{and}\ s_1\ t) \\
&\qquad \text{if } t = (y, t_0, t_1) \text{ and } x < y \\
\mathit{and}\ s\ (y, t_0, t_1) &= \mathit{red}(y,\ \mathit{and}\ s\ t_0,\ \mathit{and}\ s\ t_1) \\
&\qquad \text{if } s = (x, s_0, s_1) \text{ and } x > y
\end{aligned}$$

The procedure *and* computes the function *and* since the following properties are satisfied:

1. The equations are exhaustive and terminating.
2. The equations hold for the function *and*. This is the case since the equations reduce to the above tautologies (up to commutativity) by using the definitions of the functions *and* and *red*.

You now know enough so that you can devise algorithms for the other Boolean operations. Things are as before since for every name $\circ : BBB$ the following equations are deducible with BCAR ($\circ$ is written as infix operator):

$$\begin{aligned}(x,y,z)\circ(x,y',z') &= (x, y\circ y', z\circ z')\\ u\circ(x,y',z') &= (x, u\circ y', u\circ z')\\ (x,y,z)\circ u &= (x, y\circ u, z\circ u)\end{aligned}$$

This follows with BCAR on x and the tautologies of Proposition 8.2.1.

Exercise 8.7.1 Develop an algorithm that for two prime trees s, t yields the prime tree for $s \equiv t$. Implement the algorithm in Standard ML. Proceed as follows:

a) Complete the following equations so that they become tautologies on which the algorithm can be based.

$$\begin{aligned}(x\equiv\top) &\equiv\\ (\bot\equiv\bot) &\equiv\\ ((x,y,z)\equiv(x,y',z')) &\equiv\\ ((x,y,z)\equiv u) &\equiv\end{aligned}$$

b) Complete the declarations of the procedures *red* and *equiv* so that *equiv* computes for two prime trees s, t the prime tree for $s \equiv t$. The variable order is the order of *int*. Do not use other procedures.

```
type var = int
datatype dt = F | T | D of var * dt * dt

fun red x s t =

fun equiv T t =
  | equiv s T =
  | equiv F F =
  | equiv F (D(y,t0,t1)) =
  | equiv (D(x,s0,s1)) F =
  | equiv (s as D(x, s0, s1)) (t as D(y, t0, t1)) =
      if x=y then
      else if x<y then
      else
```

Exercise 8.7.2 Let decision trees be represented as in Exercise 8.7.1, and let propositional formulas be represented as follows:

```
datatype pf = FF | TT | V of var | NEG of pf | AND of pf * pf
            | OR of pf * pf | IMP of pf * pf | EQ of pf * pf
```

Write a procedure $pi : pf \to dt$ that yields the prime tree for a propositional formula. Be smart and only use the prime tree algorithm for implication (all propositional connectives can be expressed with $\to$ and $\bot$).

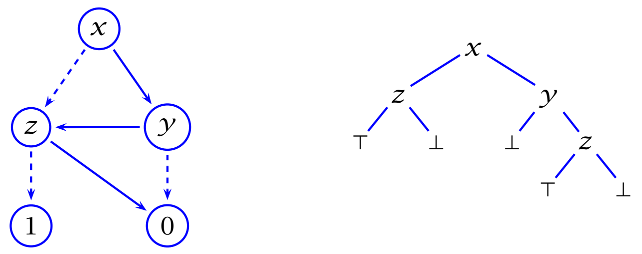


Figure 8.2: A BDD and the decision tree represented by the topmost node

Exercise 8.7.3 Find two prime tree (x, s_0, s_1) and t such that:

i) $(x, \pi(s_0 \to t), \pi(s_1 \to t))$ is not a prime tree.

ii) $\forall y \in \mathcal{V}t\colon\ x < y$.

8.8 BDDs

Trees can be represented as nodes of graphs. Graphs whose nodes represent decision trees are called BDDs (binary decision diagrams). Binary decision diagrams (BDD) were introduced by Lee (Lee 1959), and further studied and made known by Akers (Akers 1978) and Boute (Boute 1976).

Figure 8.2 shows a BDD. The node labeled with the variable x represents the decision tree shown to the right. Dotted edges of the graph lead to left subtrees and solid edges to right subtrees. Subtrees that occur more than once in a decision tree need only be represented once in a BDD (so-called **structure sharing**). In our example BDD the node labeled with z represents a subtree that occurs twice in the decision tree on the right.

Formally, a **BDD** is a function γ such that there exists a natural number $N \geq 1$ such that

1. $\gamma \in \{2, \dots, N\} \to Var \times \{0, \dots, N\} \times \{0, \dots, N\}$.
2. $\forall\ (n, (x, n_0, n_1)) \in \gamma\colon\ n > n_0 \wedge n > n_1$.

The **nodes of γ** are the numbers $0, \dots, N$. The nodes 0 und 1 represent the decision trees $\perp$ and $\top$. A node $n \geq 2$ with $\gamma n = (x, n_0, n_1)$ carries the label x and has two outgoing edges pointing to n_0 and n_1, where the edge to n_0 is dotted and the edge to n_1 is solid. Note that the second condition in the definition of BDDs ensures that BDDs are acyclic. The BDD drawn in Figure 8.2 is the following function (**in table representation**):

2	$(z, 1, 0)$
3	$(y, 0, 2)$
4	$(x, 2, 3)$

For every BDD γ we define the function

$$
\begin{aligned}
&\mathcal{T}_\gamma \in \{0,1\} \cup \mathrm{Dom}\,\gamma \to DT\\
&\quad \mathcal{T}_\gamma 0 = \bot\\
&\quad \mathcal{T}_\gamma 1 = \top\\
&\quad \mathcal{T}_\gamma n = (x, \mathcal{T}_\gamma n_0, \mathcal{T}_\gamma n_1) \qquad \text{if } \gamma n = (x, n_0, n_1)
\end{aligned}
$$

which yields for every node n of γ the decision tree represented by γ and n.

A BDD is **minimal** if different nodes represent different trees. The BDD in Figure 8.2 is minimal.

Proposition 8.8.1 (Minimality) A BDD is minimal if and only if it is injective.

Proof Let γ be a BDD such that $\mathrm{Dom}\,\gamma = \{2, \dots, n\}$. That minimality implies injectivity follows by contraposition. For the other direction assume that γ is injective. We show the minimality of γ by induction on n. For $n = 1$ the claim is obvious. Otherwise, let $\gamma n = (x, n_0, n_1)$. Assume γ is not minimal. Then the exists a $k \neq n$ such that $\mathcal{T}_\gamma k = \mathcal{T}_\gamma n$. Hence $\gamma k = (x, k_0, k_1)$ such that $\mathcal{T}_\gamma k_0 = \mathcal{T}_\gamma n_0$ and $\mathcal{T}_\gamma k_1 = \mathcal{T}_\gamma n_1$. By induction we know that the restriction of γ to $\{2, \dots, n-1\}$ is minimal. Hence $k_0 = n_0$ and $k_1 = n_1$. Hence $\gamma k = \gamma n$. Since $k \neq n$ this contradicts the assumption that γ is injective. ■

Given the table representation of a BDD, it is very easy to see whether it is minimal: The BDD is minimal if and only if no triple (x, n_0, n_1) occurs twice in the right column of the table representation.

Note that there is exactly one minimal BDD that represents all subtrees of a given prime tree. All nodes of this BDD are reachable from the node that represents the given subtree. Note that this *root* appears as last node in the table representation.

Techniques that represent terms as numbers that identify entries in tables are know as *term indexing*. BDDs are a typical example of term indexing.

Exercise 8.8.2 Let s be the propositional formula $(x \wedge y \equiv x \wedge z) \wedge (y \wedge z \equiv x \wedge z)$. Assume the variable order $x < y < z$.

a) Draw the prime tree for s.

b) Draw a minimal BDD whose nodes represent the subtrees of the prime tree for s.

c) Give the table representation of the BDD. Label each non-terminal node of your BDD with the number representing it.

Exercise 8.8.3 (Parity Function) Let the Boolean variables $x_1 < x_2 < x_3 < x_4$ be given. The *parity function* for these variables is the Boolean function that yields 1 for an assignment σ iff the sum $\sigma x_1 + \sigma x_2 + \sigma x_3 + \sigma x_4$ is an even number. Draw the minimal BDD whose root represents the prime tree for the parity function. Observe that it is easy to obtain a minimal BDD for parity functions with additional variables ($x_4 < x_5 < x_6 < \cdots$). Observe that the prime tree represented by the root node of the BDD is exponentially larger than the BDD.

Exercise 8.8.4 (Impact of Variable Order) The size of the BDD representing the parity function in Exercise 8.8.3 does not depend on the chosen variable order. In general, this is not the case. There may be an exponential blow up if an unfortunate variable order is chosen. Consider the formula

$$(x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge \cdots \wedge (x_{2n-1} \vee x_{2n})$$

The minimal BDD for this formula has $2n + 2$ nodes if we choose the order $x_1 < x_2 < \cdots < x_{2n}$. Draw it for $n = 2$. If we choose the order

$$x_1 < x_3 < \cdots < x_{2n-1} < x_2 < x_4 < \cdots < x_{2n}$$

the minimal BDD has 2^{n+1} nodes. Draw it for $n = 2$.

8.9 Polynomial Runtime through Memorization

With the BDD representation it is possible to implement the prime tree algorithms for the Boolean connectives with the runtime $O(||m|| \cdot ||n||)$ where m and n are the nodes representing the prime trees and $||m||$ and $||n||$ are the counts of the nodes reachable from m and n, respectively. The basic observation is that every call of the procedure will take as arguments two nodes m' and n' that are reachable from m and n, respectively. Hence, if we memorize for each call already done that is was done and what result it returned, we can avoid computing the same call more than once. Without this dynamic programming technique prime tree algorithms like *and* have exponential worst-case complexity. The memorization is implemented with hashing. For this it is essential that the trees are represented as numbers.

8.10 Remarks

Decision trees and graph representations of decision trees have been known for a long time, but the canonical representation with ordered and reduced decision trees and minimal graph representations were discovered only in 1986 by Randal Bryant [33]. You will find his famous paper in the Internet. Huth and Ryan's textbook [23] gives a detailed presentation of BDDs. You will also find useful information in the Wikipedia entry for binary decision diagrams.

9 Modal Logic

- Modal logic was conceived as an extension of propositional logic that can express additional modes of truth: *necessarily true* and *possibly true*.
- In 1918, C.I. Lewis published a deductive system for modal logic. His system was improved by Kurt Gödel in 1933.
- In 1963, Saul Kripke gave the by now standard semantics for modal logic and showed soundness and completeness.
- Before computer scientist got involved, modal logic was mainly developed by logicians from philosophy.
- Temporal logic is an extension of modal logic. Major contributions were made by the philosophers Arthur Prior (1967) and Hans Kamp (1968).
- In 1977, Amir Pnuelli realized that temporal logics could be used for specifying and reasoning about concurrent programs. The temporal logics LTL and CTL are now in wide use.
- In 1976, Vaugham Pratt invented dynamic logic, which is an extended modal logic to be used for program verification.
- In 1991, Klaus Schild discovers that terminological logics, then developed for knowledge representation, are in fact modal logics. Nowadays, terminological logics are known as description logics [4]. Description logic is the basis for the web ontology language OWL (see Wikipedia). There is a connection between dynamic logic and description logic.
- Modal logics are decidable and have the finite model property. Terminating tableau systems can be used as decision procedures and are the most efficient choice as it comes to description logics and OWL.
- We consider a modal logic M that subsumes the basic modal logic K and the basic description logic ALC. We develop a terminating tableau system for M.
- We study modal logic as a fragment of simple type theory, as we did before for propositional logic and first-order logic.

9.1 Modal Terms

We assume a sort I of **individuals** and work with three types of variables:

$x, y : I$	individual variables
$p, q : IB$	property variables
$r : IIB$	relation variables

Note that propositions are unary predicates and relations are binary predicates. The heart of modal logic are terms of type IB describing properties of individuals. Modal terms are obtained with the following names called **modal constants**:

$$\begin{aligned} \dot\bot, \dot\top &: IB \\ \dot\neg &: (IB)IB \\ \dot\wedge, \dot\vee, \dot\rightarrow, \dot\equiv &: (IB)(IB)IB \\ \Box, \Diamond &: (IIB)(IB)IB \end{aligned}$$

Modal terms and **modal formulas** are now defined as follows:

$t ::= p \mid \dot\bot \mid \dot\top \mid \dot\neg t \mid t \mathbin{\dot\wedge} t \mid t \mathbin{\dot\vee} t \mid t \mathbin{\dot\rightarrow} t \mid t \mathbin{\dot\equiv} t \mid \Box rt \mid \Diamond rt$	modal terms
$s ::= tx \mid rxx \mid \forall t$	modal formulas

Note that modal terms and modal formulas are λ-free although they may use the quantifier $\forall_I$.

The semantics of the modal constants is expressed by the equations in Figure 9.1. Except for the **modalities** $\Box$ and $\Diamond$, the semantics of the modal constants is easy to understand: they lift the propositional constants from Booleans (type B) to properties (type IB). The modalities are specialized quantifiers that can be explained as follows:

- $\Box rpx$ holds if every r-successor of x satisfies p.
- $\Diamond rpx$ holds if there is an r-successor of x that satisfies p.

A **modal interpretation** is a logical interpretation that satisfies the equations defining the modal constants. In the context of modal logic, we consider only modal interpretations and consider the modal constants as a additional logical constants. All names that are not constants (logical or modal) are called variables. We assume that p and q only range over names that are different from $\dot\bot$ and $\dot\top$.

We use M to refer to the set of equations in Figure 9.1. The next proposition states that our modal logic is a translational fragment of PLN.

Proposition 9.1.1 (First-Order Translation) For every modal formula s there exists a PLN-formula t such that $\mathrm{M} \vdash s = t$.

$$\begin{aligned}
\dot{\bot} &= \lambda x.\,\bot\\
\dot{\top} &= \lambda x.\,\top\\
\dot{\neg} &= \lambda px.\,\neg px\\
\dot{\wedge} &= \lambda pqx.\,px \wedge qx\\
\dot{\vee} &= \lambda pqx.\,px \vee qx\\
\dot{\rightarrow} &= \lambda pqx.\,px \rightarrow qx\\
\dot{\equiv} &= \lambda pqx.\,px \equiv qx\\
\Box &= \lambda rpx.\,\forall y.\,rxy \rightarrow py\\
\Diamond &= \lambda rpx.\,\exists y.\,rxy \wedge py
\end{aligned}$$

Figure 9.1: Equations defining the modal constants (set M)

Proof Let s be a modal formula. First we eliminate the modal constants by applying the equations in M. Then we apply β-reduction and eliminate all β-redexes. Next we apply η-expansion to subterms of the form $\forall p$ to obtain $\forall x.px$. Finally we eliminate the propositional constants $\bot$, $\top$, $\rightarrow$, and $\equiv$, which is straigthforward. ■

Notationally, the modal constants (except $\dot{\bot}$ and $\dot{\top}$) act as operators that take their arguments before $=$:

$$\begin{array}{ccc}
 & = & \\
 & \dot{\equiv} & \\
 & \dot{\rightarrow} & \\
 & \dot{\vee} & \\
 & \dot{\wedge} & \\
\dot{\neg} & \Box & \Diamond
\end{array}$$

The modal operators $\dot{\neg}$, $\Box$, and $\Diamond$ group to the right. For instance, the formula $\dot{\neg}\Box rt = \Diamond r\dot{\neg}t$ is to be read as $(\dot{\neg}(\Box rt)) = (\Diamond r(\dot{\neg}t))$.

Exercise 9.1.2 Translate the following modal formulas into first-order formulas:

a) $(p \mathbin{\dot{\vee}} q)x$

b) $\forall(p \mathbin{\dot{\rightarrow}} q)$

c) $(\Box r(p \mathbin{\dot{\rightarrow}} \Diamond rp))x$

d) $(\Box r\Diamond r\dot{\neg}p)x$

We call modal formulas of the form px or rxy **primitive**. Note that the formulas $\dot{\bot}x$ and $\dot{\top}x$ are not primitive.

$\mathsf{married\ mike\ mary}$
$\mathsf{haschild\ mike\ rob}$
$\mathsf{owns\ mike\ x}$
$\mathsf{BMW\ x}$
$\mathsf{man\ mike}$
$\forall(\mathsf{BMW} \dot{\to} \mathsf{car})$
$\forall(\mathsf{man} \dot{\to} \mathsf{person})$
$\forall(\mathsf{man} \dot{\to} \mathsf{male})$
$\forall(\mathsf{parent} \dot{\equiv} \mathsf{person} \dot{\wedge} \Diamond\mathsf{haschild}\ \dot{\top})$
$\forall(\mathsf{person} \dot{\to} \Box\mathsf{haschild\ person})$

Figure 9.2: A knowledge base

9.2 A Knowledge Base

Figure 9.2 shows an example of a knowledge base that demonstrates the use of modal logic as a knowledge representation language. Every line is a modal formula. The types of the variables are as follows:

- married, haschild, and owns are relation variables.
- BMW, man, car, person, parent are property variables.
- mike, mary, rob, x are individual variables.

The primitive formulas of the knowledge base provide facts about individuals. The non-primitive formulas provide so-called terminological information about predicates like person and haschild. If K is the knowledge base (i.e, a set of modal formulas) and s is a modal formula, we say that K **semantically entails** s if $K, \mathrm{M} \models s$ (read K, M as $K \cup \mathrm{M}$). For instance:

$$K, \mathrm{M} \models \mathsf{person\ mike}$$
$$K, \mathrm{M} \models \mathsf{male\ mike}$$
$$K, \mathrm{M} \models \mathsf{person\ rob}$$

Intuitively, mary should be person as well, but logically, there is nothing in the knowledge base that enforces this property. In fact:

$$K, \mathrm{M} \not\models \mathsf{person\ mary}$$
$$K, \mathrm{M} \not\models \neg\ \mathsf{person\ mary}$$

Figure 9.3 lists all primitive formulas that are semantically entailed by the knowledge base in Figure 9.2 in a graphical format we call a **transition graph**.

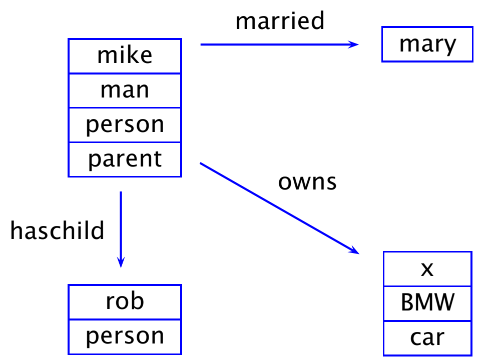


Figure 9.3: A set of primitive formulas represented as a transition graph

There are many computational services one can associate with a knowlege base. For instance, one can use modal terms as a query language. Given a knowlege base K and a modal term t, this services yields all names x such that K semantically entails tx. Here are example queries:

1. person yields mike and rob.
2. car yields x.
3. $\Diamond$haschild$\top$ yields mike.
4. $\Diamond$married person yields nothing.

Exercise 9.2.1 List all the primitive formulas given by the transition graph in Figure 9.3.

9.3 Kripke Sets

A **Kripke set** is a set of primitive modal formulas.[1] Finite Kripke sets can be represented as transition graphs. The Kripke set in Figure 9.3 is tree-like. The Kripke set $\{rxy, ryz, rzx\}$ has a cyclic transition graph:

$$\begin{tikzcd} & y \arrow[d, "r"] \\ x \arrow[ur, "r"] & \\ & z \arrow[ul, "r"] \end{tikzcd}$$

We will use Kripke sets as interpretations for modal formulas. Note that every non-empty Kripke set contains at least one individual variable. To avoid problems with the empty Kripke set, we fix an individual variable x_0 and call it the

[1] For experts: Kripke sets can be seen as a syntactic variant of Kripke structures where the states are individual variables. We can also see Kripke sets as Herbrand models for modal logic.

default variable. We define the **carrier** CA of a set of modal formulas A as follows: $CA = \{x_0\}$ if A is empty, and $CA = \{\, x \in \mathcal{N}A \mid x : I\,\}$ if A is non-empty. An interpretation **$\mathcal{I}$ agrees with a Kripke set K** if it is modal and satisfies the following conditions:

1. $\mathcal{I}I = CK$
2. $\mathcal{I}x = x$ for all $x \in CK$
3. $\mathcal{I}px = (px \in K)$ for all $p : IB$ and all $x \in CK$
4. $\mathcal{I}rxy = (rxy \in K)$ for all $r : IIB$ and all $x, y \in CK$

Note that for every Kripke set there are infinitely many interpretations that agree with it. If an interpretation agrees with a Kripke set K, the interpretations of property and relation variables are determined by K. Moreover, the interpretations of the invidual variables in CK is determined by K. As it comes to other invidual variables, K only requires that they be interpreted as elements of CK.

It turns out that all interpretations that agree with a given Kripke set K also agree on the interpretation of all modal formulas. We can thus understand a Kripke set as a specialized interpretation for modal formulas. Given a Kripke set K, we can define an **evaluation function** $\hat{K}$ such that $\hat{K}s = \hat{\mathcal{I}}s$ for all modal formulas and all interpretations $\mathcal{I}$ that agree with K. Figure 9.4 defines $\hat{K}$ by recursion on the size of formulas. Note that $\hat{K}$ is computable if K is finite. We can say that finite Kripke sets are a computable semantics for modal formulas.

Proposition 9.3.1 If an interpretation $\mathcal{I}$ agrees with a Kripke set K, then $\hat{\mathcal{I}}s = \hat{K}s$ for all modal formulas s.

Proof By induction on the size of s. ■

A Kripke set K **satisfies** a modal formula s if $\hat{K}s = 1$. A Kripke set **satisfies** a set of modal formulas if it satisfies every formula of the set. A **Kripke model** of a formula [set of modal formulas] is a Kripke set that satisfies the formula [set]. A formula [set of formulas] is **modally satisfiable** if it is satisfied by at least one modal interpretation, and **modally valid** if it is satisfied by all modal interpretations.

Proposition 9.3.2 Let s be a modal formula. Then:

1. If s is satisfied by a Kripke set, then s is modally satisfiable.
2. If s is modally valid, then s is satisfied by every Kripke set.

Proof Follows from Proposition 9.3.1 and the fact that for every Kripke set there is an interpretation agreeing with it. ■

$$\begin{aligned}
\hat{K}(px) &= (px \in K)\\
\hat{K}(\bot x) &= 0\\
\hat{K}(\dot{\top} x) &= 1\\
\hat{K}((\dot{\neg} t)x) &= \neg \hat{K}(tx)\\
\hat{K}((t_1 \mathbin{\dot{\wedge}} t_2)x) &= (\hat{K}(t_1x) \wedge \hat{K}(t_1x))\\
\hat{K}((t_1 \mathbin{\dot{\vee}} t_2)x) &= (\hat{K}(t_1x) \vee \hat{K}(t_1x))\\
\hat{K}((t_1 \mathbin{\dot{\to}} t_2)x) &= (\hat{K}(t_1x) \Longrightarrow \hat{K}(t_1x))\\
\hat{K}((t_1 \doteq t_2)x) &= (\hat{K}(t_1x) = \hat{K}(t_1x))\\
\hat{K}(\Box rtx) &= (\forall y\colon\ rxy \in K \Longrightarrow \hat{K}(ty))\\
\hat{K}(\Diamond rtx) &= (\exists y\colon\ rxy \in K \wedge \hat{K}(ty))\\
\hat{K}(\forall t) &= (\forall x \in CK\colon\ \hat{K}(tx))\\
\hat{K}(rxx) &= (rxx \in K)
\end{aligned}$$

Figure 9.4: Definition of the evaluation function $\hat{K}$ for a Kripke set K

Later we will show that every modally satisfiable modal formula is satisfied by a finite Kripke set. The finiteness part of this important result is known as *finite model property*.

Due to Proposition 9.3.2 we can use Kripke sets to prove that a modal formula is modally satisfiable or not not modally valid. As an example, consider the modal formula $(\Box r(p \mathbin{\dot{\vee}} q) \to \Box rp \mathbin{\dot{\vee}} \Box rq)x$. Is it satisfied by every modal interpretation? The Kripke set $\{rxy,\ py,\ rxz,\ qz\}$

$$\begin{tikzcd}
 & y,p \\
x \arrow[ur, "r"] \arrow[dr, "r"'] & \\
 & z,q
\end{tikzcd}$$

shows that this is not the case since it doesn't satisfy the formula.

Exercise 9.3.3 For each of the following formulas find a finite Kripke set satisfying it.

a) $(\Box r(p \mathbin{\dot{\vee}} q) \mathbin{\dot{\to}} \Box rp \mathbin{\dot{\vee}} \Box rq)x$

b) $\neg(\Box r(p \mathbin{\dot{\vee}} q) \mathbin{\dot{\to}} \Box rp \mathbin{\dot{\vee}} \Box rq)x$

c) $\forall(\Diamond rp \mathbin{\dot{\wedge}} (p \mathbin{\dot{\to}} \Diamond rq))$

Exercise 9.3.4 For each of the following formulas find a Kripke set that doesn't satisfy the formula. Your sets should employ only two individual variables. First draw the sets as transition graphs and then list them explicity.

a) $\Box r(p \dot{\vee} q) \;=\; \Box rp \dot{\vee} \Box rq$

b) $\Diamond r(p \dot{\wedge} q) \;=\; \Diamond rp \dot{\wedge} \Diamond rq$

Exercise 9.3.5 Which of the following formulas is modally valid? If the formula is not valid, find a Kripke set that doesn't satisfy it.

a) $\dot{\forall}\,(\Diamond r(p \dot{\wedge} q) \;\dot{\rightarrow}\; \Diamond rp \dot{\wedge} \Diamond rq)$

b) $\dot{\forall}\,(\Box r(p \dot{\vee} q) \;\dot{\rightarrow}\; \Box rp \dot{\vee} \Box rq)$

c) $\Box r \dot{\top} = \dot{\top}$

d) $\Diamond r \dot{\bot} = \dot{\top}$

e) $\Diamond r \dot{\bot} = \dot{\bot}$

f) $\Diamond r \dot{\top} = \Box rp \dot{\rightarrow} \Diamond rp$

Exercise 9.3.6 (Herbrand Models) In his dissertation submitted in 1929, Jacques Herbrand introduces syntactic models for first-order logic that are now known as Herbrand models. Kripke sets are in fact Herbrand models for modal logic. Herbrand models are interesting for first-order logic since a first-order formula is satisfiable if and only if it is satisfied by a Herbrand model. Herbrand models can be constructed with the tableau method.

a) A Herbrand model for propositional logic is a set of Boolean variables. Let H be a set of Boolean variables. Define the evaluation function $\hat{H}$ that assigns to every propositional formula a Boolean value. Hint: Follow the definition of the evaluation function for Kripke sets and the grammar for propositional formulas.

b) Explain how the tableau method can be used to construct finite Herbrand models for satisfiable propositional formulas.

c) Find a Herbrand model for the propositional formula $\neg x \wedge (x \rightarrow y \vee z)$.

d) A Herbrand model for PLN is a set of PLN-formulas of the form $px_1 \dots x_n$ where $n \geq 0$. Let H be such a set. Define the evaluation function $\hat{H}$ that assigns to every PLN-formula a Boolean value.

e) Find finite Herbrand models for the following PLN-formulas:

   i) $rxy \wedge qx \wedge (rxy \rightarrow rxx)$

   ii) $\forall y \exists y.\, rxy$

$$\begin{array}{rcl}
p \mathrel{\dot{\to}} q & = & \dot{\neg} p \mathrel{\dot{\vee}} q \\
p \mathrel{\dot{\equiv}} q & = & (p \mathrel{\dot{\to}} q) \mathrel{\dot{\wedge}} (q \mathrel{\dot{\to}} p) \\
\dot{\neg}\dot{\neg} p & = & p \\
\dot{\neg}(p \mathrel{\dot{\wedge}} q) & = & \dot{\neg} p \mathrel{\dot{\vee}} \dot{\neg} q \\
\dot{\neg}(p \mathrel{\dot{\vee}} q) & = & \dot{\neg} p \mathrel{\dot{\wedge}} \dot{\neg} q \\
\dot{\neg}\Box r t & = & \Diamond r \dot{\neg} t \\
\dot{\neg}\Diamond r t & = & \Box r \dot{\neg} t \\
\Box r (p \mathrel{\dot{\wedge}} q) & = & \Box r p \mathrel{\dot{\wedge}} \Box r q \\
\Diamond r (p \mathrel{\dot{\vee}} q) & = & \Diamond r p \mathrel{\dot{\vee}} \Diamond r q \\
p = q & \equiv & \forall (p \mathrel{\dot{\equiv}} q) \\
\neg p x & \equiv & (\dot{\neg} p) x
\end{array}$$

Figure 9.5: Some equations deducible from M

9.4 Some Deducible Facts

Figure 9.4 shows some interesting equations that are deducible from M.

Exercise 9.4.1 Find tableau proofs for the equations in Figure 9.4.

A modal term is **propositional** if it doesn't contain $\Box$ or $\Diamond$. If tx is a modal formula such that t is a propositional modal term, then $M \vdash tx$ if and only if t seen as a propositional formula is deducible. To turn a propositional modal term into a propositional formula, do the following:

1. Replace the dotted modal connectives with their propositional counterparts.
2. Replace the property variables with Boolean variables.

Proposition 9.4.2 Let t be a propositional modal term and s be a propositional formula that corresponds to t. Then $\mathrm{M} \vdash \forall t$ iff $\vdash s$.

Proof Let $x : B$. It suffices to show $\mathrm{M} \vdash tx$ iff $\vdash s$. By the first-order translation (Proposition 9.1.1) we obtain a first-order formula u such that $\mathrm{M} \vdash tx$ iff $\vdash u$. The formula u is like s except that the Boolean variables appear as formulas px. Hence $\vdash u$ follows from $\vdash s$ by Sub. The other direction follows by Sub and β-reduction. The substitution θ is chosen such that $\theta p = \lambda x.b$ for every property variable p occurring in t and the Boolean variable b corresponding to p. ■

Exercise 9.4.3 For each of the following formulas s, show $\mathrm{M} \vdash s$ with a tableau proof.

a) $\Diamond r \dot{\top} = \Box r p \dot{\rightarrow} \Diamond r p$

b) $\dot{\neg} \Box r t = \Diamond r \dot{\neg} t$

c) $\forall (\Box r (p \dot{\rightarrow} q) \dot{\rightarrow} \Box r p \dot{\rightarrow} \Box r q)$

9.5 Modal Completeness

We will prove a completeness result for formulas involving modal constants. The result builds on the completeness result for PLN (Theorem 6.6.7). To show the result, we need some notions and results that apply to simple type theory in general.

9.5.1 Interderivability

Let S_1, S_2 be sequents. We write S_1 / S_2 if the proof step $(\{S_1\}, S_2)$ is derivable in the basic proof system. Moreover, we write $S_1 \| S_2$ and say that S_1 and S_2 are **interderivable** if S_1 / S_2 and S_2 / S_1.

Proposition 9.5.1 Let S_1 / S_2. Then:

1. If S_1 is deducible, then S_2 is deducible.
2. If S_1 is valid, then S_2 is valid.

Proof The first claim is obvious. The second claim holds since the basic proof system is sound. ■

Proposition 9.5.2 Here are some facts that we expressed before with proof rules. The name Gen stands for generalization.

- **Contra** $A \dot{\vdash} s \,\|\, A, \neg s \dot{\vdash} \bot$
- **Ded** $A \dot{\vdash} s \rightarrow t \,\|\, A, s \dot{\vdash} t$
- **Gen** $A \dot{\vdash} \forall x.s \,\|\, A \dot{\vdash} s$ if $x \notin \mathcal{N} A$
- **Rep** $A, s \dot{\vdash} \bot \,\|\, A, t \dot{\vdash} \bot$ if $A \vdash s{=}t$
- **Rep** $A \dot{\vdash} s \,\|\, A \dot{\vdash} t$ if $A \vdash s{=}t$

Exercise 9.5.3 Prove Proposition 9.5.2.

9.5.2 Theories

A **theory** is a pair (S, F) such that S is a finite set of formulas and F is a set of formulas. We see S as the specification of the theory and F as the formulas of the theory. Here are examples of theories:

- *Propositional logic.* Here $S = \emptyset$ and F is the set of all propositional formulas.
- *First-order logic.* Here $S = \emptyset$ and F is the set of all first-order formulas.
- *Modal logic.* Here $S = \mathrm{M}$ and F is the set of all modal formulas.

A theory (S, F) is **complete** if $S, A \vDash \bot \implies S, A \vdash \bot$ for every finite $A \subseteq F$. A theory (S, F) is **decidable** if $S, A \vdash \bot$ is dedcidable for finite $A \subseteq F$. We have shown that propositional logic is complete and decidable. It is well-known that first-order logic is complete (Gödel 1929) and undecidable (Church 1936). We will show that modal logic is complete and decidable.

For a theory (S, F), we define the **equivalence closure of F under S** as follows:

$$[F]_S \;:=\; \{\, s \mid \exists t \in F\colon\; S \vdash s{\equiv}t \,\}$$

The equivalence closure $[F]_S$ consists of all formulas that under S are deductively equivalent to a formula in F.

Proposition 9.5.4 Let (S, F) be a complete theory. Then $(S, [F]_S)$ is complete.

Proof Let $A = \{s_1, \dots, s_n\} \subseteq [F]_S$ and $S, A \vDash \bot$. We show $S, A \vdash \bot$. Let $\{t_1, \dots, t_n\} \subseteq F$ such that $S \vdash s_i{=}t_i$ for all i. Then $S, t_1, \dots, t_n \vDash \bot$ by Rep. Hence $S, t_1, \dots, t_n \vdash \bot$ by completeness of (S, F). Thus $S, A \vdash \bot$ by Rep. ∎

9.5.3 Pure First-Order Logic

A formula is called a **pure first-order formula** if it can be obtained with the following grammar:

$$s \;::=\; p\,x \dots x \mid \bot \mid \top \mid \neg s \mid s \to s \mid s \wedge s \mid s \vee s \mid s \equiv s \mid \forall x.s \mid \exists x.s$$

where $x : I$ and $p : I \dots IB$ are variables

Compared to general first-order formulas, there are three restrictions:

1. Only two sorts B and I.
2. No functional names $f : I \dots II$.
3. No equations $t_1 =_I t_2$.

The terms $p(fx)$ and $x = y$ where $x, y : I$, $f : II$, and $p : IB$ are examples of first-order formulas that are not pure. One can show that pure first-order logic is still undecidable, even if further restricted to a single predicate variable $p : IIB$ (see [8]). We use **PFO** to denote the set of all pure first-order formulas.

Proposition 9.5.5 $[\mathrm{PFO}]_{\mathrm{M}}$ contains all modal formulas.

Proof The claim is a reformulation of Proposition 9.1.1. ∎

Recall the definition of PLN-formulas in § 6.6. Let **PLN** be the set of all PLN-formulas. Clearly, every pure PLN-formula is a pure first-order formula. However, the opposite is not true. For instance, PLN-formulas do not admit $\perp$ and $\rightarrow$. Moreover, the definition of PLN doesn't admit all individual names as local names. Theorem 6.6.7 states that a subset of PLN that further restricts the use of individual names is complete. We will now show that this completeness result extends to pure first-order logic. We assume that every individual name is in $\mathcal{P} \cup \mathcal{V}$ (this is consistent with the definition of PLN).

Proposition 9.5.6 $\text{PFO} \subseteq [\text{PLN}]_\emptyset$

Exercise 9.5.7 Show that the closure $[\text{PLN}]_\emptyset$ contains $\perp$ and is closed under $\rightarrow$. Moreover, show that $\forall x.s$ is in the closure if s is in the closure and $x : I$ is an individual name (possibly a parameter).

Proposition 9.5.8 (Renaming) Let s be a formula and θ and θ' be substitutions such that $\theta'(\theta s) = s$. Then $\dot{\vdash}\ \theta s \ \|\ \dot{\vdash}\ s$.

Proof Follows with Sub. ∎

The proposition gives us the possibility to rename the free individual names of a formula to those that are allowed by the subset of PLN for wich Theorem 6.6.7 establishes completeness. If the renaming is invertible, it doesn't affect deducibility and validity. We thus know that $(\emptyset, \text{PLN})$ is complete. The rest is easy, since every pure first-order formula is in the closure $[\text{PLN}]_\emptyset$. Hence the completeness of pure first-order logic follows with Proposition 9.5.4.

Theorem 9.5.9 (Completeness) Pure first-order logic is complete.

9.5.4 Defined Constants

The equational definition of constants can be expressed with a substitution that maps the constants to their defining terms. We will only consider non-recursive definitions. Hence we can require that the defined constants do not occur in the defining terms. This property is equivalent to a property called idempotence. A substitution θ is **idempotent** if $\theta x = \theta(\theta x)$ for all names x. We use $[\theta]$ to denote the set of all equations $x{=}\theta x$ such that x is a name and $\theta x \neq x$. We call a substitution θ **finite** if $[\theta]$ is a finite set of equations. Convince yourself that the equations in Figure 9.1 yield a finite and idempotent substitution θ such that $[\theta] = \text{M}$. This substitution eliminates the modal constants when it is applied.

Proposition 9.5.10 (Defined Constants) Let θ be finite and idempotent. Then $A, [\theta] \,\dot{\vdash}\, s \ \|\ \theta A \,\dot{\vdash}\, \theta s$.

Proof The direction from left to right follows with Sub if $\vdash \theta[\theta]$. This is the case since $\theta[\theta]$ contains only trivial equations $u{=}u$ since θ is idempotent. The other direction follows with Rep since $[\theta] \vdash \theta t{=}t$ for all terms t. ■

Lemma 9.5.11 Let (S, F) be complete and θ be a finite and idempotent substitution such that $\theta S = S$, $\theta F \subseteq [F]_S$, and $\theta\bot = \bot$. Then $(S \cup [\theta], F)$ is complete.

Proof Let $S, [\theta], A \vDash \bot$ and $A \subseteq F$. We show $S, [\theta], A \vdash \bot$. Since $\theta S = S$ and $\theta\bot = \bot$, we have $S, \theta A \vDash \bot$ by Proposition 9.5.10. Since $\theta A \subseteq \theta F \subseteq [F]_S$ and S is complete for $[F]_S$ (Proposition 9.5.4), we have $S, \theta A \vdash \bot$. Since $\theta S = S$ and $\theta\bot = \bot$, we have $S, [\theta], A \vdash \bot$ by Proposition 9.5.10. ■

9.5.5 The Main Result

The main result of this section says that we are complete for formulas that can be translated into pure first-order logic with the equations defining the modal constants.

Theorem 9.5.12 (Modal Completeness) $(\mathrm{M}, [\mathrm{PFO}]_\mathrm{M})$ is complete.

Proof Let θ be the finite and idempotent substitution such that $M = [\theta]$. By Proposition 9.5.4 it suffices to show that $(\mathrm{M}, \mathrm{PFO})$ is complete. Let $s \in \mathrm{PFO}$. By Lemma 9.5.11 it suffices to show that $\theta s \in [\mathrm{PFO}]_\mathrm{M}$. By checking the types of the modal constants one sees that θ can only replace $\bot$ and $\dagger$ in s. Moreover, $\bot$ and $\dagger$ can only occur as $\bot x$ and $\dagger x$ in s. Hence it suffices to show that $(\lambda x.\bot)x$ and $(\lambda x.\top)x$ are deductively equivalent to pure first-order formulas. This is obvious. ■

Corollary 9.5.13 Modal logic is complete (i.e., the theory consisting of M and the set of modal formulas).

Proof Follows from Theorem 9.5.12 and Proposition 9.5.5. ■

9.6 Modal Refutability

Let A be a finite set of modal formulas. We say that A is **modally refutable** if $\mathrm{M}, A \vdash \bot$.

Proposition 9.6.1 A finite set of modal formulas is modally unsatisfiable if and only if it is modally refutable.

Proof Follows with soundness and modal completeness (Corollary 9.5.13). ■

Many modal decision problems can be reduced to modal refutability. The next proposition states some of them.

Proposition 9.6.2 (Reduction to Modal Refutability)

1. $\mathrm{M}, A \mathrel{\dot{\vdash}} tx \;\;\|\;\; A, (\dot{\neg} t)x, \mathrm{M} \mathrel{\dot{\vdash}} \bot$
2. $\mathrm{M}, A \mathrel{\dot{\vdash}} \forall t \;\;\|\;\; A, (\dot{\neg} t)x, \mathrm{M} \mathrel{\dot{\vdash}} \bot$ if $x \notin \mathcal{N}A \cup \mathcal{N}t$
3. $\mathrm{M}, A \mathrel{\dot{\vdash}} t_1{=}t_2 \;\;\|\;\; A, \dot{\neg}(t_1 \doteq t_2)x, \mathrm{M} \mathrel{\dot{\vdash}} \bot$ if $x \notin \mathcal{N}A \cup \mathcal{N}(t_1 \doteq t_2)$

Proof Follows with Proposition 9.5.2 and some of the deducible equations in Figure 9.4. ■

Note that the interderivabilities stated by Proposition 9.6.2 hold in general, that is, A may contain non-modal formulas and t, t_1, t_2 may be non-modal terms of type IB.

Later we will show that modal logic enjoys the **finite model property**: a finite set of modal formulas is modally satisfiable if and only if it is satisfied by a finite Kripke set. The next proposition states an interesting connection between completeness and the finite model property.

Proposition 9.6.3 If modal logic has the finite model property, then modal refutability is decidable.

Proof Let A range over finite sets of modal formulas. We know that $\mathrm{M}, A \vdash \bot$ is semi-decidable (since $\vdash$ is). Hence it suffices to show that $\mathrm{M}, A \nvdash \bot$ is semi-decidable. By modal completeness it suffices to show that $\mathrm{M}, A \nvDash \bot$ is semi-decidable. This is the case since the finite model property holds and the finite Kripke sets are recursively enumerable. ■

Exercise 9.6.4 Reduce the following problems to modal refutability (analogous to Proposition 9.6.2). Prove the correctness of your reductions.

a) $\mathrm{M}, A \vdash \neg tx$

b) $\mathrm{M}, A \vdash \neg\forall t$

c) $\mathrm{M}, A \vdash \exists t$

d) $\mathrm{M}, A \vdash t_1{\neq}t_2$ where $t_1, t_2 : IB$

e) $\mathrm{M}, A \vdash \neg rxy$

f) $\mathrm{M}, A \vdash \forall x.\, t_1x \to t_2x$

9.7 A Tableau System for Modal Formulas

We define **negation normal** modal terms as follows:

$$t ::= p \mid \dot{\neg} p \mid t \mathbin{\dot{\wedge}} t \mid t \mathbin{\dot{\vee}} t \mid \Box rt \mid \Diamond rt$$

A modal formula is **negation normal** if every modal terms it contains is negation normal.

Proposition 9.7.1 For every modal term s there is a negation normal modal term t such that $\mathrm{M} \vdash s = t$.

Exercise 9.7.2 Give a procedure that yields for every modal term an equivalent negation normal term. Write the procedure as a system of terminating equations.

A set A of modal formulas is **locally consistent** if there is no formula tx such that both tx and $(\dot{\neg} t)x$ are in A.

Proposition 9.7.3 A locally inconsistent set of modal formulas is modally unsatisfiable.

Our goal is a terminating tableau system that for a finite set A of modal formulas either constructs a finite Kripke set satisfying A or shows that A is modally refutable. We will only consider negation normal modal formulas.

We can see the tableau system as a recursive procedure that attempts to construct a satisfying Kripke set for a set of modal formulas. The procedure emulates the equations defining the evaluation function for Kripke sets (see Figure 9.4). The equations tell us that a formula holds if certain smaller formulas derived from it hold. The procedure adds the derived formulas to the set (called the branch in tableau speak). If this saturation process terminates with a locally consistent set, then the primitive formulas of this set constitute a satisfying Kripke set for the initial set.

The outlined plan leads to the tableau system in Figure 9.6. The A in the side conditions is the set of all formulas on the branch. All rules but $\mathcal{R}_\Diamond$ are obvious from the equations of the evaluation function. For $\mathcal{R}_\Diamond$, the two side conditions need explanation. The side condition $y \notin \mathcal{N}A$ is needed for falsification soundness (see below) and is closely related to the freshness condition of the **Exists** rule in Figure 6.15. The second side condition of $\mathcal{R}_\Diamond$ is needed so that the rule cannot be applied repeatedly to the same $\Diamond$-formula.

9.7.1 Falsification Soundness

A tableau system is **falsification sound** if the existence of a closed tableau for a set A implies that A is unsatisfiable. For the falsification soundness of the modal tableau system we have to show two conditions:

1. If $\mathcal{R}_\neg$ applies to A, then A is modally unsatisfiable.
2. If a rule applies to A and adds formulas and A is modally satisfiable, then we can obtain a modal interpretation that satisfies A and the added formulas. In the case of $\mathcal{R}_\vee$ only one of the two alternatives must satisfy this requirement.

$$\mathcal{R}_\neg\ \frac{(\dot\neg p)x}{\emptyset}\ px\in A \qquad \mathcal{R}_\wedge\ \frac{(t_1\dot\wedge t_2)x}{t_1x,\ t_2x} \qquad \mathcal{R}_\vee\ \frac{(t_1\dot\vee t_2)x}{t_1x\mid t_2x} \qquad \mathcal{R}_\forall\ \frac{\forall t}{tx}\ x\in\mathcal{C}A$$

$$\mathcal{R}_\Box\ \frac{\Box rtx}{ty}\ rxy\in A \qquad \mathcal{R}_\Diamond\ \frac{\Diamond rtx}{rxy,\ ty}\ y\notin\mathcal{N}A\ \wedge\ \neg\exists z\colon\ rxz,\ tz\in A$$

Figure 9.6: The basic modal tableau system $\mathcal{T}_\mathrm{M}$

$$\frac{}{A,px,(\dot\neg p)x\vdash\bot} \qquad \frac{A,t_1x,t_2x\vdash\bot}{A,(t_1\dot\wedge t_2)x\vdash\bot} \qquad \frac{A,t_1x\vdash\bot\qquad A,t_2x\vdash\bot}{A,(t_1\dot\vee t_2)x\vdash\bot}$$

$$\frac{A,tx\vdash\bot}{A,\forall t\vdash\bot} \qquad \frac{A,ty\vdash\bot}{A,rxy,\Box rtx\vdash\bot} \qquad \frac{A,rxy,ty\vdash\bot}{A,\Diamond rtx\vdash\bot}\ y\notin\mathcal{N}A\cup\mathcal{N}(\Diamond rtx)$$

Proviso: All proof steps assume $M\subseteq A$.

Figure 9.7: Proof steps for the modal tableau rules

Make sure you understand why the rules in Figure 9.6 satisfy the two conditions for falsification soundness, and why the two conditions yield falsification soundness.

9.7.2 Refutation Soundness

A set A of formulas is **refutable** if the sequent $A\ \dot\vdash\ \bot$ is deducible. A tableau system is **refutation sound** if the existence of a closed tableau for a set A implies that A is refutable. Note that a tableau system is falsification sound if it is refutations sound. The best way to establish refutation soundness is to show that the proof steps corresponding to the tableau rules are derivable. The proof steps for the modal tableau rules are shown in Figure 9.7.

Proposition 9.7.4 The proof steps for the modal tableau rules are derivable.

Exercise 9.7.5 Show that the proof steps for $\dot\vee$ and $\Box$ are derivable.

Exercise 9.7.6 Extend the tableau rules to modal formulas that are not negation normal. Give the corresponding proof steps.

Exercise 9.7.7 Refute the following sets with modal tableau proofs.

a) $\Diamond r(q \dot\vee \dot\neg q)x,\ \ \Box rpx,\ \ \Box r(\dot\neg p)x$

b) $\Box r(q \dot\wedge \dot\neg q)x,\ \ (\Diamond r(\dot\neg p) \dot\vee \Diamond rp)x$

c) $(\dot\neg(\Diamond r(q \dot\vee \dot\neg q) \dot\equiv \Box rp \dot\to \Diamond rp))x$

Exercise 9.7.8 Prove the validity of the following modal formulas by reduction to modal refutability (Proposition 9.6.2) and modal tableaux.

a) $\forall(\Box r(p \dot\wedge q) \dot\to \Box rp \dot\wedge \Box rq)$

b) $\forall(\Diamond rp \dot\vee \Diamond rq \dot\to \Diamond r(p \dot\vee q))$

c) $\forall(\Diamond r(p \dot\vee q) \dot\to \Diamond rp \dot\vee \Diamond rq)$

9.7.3 Verification Soundness

Our tableau system is **verification sound** if every set of negation normal modal formulas to which no tableau rule applies is modally satisfiable. We define a class of sets that includes the sets just described. A set A of negation normal modal formulas is **evident** if it satisfies the following conditions:

1. If $(\dot\neg p)x \in A$, then $px \notin A$.
2. If $(t_1 \dot\wedge t_2)x \in A$, then $t_1x \in A$ and $t_2x \in A$.
3. If $(t_1 \dot\vee t_2)x \in A$, then $t_1x \in A$ or $t_2x \in A$.
4. If $\forall t \in A$ and $x \in \mathcal{C}A$, then $tx \in A$.
5. If $\Box rtx \in A$ and $rxy \in A$, then $ty \in A$.
6. If $\Diamond rtx \in A$, then there exists a y such that $rxy \in A$ and $ty \in A$.

Proposition 9.7.9 Let A be a set of negation normal modal formulas to which no rule of $\mathcal{T}_M$ applies. Then A is evident.

Proof Straightforward. Check the definition and check the tableau rules. ∎

Proposition 9.7.10 (Model Existence) Let A be evident. Then the largest Kripke set $K \subseteq A$ satisfies A.

Proof Let K be the largest Kripke set such that $K \subseteq A$. We show by induction on the size of formulas that K satisfies every formula $s \in A$. Let $s \in A$. Case analysis.

If s is a primitive formula, the claim is trivial since $s \in K$.

If $s = \neg t$, then t is a primitive formula and $t \notin K$. Hence $\hat{K}t = 0$ and $\hat{K}s = 1$.

If $s = (t_1 \dot\wedge t_2)x$, then t_1x and t_2x are in A. By induction $\hat{K}(t_1x) = \hat{K}(t_2x) = 1$. Hence $\hat{K}s = 1$.

If $s = \Diamond rtx$, then there exists a y such that rxy and ty are in A. By induction we have $\hat{K}(ty) = 1$. Since $rxy \in K$, we have $\hat{K}s = 1$.

The remaing cases are similiar. ■

Exercise 9.7.11 Use the tableau rules to construct an evident set that contains the modal formula $((\Diamond rp \dot\wedge \Diamond rq) \dot\wedge \Box r(\dot\neg p \dot\vee \dot\neg q))x$. Explain why the evident set gives you a Kripke set that satisfies the formula.

9.8 Termination

A **clause** is a finite set of negation normal modal terms. A clause is **$\Diamond$-free** if it doesn't contain the modal constant $\Diamond$, and **$\forall$-free** if it doesn't contain the logical constant $\forall$. We write $A \to C$ if A and C are clauses and C can be obtained from A by a basic modal tableau rule different from $\mathcal{R}_\neg$. We call a pair (A, C) such that $A \to C$ an **expansion step**.

Example 9.8.1 Here is diverging $\mathcal{T}_M$-derivation starting from a satisfiable clause.

$\forall(\Diamond rp)$	initial clause
$\Diamond rpx_0$	$\mathcal{R}_\forall$
$rx_0y,\ py$	$\mathcal{R}_\Diamond$
$\Diamond rpy$	$\mathcal{R}_\forall$
$ryz,\ pz$	$\mathcal{R}_\Diamond$
$\dots$	

The initial clause is satisfied by the Kripke set $\{rxx, px\}$. $\mathcal{T}_M$ fails to construct this set since it cannot introduce cyclic transitions rxx. □

The basic modal tableau system $\mathcal{T}_M$ terminates for $\forall$-free clauses. Hence it constitutes a decision procedure for the modal satisfiability of such clauses. If we constrain $\mathcal{R}_\Diamond$ with a further side condition, we obtain a system that terminates for all clauses A. However, we have to rework the proof of verification soundness since clauses to which no rule of the constrained system applies may fail to be evident. It turns out that such clauses can always be completed to evident clauses by adding formulas of the form rxy (so-called safe transitions). The constrained tableau system thus constitutes a decision procedure for the satisfiability of modal formulas. The termination of the constrained system also implies that the basic system can refute every unsatisfiable set of modal formulas.

We prepare the termination proofs by defining the necessary terminology. A term **t occurs in A** if it is a subterm of a formula in A. Let **Sub A** be the set of all terms that occur in A, and **Mod** A be the set of all modal terms that occur in A. By definition, $\operatorname{Mod} A \subseteq \operatorname{Sub} A$. A crucial observation for the termination proofs is the fact that the tableau rules don't add new modal terms.

Proposition 9.8.2 If $A \to C$, then $\operatorname{Mod} A = \operatorname{Mod} C$.

Exercise 9.8.3 Find A, C such that $A \to C$ and $\operatorname{Sub} C \not\subseteq \operatorname{Sub} A$.

The **height** of a clause A is the size of the largest term that occurs in A. If $A \to C$, then A and C have the same height. In other words, expansion preserves the height of clauses.

The **breadth** of a clause A is the number of elements of A (as a set). If $A \to C$, then the breadth of C is larger than the breadth of A. In other words, expansion increases the breadth of clauses.

The **vocabulary** of a clause A is the set that contains the default variable x_0 and all names that occur in A. If $A \to C$, then the vocabulary of A is a subset of the vocabulary of C. All rules but $\mathcal{R}_\Diamond$ leave the vocabulary of a clause unchanged, and $\mathcal{R}_\Diamond$ adds a new individual variable.

The **stock** of a clause A consists of all negation normal modal formulas whose size is at most the height of A and that contain only names that are in the vocabulary of A. The stock of a clause is finite since the vocabulary of a clause is finite. All rules but $\mathcal{R}_\Diamond$ preserve the stock of a clause.

The **slack** of a clause A is the number of formulas in the stock of A that are not in A. Every rule but $\mathcal{R}_\Diamond$ decreases the slack of a clause. Hence we know that an infinite derivation must employ $\mathcal{R}_\Diamond$.

Proposition 9.8.4 $\mathcal{T}_{\mathrm{M}}$ terminates for $\Diamond$-free clauses.

9.8.1 Termination for $\forall$-Free Clauses

For this result we look carefully at the new individual variables introduced by $\mathcal{R}_\Diamond$. We call a formula $\Diamond rtx$ **expanded in a clause** A if there is a y sucht that $rxy, ty \in A$. Note that $\mathcal{R}_\Diamond$ can only be applied to unexpanded $\Diamond$-formulas, and that $\mathcal{R}_\Diamond$ always expands the $\Diamond$-formula it is applied to.

Let's write $x \prec y$ if y was introduced by $\mathcal{R}_\Diamond$ to expand some formula $\Diamond rtx$. If we draw the dependency relation $x \prec y$ of a clause as a graph, we obtain a forest where the roots are initial names (i.e., names not introduced by $\mathcal{R}_\Diamond$) and all other nodes are names introduced by $\mathcal{R}_\Diamond$. We obtain termination by showing that the depth and the degree (maximal number of successors of a node) of the dependency graph of a derived clause are bounded by the initial clause. We call a clause A **admissible** if it satisfies the following two conditions:

1. If $x \prec y$ and $ty \in A$, then there is some formula $sx \in A$ such that the term s is larger than the term t.
2. If $x \in \mathcal{N}A$, then $|\{\, y \in \mathcal{N}A \mid x \prec y \,\}| \leq$ $|\{\, \Diamond rt \mid \Diamond rtx \in A \text{ and } \Diamond rtx \text{ expanded in } A \,\}| \leq |\mathrm{Mod}\, A|$

Proposition 9.8.5 Every initial clause is admissible. Morover, if A is admissible and $A \to C$, then C is admissible.

Proposition 9.8.6 Let A be an initial clause, $A \to^* C$, and $\prec$ be the dependency relation of C. Then:

1. The depth of $\prec$ is bounded by the height of A.
2. The degree of $\prec$ is bounded by $|\mathrm{Mod}\, A|$.

Proof Follows by Proposition 9.8.5, Proposition 9.8.2, and the definition of admissibility. ∎

Proposition 9.8.7 $\mathcal{T}_{\mathrm{M}}$ terminates for $\forall$-free clauses.

Proof By Proposition 9.8.6 we know that $\mathcal{R}_\Diamond$ can only be applied finitely often. Once $\mathcal{R}_\Diamond$ is not applied anymore, the remaining rules all decrease the slack of the clause and hence must terminate. ∎

Note that our termination proof relies on an informal notion of dependency relation. The dependency relation of a clause can be made formal. For this, we fix the set of initial variables and then obtain the the dependency relation of A as $\{\, (x, y) \mid \exists r\colon\ rxy \in A \text{ and } y \text{ not initial} \,\}$.

9.8.2 Termination with $\mathcal{R}^p_\Diamond$

With $\mathcal{R}_\Diamond$ the dependency relation of a derived clause will always be acyclic. Hence $\mathcal{T}_{\mathrm{M}}$ will diverge on satisfiable clauses that cannot be satisfied by a finite and acylic Kripke set. Example 9.8.1 tells us that such clauses exist. We will now consider a restricted version $\mathcal{R}^p_\Diamond$ of $\mathcal{R}_\Diamond$ such that the tableau system that uses $\mathcal{R}^p_\Diamond$ instead of $\mathcal{R}_\Diamond$ terminates. The restricted sytem may terminate with locally consistent clauses that are not evident. As it turns out, such clauses can always be completed to evident clauses by adding formulas of the form rxy. The additional formulas introduce the cycles needed to obtain a finite Kripke model.

A **pattern** is a set $\{\Diamond rt, \Box rt_1, \ldots, \Box rt_n\}$ of modal terms such that $n \geq 0$. A pattern $\{\Diamond rt, \Box rt_1, \ldots, \Box rt_n\}$ is **realized in** A if there are names x, y such that the formulas rxy, ty, and $\Box rt_1 x, \ldots, \Box rt_n x$ are in A. A formula $\Diamond rtx$ **is pattern-expanded in a clause** A if the pattern $\{\Diamond rt\} \cup \{\, \Box ru \mid \Box rux \in A \,\}$ is realized in A.

Proposition 9.8.8 If a diamond formula is expanded in A, then it is pattern-expanded in A.

We can now define the constrained tableau rule for diamond formulas:

$$\mathcal{R}^p_\Diamond \quad \frac{\Diamond rtx}{rxy,\ ty} \quad y \notin \mathcal{N}A \text{ and } \Diamond rtx \text{ not pattern-expanded in } A$$

We denote the resulting tableau system with $\mathcal{T}^p_{\mathrm{M}}$.

Proposition 9.8.9 $\mathcal{T}^p_{\mathrm{M}}$ terminates and is refutation sound.

Proof Since Mod A is invariant and finite, only finitely many patterns can be obtained with the modal terms of A. Once a pattern is realized, it stays realized. Every application of $\mathcal{R}^p_\Diamond$ realizes a pattern that was not realized before. Hence $\mathcal{R}^p_\Diamond$ can only be applied finitely often. Since the remaining rules decrease the slack, $\mathcal{T}^p_{\mathrm{M}}$ terminates.

Since all proof steps licensed by $\mathcal{R}^p_\Diamond$ are also licensed by the more permissive $\mathcal{R}_\Diamond$, we know that every $\mathcal{T}^p_{\mathrm{M}}$-tableau is also a $\mathcal{T}_{\mathrm{M}}$-tableau. Hence $\mathcal{T}^p_{\mathrm{M}}$ is refutation sound. ∎

Example 9.8.10 Here is terminating $\mathcal{T}^p_{\mathrm{M}}$-derivation starting from the satisfiable clause of Example 9.8.1.

$\forall(\Diamond rp)$	initial clause
$\Diamond rpx_0$	$\mathcal{R}_\forall$
$rx_0y,\ py$	$\mathcal{R}^p_\Diamond$
$\Diamond rpy$	$\mathcal{R}_\forall$

$\mathcal{R}^p_\Diamond$ does not apply to $\Diamond rpy$ since it is pattern-expanded. Note that the obtained clause is not evident. □

It remains to show that $\mathcal{T}^p_{\mathrm{M}}$ is verification sound. We call formulas of the form rxy **transitions**. A transition rxy is **safe in** A if $rxy \in A$ or $\forall t\colon\ \Box rtx \in A \implies ty \in A$. A formula **$\Diamond rtx$ is quasi-expanded in a clause A** if there exist y such that rxy is safe in A and $ty \in A$. A set A of negation normal modal formulas is **quasi-evident** if it satisfies all conditions for evidence except that for diamond formulas $s \in A$ it suffices that s is quasi-expanded in A (for evidence expansion is required). Note that the final clause of Example 9.8.10 is quasi-evident.

Proposition 9.8.11 Let A be a set of negation normal modal formulas to which no rule of $\mathcal{T}^p_{\mathrm{M}}$ applies. Then A is quasi-evident.

It remains to show that every quasi-evident set is satisfied by a finite Kripke set.

Proposition 9.8.12

1. If a diamond formula is expanded in A, it is also quasi-expanded in A.
2. If A is evident, A is also quasi-evident.
3. If A is quasi-evident and rxy is safe in A, then $A \cup \{rxy\}$ is quasi-evident.

Lemma 9.8.13 Let A be quasi-evident and R be the set of all transitions s such that s is safe in A and $\mathcal{N}s \subseteq \mathcal{N}A$. Then $A \cup R$ is evident.

Proof Since A is quasi-evident, A satisfies all evidence conditions but possibly (6). Adding the safe transitions does not affect the evidence conditions (1) to (4). Moreover, since the added transitions are safe, evidence condition (5) remains to hold. It remains to show that every diamond formula in A is expanded in $A \cup R$.

Let $\Diamond rtx \in A$. Since $\Diamond rtx$ is quasi-expanded in A, there is a transition rxy such that rxy is safe in A and $ty \in A$. Thus $rxy \in R$. Hence $\Diamond rtx$ is expanded in A. ■

Example 9.8.14 $A = \{\forall(\Diamond rp),\ \Diamond rpx,\ rxy,\ py,\ \Diamond rpy\}$ is a quasi-evident clause. Since A contains no box formulas, rxx, rxy, ryy, and ryx are all safe in A. Adding ryy to A yields an evident clause. □

Theorem 9.8.15 (Model Existence) Let A be a quasi-evident clause. Then there exists a finite Kripke set that satisfies A.

Proof By Lemma 9.8.13 and Proposition 9.7.10. The finiteness of the Kripke set follows from the finiteness of A since there are only finitely many transitions s such that $\mathcal{N}s \subseteq \mathcal{N}A$. ■

Corollary 9.8.16 $\mathcal{T}_{\mathrm{M}}^{p}$ is verification sound.

Proof Theorem 9.8.15 and Proposition 9.8.11. ■

Theorem 9.8.17 (Decidability) It is decidable whether a finite set of modal formulas is modally satisfiable.

Proof First we obtain an equivalent clause A by translating to negation normal form (Proposition 9.7.1). Now we apply $\mathcal{T}_{\mathrm{M}}^{p}$ to A. The claim follows since $\mathcal{T}_{\mathrm{M}}^{p}$ is terminating, refutation sound, and verification sound (Proposition 9.8.9 and Corollary 9.8.16). ■

Theorem 9.8.18 (Finite Model Property) A finite set of modal formulas is modally satisfiable if and only if it is satisfied by a finite Kripke set.

Proof First we obtain an equivalent clause A by translating to negation normal form (Proposition 9.7.1). Now we apply $\mathcal{T}_{\mathrm{M}}^{p}$ to A. The set is modally satisfiable if and only $\mathcal{T}_{\mathrm{M}}^{p}$ yields a quasi-evident clause C such that $A \subseteq C$ (follows by Proposition 9.8.9 and Proposition 9.8.11). Now the claim follows by Theorem 9.8.15. ∎

9.9 Remarks

To know more about modal logic, start with [6] and [24].

Tautologies

Boolean Connectives

$x \wedge (y \wedge z) = (x \wedge y) \wedge z$	associativity
$x \vee (y \vee z) = (x \vee y) \vee z$	
$x \wedge y = y \wedge x$	commutativity
$x \vee y = y \vee x$	
$x \wedge x = x$	idempotence
$x \vee x = x$	
$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$	distributivity
$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$	
$x \wedge (x \vee y) = x$	absorption
$x \vee (x \wedge y) = x$	
$x \wedge \top = x$	identity
$x \vee \bot = x$	
$x \wedge \bot = \bot$	dominance
$x \vee \top = \top$	
$x \wedge \neg x = \bot$	complement
$x \vee \neg x = \top$	
$\neg(x \wedge y) = \neg x \vee \neg y$	de Morgan
$\neg(x \vee y) = \neg x \wedge \neg y$	
$\neg\top = \bot$	
$\neg\bot = \top$	
$\neg\neg x = x$	double negation
$(x \vee y) \wedge (\neg x \vee z) = (x \vee y) \wedge (\neg x \vee z) \wedge (y \vee z)$	resolution
$(x \wedge y) \vee (\neg x \wedge z) = (x \wedge y) \vee (\neg x \wedge z) \vee (y \wedge z)$	

Implication

Identity

Bibliography

[1] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, volume 27 of *Applied Logic Series*. Kluwer Academic Publishers, second edition, 2002.

[2] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 3–15. ACM, 2008.

[3] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[4] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2007.

[5] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 2nd revised edition, 1984.

[6] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge University Press, 2001.

[7] George Boole. *An Investigation of the Laws of Thought*. Walton, London, 1847.

[8] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Springer, 1997.

[9] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 32:346–366, 1932.

[10] Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(1):56–68, 1940.

[11] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[12] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[13] Gottlob Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Verlag von Luois Nebert, Halle, 1879. Translated in [40], pp. 1–82.

[14] Gottlob Frege. *Grundgesetze der Arithmetik begriffsschriftlich abgeleitet*. Verlag Hermann Pohle, Jena, 1893. Translated in [40].

[15] H. Friedman. Equality between functionals. In R. Parikh, editor, *Proceedings of the Logic Colloquium 72-73*, volume 453 of *Lecture Notes in Mathematics*, pages 22–37. Springer, 1975.

[16] Gerhard Gentzen. Untersuchungen über das natürliche Schließen I, II. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.

[17] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, 1989.

[18] Kurt Gödel. Die Vollständigkeit der Axiome des logischen Functionenkalküls. *Monatshefte für Mathematik und Physik*, 37:349–360, 1930. Translated in [40], pp. 102-123.

[19] John Harrison. HOL Light tutorial (for version 2.20). http://www.cl.cam.ac.uk/~jrh13/hol-light/tutorial_220.pdf, 2006.

[20] L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15(2):81–91, June 1950.

[21] L. Henkin. A theory of propositional types. *Fundamenta Mathematicae*, 52:323–344, 1963.

[22] J. R. Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1997.

[23] Michael Huth and Mark Ryan. *Logic in Computer Science*. Cambridge, second edition, 2004.

[24] Mark Kaminski and Gert Smolka. Terminating tableau systems for modal logic with equality. Technical report, Saarland University, 2008. http://www.ps.uni-sb.de/Papers/abstracts/KaminskiSmolka08equality.pdf.

[25] Jean-Louis Krivine. *Introduction to Axiomatic Set Theory*. Reidel, Dordrecht, Holland, 1971.

[26] Azriel Levy. *Basic Set Theory*. Springer, Berlin - New York, 1979.

[27] Daniel R. Licata, Noam Zeilberger, and Robert Harper. Focusing on binding and computation. In *LICS*. IEEE, 2008.

[28] J. C. Mitchell. *Foundations for Programming Languages*. Foundations of Computing. The MIT Press, 1996.

[29] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[30] G. Peano. Arithmetices principia, nova methodo exposita. Turin, 1889. Translated in [40], pp. 83–97.

[31] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[32] G. D. Plotkin. Lambda-definability in the full type hierarchy. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 365–373. Academic Press, 1980.

[33] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[34] Bertrand Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30:222–262, 1908.

[35] Moses Schönfinkel. Über die Bausteine der Mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924.

[36] Allen Stoughton. Substitution revisited. *Theoretical Computer Science*, 59:317–325, 1988. http://people.cis.ksu.edu/~stough/research/subst.ps.

[37] W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

[38] A.S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, second edition, 2000.

[39] Christian Urban, Stefan Berghofer, and Michael Norrish. Barendregt's variable convention in rule inductions. In *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2007.

[40] J. van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*. Source Books in the History of the Sciences. Harvard University Press, 2002.

[41] Edward N. Zalta, editor. *Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, CSLI, Stanford University, 2008. http://plato.stanford.edu/.