Introduction to Computational Logic

Lecture Notes SS 2009

August 2, 2009

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

Contents

1 Introduction

1 Introduction

2 Structure of Mathematical Statements

In this chapter 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

A function is something that takes an argument and yields a result. We capture this idea by defining a function f as a set of pairs, where $(x, y) \in f$ means that f for the argument x yields the result y. Given this definition, the empty set is the unique function that for no argument yields a result. Let us go through the exact definitions we will use.

A **binary relation** is a set of pairs. If R is a binary relation, we define the **domain** and **range** of R as follows:

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

A **function** is a binary relation f such that for every $x \in \mathrm{Dom}\, f$ there exists exactly one $y \in \mathrm{Ran}\, f$ such that $(x, y) \in f$. Given two sets X and Y, a **function** $X \to Y$ is a function f such that $\mathrm{Dom}\, f = X$ and $\mathrm{Ran}\, f \subseteq Y$. 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 $f x$ for the unique y such that $(x, y) \in f$. We write $f x + 5$ for $(f x) + 5$.

It is often convenient to describe functions with the **lambda notation**. 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:

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

The equation shows the analogy between the lambda notation and the more common set notation. In both notations x appears as a bound variable. The equation also demonstrates another important point: One must distinguish between an object and its description. One and the same object can have many different descriptions. The **lambda notation** is due to the American logician Alonzo Church [18].

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 $(x_1, \ldots, x_n)$ that list the arguments $x_1, \ldots, x_n$ 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**. For 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 [27] and was fully developed in a paper by Moses Schönfinkel [58] on the primitives of mathematical language. From a logical point of view the cascaded representation has the advantage that it doesn't require tuples. Following common practice, we omit parentheses as follows:

$$\begin{aligned} f\,x\,y &\rightsquigarrow (f\,x)\,y \\ X \to Y \to Z &\rightsquigarrow X \to (Y \to Z) \end{aligned}$$

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

Exercise 2.1.1 Describe a function $f \in \mathbb{Z} \to \{0\}$ both with lambda notation and set notation. How many functions $\mathbb{Z} \to \{0\}$ are there?

Exercise 2.1.2 Which of the following statements are valid?

a) If $X \subseteq X'$, then $(X \to Y) \subseteq (X' \to Y)$.

b) If $X \subseteq X'$, then $(X' \to Y) \subseteq (X \to Y)$.

c) If $Y \subseteq Y'$, then $(X \to Y) \subseteq (X \to Y')$.

d) If $Y \subseteq Y'$, then $(X \to Y') \subseteq (X \to Y)$.

2.2 Boolean Operations

We use the numbers 0 and 1 as **truth values**, where 0 may be thought of as "false" and 1 as "true". In programming languages, truth values are commonly called *Boolean values*. We define the set

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

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

$$(3 < 7) = 1$$
$$(7 \leq 3) = 0$$
$$(3 = 7) = 0$$

The following equations define well-known Boolean operations:

$\neg x = 1 - x$	**Negation**
$(x \wedge y) = \min\{x, y\}$	**Conjunction**
$(x \vee y) = \max\{x, y\}$	**Disjunction**
$(x \to y) = (\neg x \vee y)$	**Implication**
$(x \equiv y) = (x = y)$	**Equivalence**

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

Note that the definition of negation, conjunction, and disjunction does exploit that truth values are numbers. Here are alternative definitions not exploiting this fact:

$\neg x$ = if x=1 then 0 else 1	**Negation**
$(x \wedge y)$ = if x=1 then y else 0	**Conjunction**
$(x \vee y)$ = if x=1 then 1 else y	**Disjunction**

George Boole [12] was an English logician who studied the algebraic properties of negation, conjunction, and disjunction. His system is now known as

Boolean algebra and is acknowledged as the first abstract algebraic system studied. Boolean algebra does not commit to just two truth values 0 and 1. Working with just two truth values is common in programming languages and computer hardware. Two truth values will also be perfect for the logical systems we are going to consider. But there is research on many-valued logic.

Exercise 2.2.1 Consider the values

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

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

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

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

2.3 Operator Precedence

In the following an operator will be a symbol for a binary or a unary 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:

λ	lambda
$\equiv$	equivalence
$\to$	implication
$\vee$	disjunction
$\wedge$	conjunction
$\neg$	negation
$= \quad < \quad \leq \quad > \quad \geq$	comparisons
$+ \quad -$	addition, substraction
$\cdot$	multiplication
	ordinary application

Symbols appearing lower in the hierarchy take their arguments before symbols appearing higher in the hierarchy. Note that λ takes its body last and that ordi-

nary application (e.g., fx) takes its arguments first. Here are examples of notations that omit parentheses according to the precedence hierarchy:

$$\begin{aligned} x \vee x \wedge y \equiv x &\rightsquigarrow (x \vee (x \wedge y)) \equiv x \\ \neg x = y &\rightsquigarrow \neg(x = y) \\ \neg x \equiv y &\rightsquigarrow (\neg x) \equiv y \\ \neg\neg x = y \equiv y = x &\rightsquigarrow (\neg(\neg(x = y))) \equiv (y = x) \\ 3 \cdot fx + 7 &\rightsquigarrow (3 \cdot (fx)) + 7 \\ \lambda x{\in}X.\, fxy + 7 &\rightsquigarrow \lambda x{\in}X.\, (((fx)y) + 7) \end{aligned}$$

We arrange that all infix operators (but ordinary application) group to the right. For instance,

$$\begin{aligned} x \wedge y \wedge z &\rightsquigarrow x \wedge (y \wedge z) \\ x \to y \to z &\rightsquigarrow x \to (y \to z) \end{aligned}$$

As already stated, ordinary application groups to the left:

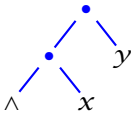
$$fxy \rightsquigarrow (fx)y$$

The operator $\to$ is used both for implication and function types (see § 2.1), and both uses group to the right.

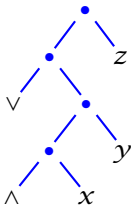
We write $s \neq t$ and $s \not\equiv t$ as abbreviations for $\neg(s = t)$ and $\neg(s \equiv t)$.

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**. In the following, we will draw terms as trees. For instance, the notation $x \wedge y$ describes the term

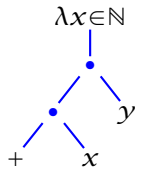


and the notation $x \wedge y \vee z$ describes the term



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 modelled as cascaded functions.

The λ-notation decribes special terms called **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.

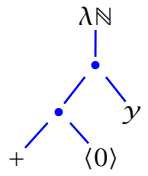
A few words about how we say things. When we say *the term* $\lambda x{\in}\mathbb{N}.\ 2{\cdot}x$, we mean the term described by the notation $\lambda x{\in}\mathbb{N}.\ 2{\cdot}x$. And when we say *the function* $\lambda x{\in}\mathbb{N}.\ 2{\cdot}x$, we mean the function described by the term described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

Given a semantic object, there are usually many different terms that desribe it. For instance, the terms 3 and $5 - 2$ both describe the number 3. When we want to describe a semantic object, we can choose a term for the object and then a notation for the term.

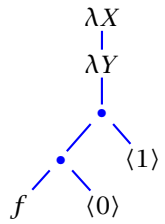
2.5 Locally Nameless Term Representation

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 $\lambda x{\in}X.\ \lambda y{\in}Y.\ f\,y\,x$:



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 [23]. For this reason, numeric argument references in terms are often called **de Bruijn indices**.

We call two terms **α-equivalent** if they have the same locally nameless representation. There is the intuition that two terms are α-equivalent if and only if they are equal up to renaming of bound variables.

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

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

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

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

2.6 Formulas, Identity Predicates, 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 predicate 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. Identity predicates 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 [20] 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 [12] 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:

$\forall_X \in (X \to \mathbb{B}) \to \mathbb{B}$ **universal quantifier**

$\forall_X f = (f = (\lambda x{\in}X.1))$

$\exists_X \in (X \to \mathbb{B}) \to \mathbb{B}$ **existential quantifier**

$\exists_X f = (f \neq (\lambda x{\in}X.0))$

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:

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

Frege and Russell 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 and Relations as Predicates

A **predicate** is a function $X_1 \to \cdots \to X_n \to \mathbb{B}$. Roughly speaking, a predicate is a function that after taking all its arguments returns a truth value. The Boolean operations, the quantifiers, and of course the identitity predicates are examples of predicates. It turns out that sets and relations can be represented as predicates.

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

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

Given the representation of sets as predicates, 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 $X \to \mathbb{B}$.

Let $R \subseteq X \times X$ be a binary relation. Then we can represent R through its **characteristic predicate** $\lambda x{\in}X.\lambda y{\in}X.(x, y){\in}R$. The representation extends to relations with more than 2 arguments.

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. Such 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. For the empty set a choice function for X yields some element of X. If a choice function for X is applied to a singleton set $\{x\}$, there is no choice and it must return x. This is the most interesting use for choice functions. Let $C_{\mathbb{Z}}$ be a choice function for $\mathbb{Z}$. Then

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

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 predicate $=_{\mathbb{N}}$.

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:

$$x{=_X}y \ \equiv\ \forall p{\in}X{\to}\mathbb{B}.\ px \to py$$

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 $p = \lambda z.\, z{=}x$ 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 $\mathbb{B}{\to}\mathbb{B}{\to}\mathbb{B}$ can also express conjunction:

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

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 **Boolean extensionality law** says that two truth values are equal if they imply each other:

$$(x \to y) \wedge (y \to x)\ \to\ x = y$$

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

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

The extensionality law holds since functions are sets of pairs. The left hand side of the law just says that the sets f and g consist of exactly the same pairs. Given the extensionality laws, we can express equality of truth values and functions as follows:

$$x = y \;\equiv\; (x \to y) \wedge (y \to x)$$
$$f = g \;\equiv\; \forall x{\in}X.\, fx = gx$$

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 $X \to \mathbb{B}$.

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 Identity Predicates) 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 [36] 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 $=_{(\mathbb{B}\to\mathbb{B}\to\mathbb{B})\to\mathbb{B}}$.

g) Express $\vee$ with 0, $\neg$, $=_{\mathbb{B}}$, and $\forall_{\mathbb{B}\to\mathbb{B}\to\mathbb{B}}$.

h) Express $\to$ with 0, 1, $\neg$, $=_{\mathbb{B}}$, and $\forall_{\mathbb{B}\to\mathbb{B}\to\mathbb{B}}$.

2.11 Laws for the Lambda Notation

The **η-law** for functions looks as follows:

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

The η-law holds 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.

Finally, we look at the **β-law**. The β-law is a syntactic law whose formulation requires the notion of substitution. Here are instances of the β-law:

$$(\lambda x{\in}\mathbb{N}.x + 2)5 \;=\; 5 + 2$$
$$(\lambda x{\in}\mathbb{N}.x + 2)(x + y) \;=\; (x + y) + 2$$

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 for 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.

2.12 Remarks

The outlined logical language is largely due to Alonzo Church [20]. It is associated with logical systems known as *simple type theory* or *simply-typed higher-order logic*. Church started with an untyped language [18] to describe computational functions and later used a typed language [20] to describe mathematical

functions. Types originated with Bertrand Russell [57]. A logical language with quantification was first studied by Gottlob Frege [26].

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 as much as we can. The syntactic system we study is known as lambda calculus and concerns terms with lambda abstractions. We first consider untyped terms.

3.1 Untyped Terms

We assume that a set Nam of **names** is given and a bijection $\mathrm{Nam} \cong \mathbb{N}$. We could choose $\mathrm{Nam} = \mathbb{N}$, but requiring this equality only up to bijection gives us more flexibility. The set of terms Ter is defined inductively:

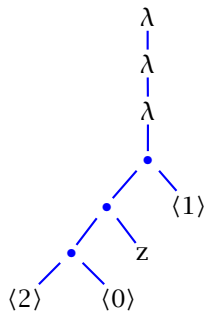
1. Every name is a term.
2. If s and t are terms, then st is a term.
3. If x is a name and s is a term, then $\lambda x.s$ is a term.

We understand the definition such that every term is exactly one of the following: a name x, an **application** st, or an **abstraction** $\lambda x.s$ To be concrete, we represent a term as a pair (i, y) 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 $y \in \mathbb{N}$, for applications $y \in \mathrm{Ter} \times \mathrm{Ter}$, and for λ-terms $y \in \mathrm{Nam} \times \mathrm{Ter}$. Note that $(\lambda x.x) \neq \lambda y.y$ if $x \neq y$. The definition of terms can be summarized with the grammar $s ::= x \mid ss \mid \lambda x.s$.

We will use the letters x, y, z for names and the letters s, t, u, v for terms. An abstraction $\lambda x.s$ may also be called a **λ-term**. Given an abstraction $\lambda x.s$, we call x the **argument name** and s the **body** of the abstraction. Argument names may also be referred to as **local names**. As it comes to notation, we follow the conventions introduced in § 2.3. For instance:

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

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:

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

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

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

For instance, $\mathcal{N}(\lambda f x y.f y z x) = \{z\}$ 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 inductively:

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

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 **λ-free** 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 Is x a subterm of $\lambda x.y$?

Exercise 3.2.3 Determine all pairs C, s such that $C[s] = xxxx$ and s is a application.

Exercise 3.2.4 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, Capturing, and Renaming

Substitution is a syntactic operation. We have encountered substitution first in § 2.11 when we formulated the β-law. In its simplest form, substitution is an operation s_t^x that replaces the free occurrences of the name x in the term s with the term t:

$$\begin{aligned}
(fxy)_z^x &= fzy \\
(fxy)_{fxy}^x &= f(fxy)y \\
(\lambda x.fxy)_z^x &= \lambda x.fxy \\
(\lambda x.fxyy)_z^y &= \lambda x.fxzz
\end{aligned}$$

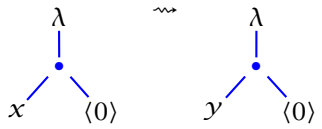
There is a complication known as **capturing**. Consider $(\lambda x.y)_x^y$ where the y in $\lambda x.y$ must be replaced with x. If we do this naively, we obtain $\lambda x.x$, which means that the external occurrence of x has been captured by the binder λx as a local argument reference. To meet the needs of the β-law, substitution must be defined such that capturing does not happen.

Example 3.3.1 Consider the notation $\lambda x{\in}\mathbb{B}.\lambda y{\in}\mathbb{B}.x$ which describes a function that takes two arguments and yields the first argument. Obviously,

$$(\lambda x{\in}\mathbb{B}.\lambda y{\in}\mathbb{B}.x)y \neq (\lambda y{\in}\mathbb{B}.y)$$

for every $y \in \mathbb{B}$ since on the left we have the function $\lambda z{\in}\mathbb{B}.y$ that returns y for every argument while on the right we have the identity function $\lambda y{\in}\mathbb{B}.y$ that returns different results for different arguments. From this we learn that the substitution $(\lambda y.x)^x_y$ must not yield $\lambda y.y$ but a term like $\lambda x.y$ or $\lambda z.y$. Note that the names x, y, and z are assumed to be different. □

If we define substitution on the locally nameless representation of terms, the capturing problem completely disappears since there are no local names. For instance, if $s = \lambda y.xy$ and we want to obtain s^x_y, naive replacement of x with y yields exactly what we want:

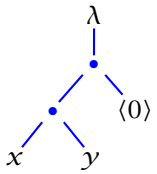


Thus the no-capture requirement for substitution can also be derived from the requirement that substitution on the term level should be compatible with substitution on the LNR level.

In summary we can say that substitution must be defined such that it preserves the binding structure of a term.

To avoid capturing, we will define substitution such that it may **rename** (i.e., replace) local names. For instance, $(\lambda x.y)^y_x$ may yield $\lambda z.x$, which does describe the function we want to obtain. Renaming of local names is referred to as **α-renaming**.

How should substitution choose names if it has to rename local names? It is helpful to first consider a simplified problem: How can we choose local names for the λ-nodes of an LNR such that we obtain a term whose LNR is the given one? For instance, consider the LNR



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

Now 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.

3.4 Substitution Operators

It is best to define the substitution operation such that it can replace more than one name. To do so, we will define a substitution operator S that applies a substitution θ to a term s. A substitution θ will be a function that for each name gives a term that should replace the name. Replacing all names is no problem since $\theta x = x$ means that x will be replaced by x, which is the same as not replacing x.

Definition. A **substitution** is a function Nam → Ter. The **carrier** of a substitution θ is the set $C\theta := \{ x \in \text{Nam} \mid \theta x \neq x \}$. We use ι to denotes the **identity substitution** $\lambda x{\in}\text{Nam}.\, x$. Moreover, we use the notation $[x_1{:=}s_1, \ldots, x_n{:=}s_n]$ to denote the substitution θ such that $C\theta \subseteq \{x_1, \ldots, x_n\}$ and $\theta x_i = s_i$ for $i \in \{1, \ldots, n\}$. We may also use $[]$ to denote the identity substitution ι.

If we apply a substitution θ to a term $\lambda x.s$, we may have to rename the argument name x. For this we require a renaming function ρ that provides us with a safe name that we can use instead of x. If it is safe to keep x, the renaming function may of course yield x.

Definition. We say that a name y **is safe for** θ, x, s if there is no $z \in \mathcal{N}(\lambda x.s)$ such that $y \in \mathcal{N}(\theta z)$. A **renaming function** ρ is a function (Nam → Ter) → Nam → Ter → Nam such that $\rho\theta x s$ is safe for all θ, x, s. For every renaming function ρ we define inductively a **substitution operator** S:

$$\begin{aligned} S\theta x &= \theta x \\ S\theta(st) &= (S\theta s)(S\theta t) \\ S\theta(\lambda x.s) &= \lambda y.\, S(\theta^x_y)s \quad \text{where } y = \rho\theta x s \end{aligned}$$

The notation θ^x_y describes the substitution that is like θ except that it maps x to y.

Proposition 3.4.1 For every substitution operator S:
If s is λ-free, then $S\iota s = s$.

Proposition 3.4.2 (Free Names) For every substitution operator S:
$\mathcal{N}(S\theta s) = \cup\{ \mathcal{N}(\theta x) \mid x \in \mathcal{N} s \}$.

A renaming function ρ is **conservative** if $\rho\theta xs = x$ whenever x is safe for θ, x, s. A substitution operator S is **conservative** if it is obtained with a conservative renaming function.

Proposition 3.4.3 For every conservative substitution operator S:
$S\theta s = s$ if no name in the carrier of θ is free in s.

We fix some conservative substitution operator S_0 and define the notation $s_t^x := S_0[x{:=}t]s$. Note that $[x{:=}t] = \lambda y{\in}\mathrm{Nam}.\,\text{if } y = x \text{ then } t \text{ else } y$.

Proposition 3.4.4

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

Exercise 3.4.5 Apply the following substitutions.

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

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

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

Exercise 3.4.6 Let $x \neq y$. Find C such that $(C[x])_y^x \neq C[y]$. Hint: Exploit that $C[x]$ may capture x.

Exercise 3.4.7 Give the carrier of the identity substitution.

Exercise 3.4.8 Find a renaming function ρ such that $S\iota(\lambda x.x) \neq S[x{:=}y](\lambda x.x)$ for the corresponding substitution operator S. Assume that x and y are different.

3.5 Alpha Equivalence

Two terms are *α-equivalent* if they have the same LNR. This sentence gives the right intuition but doesn't suffice for a formal definition since the LNR of terms is not formally defined. Fortunately, there is a substitution operator that provides for an elegant formal definition of α-equivalence.

Let ρ_α be the renaming function such that $\rho_\alpha\theta xs$ is the least name that is safe for θ, x, s (for the order we exploit the bijection $\mathrm{Nam} \cong \mathbb{N}$). Moreover, let S_α be the substitution operator obtained with ρ_α. Note that ρ_α and S_α are not conservative. We define **α-equivalence** as follows:

$$s \sim_\alpha t \;:\Longleftrightarrow\; S_\alpha\iota s = S_\alpha\iota t$$

This definition works since $S_\alpha \iota s$ yields a variant of s where all local names are renamed to be the least possible ones. Here are examples ($x \cong 0$, $y \cong 1$):

$$\begin{aligned} S_\alpha \iota(\lambda x.x) &= \lambda x.x \\ S_\alpha \iota(\lambda y.y) &= \lambda x.x \\ S_\alpha \iota(\lambda yx.xy) &= \lambda xy.yx \end{aligned}$$

Proposition 3.5.1 If $s \sim_\alpha t$, then $\mathcal{N}s = \mathcal{N}t$.

With the help of α-equivalence we can state three important properties of substitution.

Proposition 3.5.2 For all substitution operators S and S':

Preservation $S\theta s \sim_\alpha s$ if no name in the carrier of θ is free in s

Coincidence $S\theta s \sim_\alpha S'\theta' s'$ if $s \sim_\alpha s'$ and $\theta x \sim_\alpha \theta' x$ for all $x \in \mathcal{N}s$

Renaming $S\theta(\lambda x.s) \sim_\alpha \lambda y.S(\theta^x_y)s$ if y not free in $S\theta(\lambda x.s)$

From the coincidence property we learn that all substitution operators are equal up to α-equivalence.

There are many substitution operators S such that $s \sim_\alpha t \iff Ss = St$. To better understand this fact, we define a class of substitution operators that have this property. A renaming function ρ is **strict** if $\rho\theta x s = \rho\theta' x' s'$ whenever

$$\bigcup_{y \in \mathcal{N}(\lambda x.s)} \mathcal{N}(\theta y) = \bigcup_{y \in \mathcal{N}(\lambda x'.s')} \mathcal{N}(\theta' y)$$

A substitution operator S is **strict** if it is obtained with a strict renaming function. Note that ρ_α and S_α are strict. Informally, we can understand a strict renaming function as a renaming function that bases its choice of a name only on the absolutely necessary information, which is the set of unsafe names.

Proposition 3.5.3 For every strict substitution operator S:

1. $S\iota(S\iota s) = S\iota s$
2. $s \sim_\alpha t \iff S\iota s = S\iota t$

Exercise 3.5.4 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 3.5.5 Determine $S_\alpha \iota t$ 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 3.5.6 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) $\lambda x.s \sim_\alpha \lambda y.t \iff s^x_y \sim_\alpha t$

Exercise 3.5.7 Are there conservative substitution operators that are strict?

Exercise 3.5.8 Prove Proposition 3.5.1.

3.6 Alpha Normality

Two α-equivalent terms will always denote the same semantic object. Since an LNR describes a term up to α-equivalence, we could use LNRs rather than terms to describe semantic objects. We could also insist that for every abstraction $\lambda x.s$ the argument name x is the least name that is not free in $\lambda x.s$. Terms with this property are called α-normal. The official definition is as follows.

The set of **α-normal terms** is defined inductively:

1. Every name is α-normal.
2. st is α-normal iff s and t are α-normal.
3. $\lambda x.s$ is α-normal iff s is α-normal and x is the least name that is not free in $\lambda x.s$.

The next proposition states that LNRs are in one-to-one correspondence with α-normal terms.

Proposition 3.6.1 For every term s there is exactly one α-normal term t such that $s \sim_\alpha t$. This term is $S_\alpha \iota s$.

A substitution θ is **α-normal** if θx is an α-normal term for every name x.

Proposition 3.6.2 If θ is α-normal, then $S_\alpha \theta s$ is α-normal.

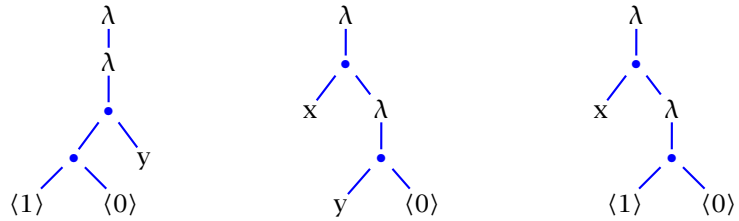
Proof By induction on the inductive definition of S_α. We have to verify the claim for each of the three defining equations. The three equations correspond to the

three clauses of the inductive definition of α-normality. For the first equation the claim follows from the α-normality of θ, for the second equation the claim follows by inductive hypothesis, and for the third equation the claim follows by inductive hypothesis and the fact that the fresh name y is chosen with ρ_α. ∎

Here is an algorithm that, given an LNR L, computes an α-normal term s such that L is the LNR of s.

1. If L contains no unassigned λ-node, return the term described by L.
2. Otherwise, choose a topmost λ-node v in L. To v assign the least name x that does not occur in the subtree rooted by v. Replace all argument references to v with x.
3. Continue until no unassigned λ-node is left.

Exercise 3.6.3 Assume $x \cong 0$, $y \cong 1$, and $z \cong 2$. Give α-normal terms whose LNRs look as follows:



Exercise 3.6.4 Give an α-normal term s and a substitution θ such that $S_\alpha \theta s$ is not α-normal.

3.7 Beta Reduction and Beta Equivalence

The β-law introduced in § 2.11 says that the term $(\lambda x.s)t$ describes the same value as the term s^x_t. Hence we can replace a term $(\lambda x.s)t$ with the seemingly simpler term s^x_t. Such a replacement is known as a **β-reduction**. Here is an example:

$$\begin{aligned} & (\lambda f x.f(fx))(\lambda x.x) \\ \to_\beta \;& \lambda x.(\lambda x.x)((\lambda x.x)x) \\ \to_\beta \;& \lambda x.(\lambda x.x)x \\ \to_\beta \;& \lambda x.x \end{aligned}$$

We need some definitions concerning β-reduction. A term of the form $(\lambda x.s)t$ is called a **β-redex**.[1] A term is **β-normal** if it contains no β-redex. We write $s \to_\beta t$

[1] Redex is an artificial word introduced by Church that abbreviates reducible expression.

if $s = C[(\lambda x.u)v]$ and $t = C[u[x:=v]]$ for some context C and some term $(\lambda x.u)v$. By Proposition 3.4.2 we have $\mathcal{N}s \supseteq \mathcal{N}t$ if $s \to_\beta t$.

Consider the term $\omega := \lambda x.xx$ and the β-reduction $\omega\omega \to_\beta \omega\omega$. From this example we learn that there are untyped terms for which β-reduction does not terminate.

We define **β-equivalence** $\sim_\beta$ as the least equivalence relation on terms containing the relations $\sim_\alpha$ and $\to_\beta$. In other words, $\sim_\beta$ is the least equivalence relation on terms such that $s \sim_\beta t$ if $s \sim_\alpha t$ or $s \to_\beta t$. A term t is a **β-normal form** of a term s if $s \sim_\beta t$ and t is β-normal. Not every term has a β-normal form (e.g., $\omega\omega$).

To decide $s \sim_\beta t$, we may apply β-reduction to s and t. If we are lucky, we obtain two β-normal terms s' and t'. The next theorem tells us that $s \sim_\beta t$ iff $s' \sim_\alpha t'$.

Theorem 3.7.1 (β-Normal Form)
Let s and t be β-normal. Then $s \sim_\beta t \iff s \sim_\alpha t$.

The normal form theorem follows from a more general result known as confluence of β-reduction [42, 7]. Confluence of β-reduction was first shown in 1935 by Church and Rosser.

Corollary 3.7.2 A term has at most one β-normal form (up to α-equivalence).

Exercise 3.7.3 Use β-reduction to derive β-normal forms for 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 3.7.4 Determine all pairs C, s such that $C[s] = (\lambda f.fx)((\lambda fx.fx)(\lambda x.x))$ and s is a β-redex.

Exercise 3.7.5 Find terms as follows.

a) A term that has no β-normal form.

b) A term that has a β-normal form but on which β-reduction does not terminate.

c) A term s_0 such that there exists an infinite derivation $s_0 \to_\beta s_1 \to_\beta s_2 \to_\beta \ldots$ such that $|s_n| < |s_{n+1}|$ for all $n \in \mathbb{N}$.

Exercise 3.7.6 We did define α-normal terms inductively and β-normal terms non-inductively. It is also possible to define α-normal terms non-inductively and β-normal terms inductively.

a) Give an inductive definition of the set of β-normal terms.

b) Give a non-inductive definition of α-normal terms. Define a suitable notion of α-redex first.

Example 3.7.7 (Explosive Terms) There are small terms that require an enormous number of β-reductions until a β-normal form is reached. Consider the closed terms $I := \lambda x.x$ and $T := \lambda fx.f(fx)$ (identity and twice). It takes 4 steps to obtain a β-normal form of TII:

$$TII \to_\beta (\lambda x.I(Ix))I \to_\beta I(II) \to_\beta II \to_\beta I$$

For a β-normal form of $TTII$ we need 12 steps:

$$TTII \to^2_\beta T(TI)I \to^2_\beta TI(TII) \to^4_\beta TII \to^4_\beta I$$

For $TTTII$ we need 54 steps:

$$\begin{aligned}
&TTTII \to^2_\beta T(TT)II \to^2_\beta TT(TTI)I \to^2_\beta TT(T(TI))I \to^2_\beta T(T(T(TI)))I\\
&\to^2_\beta T(T(TI))(T(T(TI))I) \to^2_\beta T(T(TI))(T(TI)(T(TI)I))\\
&\to^2_\beta T(T(TI))(T(TI)(TI(TII))) \to^4_\beta T(T(TI))(T(TI)(TII))\\
&\to^4_\beta T(T(TI))(T(TI)I) \to^2_\beta T(T(TI))(TI(TII)) \to^4_\beta T(T(TI))(TII)\\
&\to^4_\beta T(T(TI))I \to^2_\beta T(TI)(T(TI)I) \to^2_\beta T(TI)(TI(TII)) \to^4_\beta T(TI)(TII)\\
&\to^4_\beta T(TI)I \to^2_\beta TI(TII) \to^4_\beta TII \to^4_\beta I
\end{aligned}$$

For $TTTTII$ the explosion gets worse, and for $TTTTTII$ it gets out of hand. One can show that $TTTTII$ requires at least 2^{16} β-reductions, and that $TTTTTII$ requires at least $2^{2^{16}}$ β-reductions. Note that $16 = 2^{2^2}$, so there are as many 2's as there are T's. □

Exercise 3.7.8 Express I and T as polymorphic procedures in ML and checkout how long it takes to evaluate $TTTTII$ and $TTTTTII$.

3.8 Eta Reduction and Beta-Eta Equivalence

The η-law says that a term $\lambda x.sx$ describes the same value as the term s if x is not free in s. The corresponding simplification rule is known as **η-reduction**. Here is an example:

$$\begin{aligned}
& (\lambda x.fx)(\lambda xy.gxy)\\
\to_\eta\ & f(\lambda xy.gxy)\\
\to_\eta\ & f(\lambda x.gx)\\
\to_\eta\ & fg
\end{aligned}$$

In contrast to β-reduction, η-reduction terminates. An **η-redex** is a term of the form $\lambda x.sx$ where x is not free in s. A term is **η-normal** it it contains no η-redex. We write $s \to_\eta t$ if $s = C[\lambda x.ux]$, $t = C[u]$, and x is not free in u.

Proposition 3.8.1 η-reduction preserves β-normality. That is, if s is β-normal and $s \to_\eta t$, then t is β-normal.

We define **$\beta\eta$-equivalence** $\sim_{\beta\eta}$ as the least equivalence relation on terms containing $\to_\beta$ and $\to_\eta$. $\beta\eta$-equivalence provides for α-renaming. Given an abstraction $\lambda x.s$ and a name y that is not free in s, we have

$$\lambda x.s \;\leftarrow_\eta\; \lambda y.(\lambda x.s)y \;\to_\beta\; \lambda y.s^x_y$$

and hence $\lambda x.s \sim_{\beta\eta} \lambda y.s^x_y$.

Proposition 3.8.2 $\sim_\alpha \subseteq \sim_\beta \subseteq \sim_{\beta\eta}$.

A term is **$\beta\eta$-normal** if it is β-normal and η-normal. A term t is a **$\beta\eta$-normal form** of a term s if $s \sim_{\beta\eta} t$ and t is $\beta\eta$-normal.

Theorem 3.8.3 ($\beta\eta$-Normal Form)
Let s and t be $\beta\eta$-normal. Then $s \sim_{\beta\eta} t \iff s \sim_\alpha t$.

Exercise 3.8.4 Use β- and η-reduction to derive $\beta\eta$-normal forms of the following terms.

a) $\lambda xy.fx$

b) $\lambda xy.fy$

c) $\lambda xy.fxy$

d) $\lambda y.(\lambda x.fxy)x$

3.9 Compatibility and Stability

We have defined several binary relations on terms: $\sim_\alpha$, $\sim_\beta$, $\sim_{\beta\eta}$, $\to_\beta$, and $\to_\eta$. They all have two properties known as compatibility and stability. A binary relation $R \subseteq \mathrm{Ter} \times \mathrm{Ter}$ is

- **compatible** if $(C[s], C[t]) \in R$ whenever $(s,t) \in R$.
- **stable** if $(S\theta s, S\theta t) \in R$ whenever $(s,t) \in R$.

Proposition 3.9.1 $\sim_\alpha$, $\sim_\beta$, $\sim_{\beta\eta}$, $\to_\beta$, and $\to_\eta$ are compatible and stable.

Proposition 3.9.2 If $n \geq 1$ and $x_1, \ldots, x_n$ are pairwise distinct, then $S[x_1{:=}s_1, \ldots, x_n{:=}s_n]t \;\sim_\beta\; (\lambda x_1 \ldots x_n.t)s_1 \ldots s_n$.

$\mathcal{B}$: (Nam → Ter) → Ter → Ter

$$\begin{aligned}
\mathcal{B}\theta x &= \theta x \\
\mathcal{B}\theta(\lambda x.s) &= \lambda y.\mathcal{B}(\theta^x_y)s \quad \text{where } y = \rho\theta x s \\
\mathcal{B}\theta(st) &= \mathcal{B}[x{:=}\mathcal{B}\theta t]u \quad \text{if } \mathcal{B}\theta s = \lambda x.u \\
\mathcal{B}\theta(st) &= (\mathcal{B}\theta s)(\mathcal{B}\theta t) \quad \text{otherwise}
\end{aligned}$$

Figure 3.1: The normalization procedure

Exercise 3.9.3 Prove the following:

a) $\lambda x.s \sim_\beta \lambda x.t \iff s \sim_\beta t$

b) $\lambda x.s \sim_{\beta\eta} \lambda x.t \iff s \sim_{\beta\eta} t$

c) $sx \sim_{\beta\eta} tx \iff s \sim_{\beta\eta} t$ if x not free in s or t

3.10 Normalization

We now consider a procedure that given a term tries to compute a β-normal form. The procedure proceeds by β-reducing the term as long as it is possible. If the normalization process terminates, a β-normal form of the initial term has been obtained. Since there are terms for which β-reduction does not terminate, the procedure will not always succeed. However, it will turn out that the procedure always succeeds for typed terms. Given a renaming function ρ, we define the normalization procedure $\mathcal{B}$ as shown in Figure 3.1. $\mathcal{B}$ works like the substitution procedure S except that after processing an application it checks whether a β-redex is obtained. If this is the case, $\mathcal{B}$ does not return the β-redex but recursively attempts to compute its normal form.

A substitution θ is **β-normal** if θx is β-normal for every name x.

Proposition 3.10.1 Let $\mathcal{B}$ terminate on θ, s. Then:

1. $\mathcal{B}\theta s \sim_\beta S\theta s$.
2. If θ is β-normal, then $\mathcal{B}\theta s$ is β-normal.

Proof Both claims follow by procedural induction. We verify the first claim for each of the defining equations of $\mathcal{B}$. By coincidence (Proposition 3.5.2) we can assume without loss of generality that $\mathcal{B}$ and S use the same renaming function ρ.

1. equation. The claim holds since $S\theta x = \theta x$.

2. equation. Let $y = \rho\theta x s$. Then $S\theta(\lambda x.s) = \lambda y.S(\theta^x_y)s \sim_\beta \lambda y.\mathcal{B}(\theta^x_y)s$ by the definition of S, the inductive hypothesis, and compatibility (Proposition 3.9.1).

3. *equation.* Let $\mathcal{B}\theta s = \lambda x.u$. Then $S\theta(st) = (S\theta s)(S\theta t) \sim_\beta (\mathcal{B}\theta s)(\mathcal{B}\theta t) = (\lambda x.u)(\mathcal{B}\theta t) \to_\beta u^x_{\mathcal{B}\theta t} \sim_\alpha S[x{:=}\mathcal{B}\theta t]u \sim_\beta \mathcal{B}[x{:=}\mathcal{B}\theta t]u$ by the definition of S, the inductive hypothesis and compatibility, coincidence, and once more the inductive hypothesis.
4. *equation.* $S\theta(st) = (S\theta s)(S\theta t) \sim_\beta (\mathcal{B}\theta s)(\mathcal{B}\theta t)$ by the definition of S, the inductive hypothesis, and compatibility.

The verification of the second claim is left as an exercise. ∎

Corollary 3.10.2 Let $\mathcal{B}$ terminate on ι, s. Then $\mathcal{B}\iota s$ is a β-normal form of s.

Exercise 3.10.3 Find a term s such that $\mathcal{B}$ does not terminate on ι, s. Try to find such a term that has a β-normal form.

Exercise 3.10.4 How would you modify $\mathcal{B}$ so that $\mathcal{B}\iota s$ is a $\beta\eta$-normal term if $\mathcal{B}$ terminates on ι, s?

Exercise 3.10.5 Let $\mathcal{B}_\alpha$ be the normalization procedure that is obtained with the renaming function ρ_α. Find a term s such that $\mathcal{B}_\alpha$ terminates on ι, s with a term that is not α-normal.

Exercise 3.10.6 Prove Proposition 3.10.1 (2).

3.11 Disgression: Computable Functions

In the 1930's Church was the first researcher who gave an explicit mathematical definition of the class of computable functions. His tool was the untyped lambda calculus, which is the system given by untyped terms and β-reduction. Church represented the natural numbers as the terms

$$\bar{0} := \lambda fx.x, \quad \bar{1} := \lambda fx.fx, \quad \bar{2} := \lambda fx.f(fx), \quad \bar{3} := \lambda fx.f(f(fx)), \quad \ldots$$

and defined a function $f \subseteq \mathbb{N} \times \mathbb{N}$ to be **λ-computable** if there is a closed term s such that the following conditions hold for all $m, n \in \mathbb{N}$:

1. f is defined on m iff $s\bar{m}$ has a β-normal form.
2. $fm = n$ iff $s\bar{m} \sim_\beta \bar{n}$.

We can see the term s as a procedure that computes the function f. Church's student Turing showed that a function is λ-computable if and only if it is computable with a Turing machine. This gave rise to the Church-Turing thesis, which states that the informal idea of computable function is captured by the formal definition based on lambda calculus or Turing machines. Given the link between β-equivalence and computability, we have the following theorem.

Theorem 3.11.1 (Undecidability) β-equivalence of terms is undecidable.

The terms $\bar{0}, \bar{1}, \bar{2}, \ldots$ representing the natural numbers are known as **Church numerals**. Church numerals are β-normal closed terms that can be understood as procedures. The Church numeral $\bar{n}$ is a procedure that takes a procedure f and a value x as arguments and applies f n-times to x. Based on this intuition, we can formulate procedures for addition, multiplication, and exponentiation of Church numerals:

$$\begin{array}{lll} add & := & \lambda mnfx.\, mf(nfx) \qquad\qquad m+n \\ mul & := & \lambda mnfx.\, m(nf)x \qquad\qquad m \cdot n \\ exp & := & \lambda mnfx.\, nmfx \qquad\qquad m^n \end{array}$$

On first view this looks like magic, but it is just higher-order functional programming where the natural numbers are represented as procedures. A programming-oriented explanation can be found in Pierce [54].

There is a connection with Example 3.7.7. The term T there is the Church numeral for 2. Now observe that $exp\,\bar{m}\,\bar{n} \sim_{\beta\eta} \bar{n}\,\bar{m}$. Thus $TT \sim_{\beta\eta} \overline{2^2}$, $TTT \sim_{\beta\eta} \overline{2^{2^2}}$, and so on. This informally explains why β-reduction of $TTTTTII$ takes so many steps.

In the lambda calculus, both procedures and values are expressed as closed terms. Since every closed term can be understood as a procedure, every value is expressed as a procedure. An interpreter for the lambda calculus takes a closed term $st_1 \ldots t_n$ and attempts to compute a β-normal form, where s is a procedure and $t_1, \ldots, t_n$ are the arguments the procedure is applied to. As an interpreter, the normalization procedure $\mathcal{B}\iota$ is flawed since it does not terminate for all terms that have a β-normal form. If we analyse the execution of $\mathcal{B}\iota u$, we obtain a reduction sequence $u \to_\beta u_1 \to_\beta u_2 \to_\beta \ldots$ where always the leftmost innermost β-redex is reduced. Curry proved in 1958 that leftmost-outermost β-reduction has the property that it always yields a normal form if there is one (see the quasi-leftmost-reduction theorem in Hindley-Seldin [42]).

Exercise 3.11.2 Write a procedure $\mathcal{L} : (\mathrm{Nam} \to \mathrm{Ter}) \to \mathrm{Ter} \to \mathrm{Ter}$ such that $\mathcal{L}\iota$ yields a β-normal form for every term that has a β-normal form.

3.12 Typed Terms

We now consider typed terms. Like terms types will be defined as syntactic objects. Every name will be equipped with a type. Following the inductive definition of terms, we obtain a unique type for every well-formed term. While names and abstractions are always well-formed, an application st is only well-formed if the

type of s is a function type whose argument type is the type of t. We will show that the normalization procedure terminates for every typed term. Hence every typed term has a β-normal form.

We start from a nonempty set of **base types** and define the set Ty of **types** inductively:

1. Every base type is a type.
2. If σ and τ are types, then $\sigma\tau$ is a type.

Types of the form $\sigma\tau$ are called **function types**. We omit parentheses according to $\sigma\tau\mu \rightsquigarrow \sigma(\tau\mu)$. A function type $\sigma\tau$ may also be written as $\sigma \to \tau$. We reserve the letter β for base types and the letters σ, τ, μ for types. The **size $|\sigma|$ of a type σ** is defined inductively:

$$|\_| \in \mathrm{Ty} \to \mathbb{N}$$
$$|\beta| = 1$$
$$|\sigma\tau| = 1 + |\sigma| + |\tau|$$

We assume that a set Nam of **names** is given and a bijection $\mathrm{Nam} \cong \mathbb{N} \times \mathrm{Ty}$. Due to the bijection every name has a unique type, and for every type there are infinitely many names that have this type. We now define the set Ter of **typed terms** inductively, where every term is equipped with a unique type:

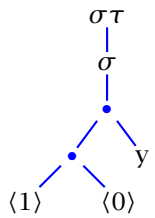
1. Every name of type σ is a term of type σ.
2. If x is a name of type σ and s is term of type τ, then $\lambda x.s$ is a term of type $\sigma\tau$.
3. If s is a term of type $\tau\mu$ and t is a term of type τ, then st is a term of type μ.

We write $s : \sigma$ if s is a term of type σ. Using this notation, we can summarize the definition of typed terms with the following rules:

$$\frac{}{x : \sigma}\ x \text{ has type } \sigma \qquad \frac{x : \sigma \qquad s : \tau}{\lambda x.s : \sigma\tau} \qquad \frac{s : \tau\mu \qquad t : \tau}{st : \mu}$$

We use Ter_σ to denote the set of all terms of type σ. When we write term in the following, we always mean typed term. We reserve the letters x, y, z for names and the letters s, t, u, v for terms.

The definitions we made for untyped terms all carry over to typed terms. The LNR of a typed term records the types of the local names at its λ-nodes. For instance, if $f : \sigma\tau$ and $x : \sigma$, the LNR of $\lambda fx.fxy$ looks as follows:



Every subterm of a typed term is a typed term. As it comes to substitutions, we only admit type-preserving functions $\theta \in \mathrm{Nam} \to \mathrm{Ter}$, that is, that is, θx must have the same type as x for every name x. Moreover, every renaming function ρ must be type-preserving, that is, $\rho\theta x s$ must have the same type as x. As a consequence, the substitution and normalization operators become well-defined for typed terms and are themselves type-preserving functions.

Proposition 3.12.1 (Type Preservation)

1. If $s : \sigma$, then $S\theta s : \sigma$.
2. If $\mathcal{B}$ terminates on θ, s and $s : \sigma$, then $\mathcal{B}\theta s : \sigma$.
3. If $s : \sigma$ and $s \to_\beta t$ or $s \to_\eta t$, then $t : \sigma$.
4. If $s : \sigma$ and $s \sim_\alpha t$ or $s \sim_\beta t$ or $s \sim_{\beta\eta} t$, then $t : \sigma$.

Exercise 3.12.2 Find closed typed terms that have the following types.

a) $\sigma\sigma$

b) $\sigma\tau\sigma$

c) $(\sigma\tau)(\tau\mu)\sigma\mu$

d) $\sigma(\sigma\tau)\tau$

Exercise 3.12.3 For each of the following terms find types for the names occurring in the term such that the term becomes a well-formed typed term.

a) $\lambda xy.x$

b) $\lambda f.fyx$

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

3.13 Termination of Typed Normalization

We will now show that the normalization procedure $\mathcal{B}$ defined in Figure 3.1 terminates for typed terms. As stated before, we assume that $\mathcal{B}$ is only applied to type-preserving substitutions θ and that the underlying renaming function is type-preserving. Given these assumptions, it is easy to check that $\mathcal{B}$ is well-defined and type-preserving as stated in Proposition 3.12.1 (2).

We call terms of the form $xs_1 \ldots s_n$ **concrete** ($n \geq 0$). We refer to the name x as the **head of the term**. Given a term s, we use τs to denote the size of the type of s. Given a concrete term s whose head is x, we have $\tau x \geq \tau s$.

Theorem 3.13.1 (Termination)
$\mathcal{B}$ terminates on β-normal type-preserving substitutions and typed terms.

Proof Given a β-normal type-preserving substitution θ and a typed term s, we define three natural numbers:

- The *sign of* s, which is 0 if s is β-normal and 1 otherwise.
- The *power of* θ, which is the maximal number τx such that $x \in \mathcal{N} s$ and θx is an abstraction. If no such name x exists, the power of θ is 0.
- The *size of* s.

The corresponding lexical order is a termination order for $\mathcal{B}$. We verify the claim for each of the defining equations of $\mathcal{B}$.

1. *equation.* The claim is trivial since there is no recursive call.
2. *equation.* The claim holds since the sign and the power are preserved and the size is decreased.
3. *equation.* The recursive calls $\mathcal{B}\theta s$ and $\mathcal{B}\theta t$ obey the termination order since the sign and the power are not increased, and the size is decreased. Let $\mathcal{B}\theta s = \lambda x.u$. It remains to show that the recursive call $\mathcal{B}[x{:=}\mathcal{B}\theta t]u$ obeys the termination order. Case analysis.
   - st not β-normal. Then the sign is decreased since u is β-normal (Proposition 3.10.1).
   - st is β-normal. Then s is concrete. Since $\mathcal{B}\theta s = \lambda x.u$, θ must replace the head of s with an abstraction. Hence the power of θ is at least τs. Since $\mathcal{B}$ is type-preserving (Proposition 3.12.1) and $\mathcal{B}\theta s = \lambda x.u$, we have $\tau s > \tau x$. Hence the power of θ is greater than the power of $[x{:=}\mathcal{B}\theta t]$. The claim follows since the sign is preserved and the power is decreased.
4. *equation.* The claim holds since the sign and the power are not increased, and the size is decreased. ∎

Corollary 3.13.2 (Weak Normalization) Every typed term has a β- and a $\beta\eta$-normal form.

Proof Let s be a typed term. Since ι is type-preserving, we know by Theorem 3.13.1 that $\mathcal{B}$ terminates on ι, s. Hence we know by Proposition 3.10.1 that $\mathcal{B}\iota s$ is a β-normal form of s. Next we apply η-reduction to $\mathcal{B}\iota s$. Since η-reduction terminates and preserves β-normality, we obtain a $\beta\eta$-normal term that is $\beta\eta$-equivalent to s. ∎

Corollary 3.13.3 β- and $\beta\eta$-equivalence of typed terms are decidable.

Proof Follows from Corollary 3.13.2 and the normal form theorems 3.7.1 and 3.8.3. ∎

Statman [63] shows that deciding β-equivalence of typed terms has non-elementary complexity.

One can show that β-reduction terminates on typed terms no matter how the reduction steps are applied. This result is known as **strong normalization**. Strong normalization is much harder to prove than weak normalization. Several proofs of strong normalization appeared around 1967 (see [7] for references). The proof that generalizes best to richer systems is due to Tait [65]. You can find Tait's proof in [42]. Weak normalization was first shown by Turing [29].

Exercise 3.13.4 Identify where the proof of the termination theorem breaks if we drop the assumption that θ is β-normal.

Exercise 3.13.5 Modify the normalization procedure $\mathcal{B}$ such that it yields β-normal terms even if θ is not β-normal. Give a termination proof for the modified procedure. Hint: Define the sign of θ to be 0 if θx is β-normal for all $x \in \mathcal{N}s$ and 1 otherwise.

Exercise 3.13.6 Write a normalization procedure $\mathcal{L}$ as asked for in Exercise 3.11.2 and prove that it terminates for all type-preserving substitutions and all typed terms.

3.14 Remarks

Syntactic systems based on terms with lambda abstractions are known as **lambda calculi**. We have looked at the **untyped lambda calculus** and the **simply typed lambda calculus**. Both systems originated with Church [18, 20]. There are lambda calculi with much richer type systems [8]. Hindley-Seldin [42] is an excellent textbook that covers both the untyped and the simply typed lambda calculus and gives many historical references. Other books on lambda calculus are Barendregt [7] and Hindley [41].

The lambda calculus is an essential cornerstone of the theory of programming languages. Pierce [54] gives a programming languages oriented introduction to the lambda calculus. A more advanced textbook on the theory of programming languages that covers lambda calculus is Mitchell [50].

The formal definition of α-equivalence based on S_α is due to Stoughton [64]. In this paper you find a careful study of S_α with complete proofs. We have generally omitted proofs since they require considerable technical detail. You find proofs of the basic properties of the lambda calculus in Hindley-Seldin [42].

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 [23], which was conceived as an implementation of terms.

4 Interpretation and Specification

We now formalize the semantic interpretation of types and typed terms. We will define formulas and logical interpretations and say what it means that an interpretation satisfies a formula. This way we obtain *simple type theory*, the logic we will be working with. Simple type theory can specify the natural numbers, and we will use this example to study the method of logical specification.

4.1 Interpretations

Recall that Ty is the set of all types and Ter is the set of all typed terms. We assume that Ty and Ter are disjoint. We will only consider typed terms.

We start with the interpretation of types. Every base type is interpreted as a nonempty set. Moreover, a function type $\sigma\tau$ is interpreted as a set of functions from the interpretation of σ to the interpretation of τ.

Definition. A **frame** is a function $\mathcal{D}$ defined on Ty that maps every type to a nonempty set and satisfies $\mathcal{D}(\sigma\tau) \subseteq (\mathcal{D}\sigma \to \mathcal{D}\tau)$ for every function type $\sigma\tau$. A **standard frame** is a frame $\mathcal{D}$ where $\mathcal{D}(\sigma\tau) = (\mathcal{D}\sigma \to \mathcal{D}\tau)$ for every function type $\sigma\tau$.

We now come to the interpretation of names. The interpretation of a name must respect its type, that is, a name must be interpreted as an element of the interpretation of its type.

Definition. An **assignment into a frame $\mathcal{D}$** is a function $\mathcal{I}$ defined on Ty $\cup$ Nam such that $\mathcal{D} \subseteq \mathcal{I}$ and $\mathcal{I}x \in \mathcal{I}\sigma$ for every name $x : \sigma$.

Given an assignment $\mathcal{I}$ into a frame $\mathcal{D}$ we have $\mathcal{I}\sigma = \mathcal{D}\sigma$ for all types σ. Thus an assignment builds in the frame.

Given an assignment $\mathcal{I}$, we are able to evaluate every λ-free term $t : \sigma$ to an element of $\mathcal{I}\sigma$. This follows from the equations

$$\hat{\mathcal{I}}x = \mathcal{I}x$$
$$\hat{\mathcal{I}}(st) = (\hat{\mathcal{I}}s)(\hat{\mathcal{I}}t)$$

and a straightforward inductive proof. Note that the **evaluation function $\hat{\mathcal{I}}$** is obtained from the assignment $\mathcal{I}$ by induction on terms.

To discuss the evaluation of abstractions we need a definition. Let $\mathcal{I}$ be an assignment into a frame $\mathcal{D}$, $x : \sigma$ be a name, and $a \in \mathcal{I}\sigma$. Then $\mathcal{I}^x_a$ denotes the assignment into $\mathcal{D}$ that agrees everywhere with $\mathcal{I}$ but possibly on x where it yields a.

It seems that we can extend the definition of the evaluation function to abstractions using the following equation:

$$\hat{\mathcal{I}}(\lambda x.s) = \lambda a{\in}\mathcal{I}\sigma.\ \widehat{\mathcal{I}^x_a}s \qquad \text{if } \lambda x.s : \sigma\tau$$

The function at the right-hand side certainly exists (s is smaller than $\lambda x.s$, so we know by induction that $\widehat{\mathcal{I}^x_a}s$ is defined). We also know that it is a function $\mathcal{I}\sigma \to \mathcal{I}\tau$. However, if may be that $\mathcal{I}(\sigma\tau)$ does not contain the function if the frame of $\mathcal{I}$ is nonstandard. So we learn that an assignment into a non-standard frame may fail to evaluate all terms because certain abstractions may describe functions that are not in the frame. Such nonstandard frames are unwanted and we will exclude them. To do so with a rigorous definition, we first define possibly partial evaluation functions for all assignments.

Definition. Let $\mathcal{D}$ be a frame. We define a function $\hat{}$ that maps every assignment $\mathcal{I}$ into $\mathcal{D}$ into a function $\hat{\mathcal{I}} \subseteq \{\,(s,a) \mid \exists\sigma\colon\ s \in \mathrm{Ter}_\sigma \wedge a \in \mathcal{D}\sigma\,\}$:

$$\begin{aligned}
\hat{\mathcal{I}}x &:= \mathcal{I}x \\
\hat{\mathcal{I}}(st) &:= fa \qquad \text{if } \hat{\mathcal{I}}s = f \text{ and } \hat{\mathcal{I}}t = a \\
\hat{\mathcal{I}}(\lambda x.s) &:= f \qquad \text{if } \lambda x.s : \sigma\tau,\ f \in \mathcal{D}(\sigma\tau),\ \text{and } \forall a \in \mathcal{D}\sigma\colon\ \widehat{\mathcal{I}^x_a}s = fa
\end{aligned}$$

The definition is by induction on terms. We call $\hat{\mathcal{I}}$ the **evaluation function of** $\mathcal{I}$. We say that an assignment $\mathcal{I}$ **evaluates a term** s if $\hat{\mathcal{I}}$ is defined on s.

Definition. An **interpretation** is an assignment that evaluates every term. A **standard interpretation** is an interpretation whose restriction to types is a standard frame. A frame $\mathcal{D}$ is **admissible** if every assignment into $\mathcal{D}$ is an interpretation.

Proposition 4.1.1 Every standard frame is admissible.

Proposition 4.1.2 (Coincidence) Let $\mathcal{I}$ and $\mathcal{J}$ be assignments that agree on all types and on the free names of s. Then the evaluation functions $\hat{\mathcal{I}}$ and $\hat{\mathcal{J}}$ agree on s (either both yield the same value or both are undefined).

Proof By induction on s. ∎

Proposition 4.1.3 A frame $\mathcal{D}$ is admissible if there is an assignment into $\mathcal{D}$ that evaluates every closed term.

Proof By Proposition 4.1.2 we know that every assignment into $\mathcal{D}$ evaluates every closed term.

Let $\mathcal{I}$ be an assignment that evaluates every closed term. We show by induction on $|\mathcal{N}s|$ (the number of free names) that $\mathcal{I}$ evaluates every term s. If $|\mathcal{N}s| = 0$, the claims follows by assumption since s is closed. Otherwise, let $x \in \mathcal{N}s$. By the inductive hypothesis we know that there is a function f such that $\hat{\mathcal{I}}(\lambda x.s) = f$. Hence we have $\hat{\mathcal{I}}((\lambda x.s)x) = f(\mathcal{I}x)$. By the definition of $\hat{\mathcal{I}}$ for abstractions we know that $\widehat{\mathcal{I}^x_{\mathcal{I}x}}$ is defined on s. Hence $\hat{\mathcal{I}}$ is defined on s. ∎

Given an admissible frame, all assignments into the frame evaluate all terms according to the evaluation equations we have for standard interpretations.

Proposition 4.1.4 For every interpretation $\mathcal{I}$:

1. $\hat{\mathcal{I}}x = \mathcal{I}x$
2. $\hat{\mathcal{I}}(st) = (\hat{\mathcal{I}}s)(\hat{\mathcal{I}}t)$
3. $\hat{\mathcal{I}}(\lambda x.s) = \lambda a{\in}\mathcal{I}\sigma.\ \widehat{\mathcal{I}^x_a}s \quad$ if $x : \sigma$

If two interpretations agree on all free names of a term, they don't necessarily evaluate the term to the same value. Consider for instance the term $\lambda x.x$ and two interpretations that disagree on the type of x. From this example we learn that we have to look at the types of bound names if we want to formulate a coincidence property for interpretations. We define the **footprint of a term** s to be the set $\mathcal{N}s \cup \Sigma$ where Σ is the set of all types σ such that s contains an abstraction $\lambda x.t$ such that $x : \sigma$.

Proposition 4.1.5 (Coincidence)
$\hat{\mathcal{I}}s = \hat{\mathcal{J}}s$ provided $\mathcal{I}$ and $\mathcal{J}$ are interpretations that agree on the footprint of s.

Proof By induction on s. ∎

The definition of interpretations and evaluation could be much simplified if we only considered standard frames. In fact, as it comes to mathematical statements (see Chapter 2), standard frames suffice. However, we will need admissible nonstandard frames for the formulation of some of the main technical results.

Exercise 4.1.6 Argue that all assignments into a frame are interpretations if at least one assignment into the frame is an interpretation.

Exercise 4.1.7 Let $\mathcal{D}$ and $\mathcal{E}$ be frames such that $\mathcal{D}(\sigma\tau) \subsetneq \mathcal{E}(\sigma\tau)$ for some function type $\sigma\tau$. Explain why $\mathcal{D}((\sigma\tau)\mu)$ and $\mathcal{E}((\sigma\tau)\mu)$ are disjoint sets.

4.2 Semantic Equivalence

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

$$s \approx t \;:\Longleftrightarrow\; s \text{ and } t \text{ have the same type and } \hat{\mathcal{I}}s = \hat{\mathcal{I}}t \text{ for every interpretation } \mathcal{I}$$

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.

Theorem 4.2.1 $s \approx t \iff s \sim_{\beta\eta} t$

The theorem states that semantic equivalence coincides with $\beta\eta$-equivalence. This means that the β- and η-law are valid for semantic equivalence and together fully capture semantic equivalence. For now we are happy to state the theorem; a proof is beyond the scope of this chapter.

The two directions of the theorem are known as soundness and completeness. **Soundness** is the direction $\Longleftarrow$, that is $\sim_{\beta\eta} \subseteq \approx$. **Completeness** is the direction $\Longrightarrow$, that is $\approx \subseteq \sim_{\beta\eta}$. Soundness is the property we used to justify the α-, β- and η-law. We also used soundness to argue that substitution must not capture. Hence it is no surprise that soundness holds. Completeness, on the other hand, says that the β- and $\eta-$ law capture all properties of the λ-notation as it comes to semantic equivalence. This is an interesting and non-obvious fact.

One can show that two terms of the same type are semantically equivalent if and only if they evaluate to the same value in every standard interpretation (Friedman's theorem [28]). Hence there is no need to consider nonstandard interpretations as it comes to semantic equivalence.

Corollary 4.2.2 Two λ-free terms are semantically equivalent if and only if they are identical.

Proof Let s and t be λ-free terms that are semantically equivalent. Then s and t are $\beta\eta$-normal and $\beta\eta$-equivalent. By Theorem 3.8.3 we know that s and t are α-equivalent. Since s and t are λ-free, they must be identical. ∎

Exercise 4.2.3 Argue that from the results stated it follows that semantic equivalence is decidable.

Exercise 4.2.4 Argue that from the results stated it follows that semantic equivalence is compatible and stable.

Exercise 4.2.5 Find terms s and t such that $s \approx t$ and $s \not\sim_\beta t$.

$$
\begin{aligned}
\mathcal{I}o &= \{0,1\}\\
\mathcal{I}\bot &= 0\\
\mathcal{I}\top &= 1\\
\mathcal{I}\neg &= \lambda a{\in}\mathcal{I}o.\ \text{if } a{=}0 \text{ then 1 else 0}\\
\mathcal{I}(\vee) &= \lambda a{\in}\mathcal{I}o.\ \lambda b{\in}\mathcal{I}o.\ \text{if } a{=}0 \text{ then } b \text{ else 1}\\
\mathcal{I}(\wedge) &= \lambda a{\in}\mathcal{I}o.\ \lambda b{\in}\mathcal{I}o.\ \text{if } a{=}0 \text{ then 0 else } b\\
\mathcal{I}(\to) &= \lambda a{\in}\mathcal{I}o.\ \lambda b{\in}\mathcal{I}o.\ \text{if } a{=}0 \text{ then 1 else } b\\
\mathcal{I}(=_\sigma) &= \lambda a{\in}\mathcal{I}\sigma.\ \lambda b{\in}\mathcal{I}\sigma.\ \text{if } a{=}b \text{ then 1 else 0}\\
\mathcal{I}(\exists_\sigma) &= \lambda f{\in}\mathcal{I}(\sigma o).\ \text{if } f{=}(\lambda a{\in}\mathcal{I}\sigma.0) \text{ then 0 else 1}\\
\mathcal{I}(\forall_\sigma) &= \lambda f{\in}\mathcal{I}(\sigma o).\ \text{if } f{=}(\lambda a{\in}\mathcal{I}\sigma.1) \text{ then 1 else 0}
\end{aligned}
$$

Figure 4.1: Requirements for logical interpretations

4.3 Simple Type Theory

We are now ready to define the logic we will be working with. We fix a base type o for the truth values and names for the logical operations:

$$
\begin{aligned}
\bot, \top &: o && \\
\neg &: oo && \\
\wedge, \vee, \to &: ooo && \\
=_\sigma &: \sigma\sigma o && \text{for every type } \sigma\\
\forall_\sigma, \exists_\sigma &: (\sigma o)o && \text{for every type } \sigma
\end{aligned}
$$

We call the names for the logical operations **logical constants**, and all other names **variables**. Since there are no other constants but logical constants, we often just say constant if we mean logical constant. Terms whose type is o are called **formulas**. The base type o is pronounced like the latin letter o. We refer to the base types that are different from o as **sorts** and reserve the letter α for sorts. We fix a sort ι (pronounced like the latin letter i) to be used in examples.

A **logical interpretation** is an interpretation that interprets o as the set $\{0,1\}$ and the constants as one would expect. Formally, we require that a logical interpretation $\mathcal{I}$ satisfies the conditions stated in Figure 4.1. Given a logical interpretation, formulas express mathematical statements about the objects provided by the interpretation. Given a formula s and a logical interpretation $\mathcal{I}$, we say *s is true in $\mathcal{I}$* if $\hat{\mathcal{I}}s = 1$, and *s is false in $\mathcal{I}$* if $\hat{\mathcal{I}}s = 0$.

We have now set up a logic that represents mathematical statements as terms and accounts for their meaning through interpretations. In particular, we have a formal definition of what it means that a mathematical statement is true. The logic agrees with the presentation in Chapter 2. There the notion of an interpretation is not explicit, but the way the interpretations of types and names are described certainly agrees with the notion of a logical standard interpretation.

We call the logical language given by typed terms, formulas and logical interpretations **STT**, which abbreviates **simple type theory**. The name simple type theory originated with Church [20], who presented the first STT-like logic.

Proposition 4.3.1 (Logical Coincidence) $\hat{\mathcal{I}}s = \hat{\mathcal{J}}s$ provided $\mathcal{I}$ and $\mathcal{J}$ are logical interpretations that agree on all types and all variables in the footprint of s.

Proof Straightforward consequence of Proposition 4.1.5. ∎

We now define a variety of technical terms and notations that will be useful in the following. We use $\mathcal{V}s$ to denote the set of all variables that are free in S. Note that $\mathcal{N}s - \mathcal{V}s$ contains exactly the constants that occur in s. We use the notations $\mathcal{N}_\sigma s := \{x \in \mathcal{N}s \mid x : \sigma\}$ and $\mathcal{V}_\sigma s := \{x \in \mathcal{V}s \mid x : \sigma\}$. We use $\equiv$ to denote the constant $=_o$. The constants $\bot$, $\top$, $\neg$, $\wedge$, $\vee$, $\to$, $\equiv$ are called **propositional constants**, the constants $=_\sigma$ are called **identities**, and the constants $\forall_\sigma$ and $\exists_\sigma$ are called **quantifiers**. A formula is called **identity-free** if it contains no identity, and **quantifier-free** if it contains no quantifier. Formulas of the form $s =_\sigma t$ are called **equations**. The constants $=_{\sigma\tau}$, $\forall_{\sigma\tau}$, and $\exists_{\sigma\tau}$ where σ and τ are types are referred to as **higher-order constants**. Note that high-order identities tests identity of functions, and that higher-order quatifiers quantify over functions. A **higher-order formula** is a formula that contains a higher-order constant.

A **relational type** is a type that can be obtained with the grammar $\sigma ::= \beta \mid \sigma \ldots \sigma o$ where β ranges over base types. Roughly speaking, relational types do not involve functions into sorts, but just functions into o. A type of the form $\sigma_1 \ldots \sigma_n \alpha$ where $n \geq 1$ is always non-relational. A **relational name** is a name whose type is relational. Note that all propositional constants are relational. The identity $=_\iota$ and the quantifier $\forall_\iota$ are examples for non-relational constants. A term is **relational** if its type is relational and all its free names are relational.

We use the word **model** as a synonym for logical interpretation. By a **standard model** we mean a logical standard interpretation, and by a **nonstandard model** we mean a logical nonstandard interpretation. We write $\mathcal{I} \models s$ if and only if $\mathcal{I}$ is a model and s is a formula that is true in $\mathcal{I}$. Given a model $\mathcal{I}$ and a formula s, either $\mathcal{I} \models s$ or $\mathcal{I} \models \neg s$, and $\mathcal{I} \not\models s$ if and only if $\mathcal{I} \models \neg s$. We express $\mathcal{I} \models s$ equivalently by saying $\mathcal{I}$ **satisfies** s, or $\mathcal{I}$ **is a model of** s.

Let A be a set of formulas. We say a logical interpretation $\mathcal{I}$ is a **model of** A if $\mathcal{I}$ satisfies every formula $s \in A$. We write $\mathcal{I} \models A$ to say that $\mathcal{I}$ is a model of A.

Moreover, we use the notations

$$\mathcal{N}_\sigma A := \{ x \mid \exists s \in A\colon\ x \in \mathcal{N}_\sigma s \}$$
$$\mathcal{V}_\sigma A := \{ x \mid \exists s \in A\colon\ x \in \mathcal{V}_\sigma s \}$$

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 $=_o$, and $s \not\equiv t$ for $\neg(s \equiv t)$. Moreover, we adopt the notational conventions stated in § 2.3 and arrange the following conventions for the quantifiers:

$$\begin{array}{rcl@{\qquad}rcl} \forall x.s & \rightsquigarrow & \forall(\lambda x.s) & \exists x.s & \rightsquigarrow & \exists(\lambda x.s) \\ \forall xy.s & \rightsquigarrow & \forall(\lambda x.\forall(\lambda y.s)) & \exists xy.s & \rightsquigarrow & \exists(\lambda x.\exists(\lambda y.s)) \\ \forall x\exists y.s & \rightsquigarrow & \forall(\lambda x.\exists(\lambda y.s)) & \exists x\forall y.s & \rightsquigarrow & \exists(\lambda x.\forall(\lambda y.s)) \end{array}$$

Exercise 4.3.2 Find formulas that satisfy the following properties.

a) A logical interpretation satisfies the formula if and only if it interprets the sort ι as a set that has exactly one element.

b) A logical interpretation satisfies the formula if and only if it interprets the sort ι as a set that has at most two elements.

Exercise 4.3.3 Let $x : \iota$, $y : \iota$, $z : \iota$, and $f : \iota o$ be different variables. Say for each of the following formulas what it means that a logical standard interpretation satisfies it.

a) $fx \to fy$

b) $x \neq y \to fx \to fy \to fz$

Why is the meaning of the formulas not obvious for logical interpretations that are not standard?

Exercise 4.3.4 Let $U : (\sigma o)o$ be a variable and $\mathcal{I}$ be a logical interpretation that satisfies the formulas $U(\lambda x.\top)$ and $\forall f x.\, Uf \to fx$.

a) Is the function $\mathcal{I}U$ uniquely determined?

b) Can you describe the function $\mathcal{I}U$ explicitly?

Exercise 4.3.5 Let $U, E : (\sigma o)o$ be variables. Find formulas that employ no other constants but $\bot$, $\top$, $\neg$, $\to$, and $=_{(\sigma o)\sigma o}$ and satisfy the following properties. Take Exercise 4.3.4 for inspiration.

a) An interpretation satisfies the formula if and only if it interprets U as the universal quantifier at σ.

b) An interpretation satisfies the formula if and only if it interprets E as the existential quantifier at σ.

Exercise 4.3.6 STT comes with a redundant set of constants.

a) Specify $\bot$, $\neg$, $\top$, $\forall_\sigma$, $\exists_\sigma$, $\wedge$, $\vee$, and $\to$ just using the identities $=_\sigma$. That is, find terms that use no other constants but identities and that every logical interpretation evaluates to the values the listed constants are evaluated to.

b) Assume $\to$ and $\forall_{\sigma o}$ and specify $=_\sigma$.

Hint: Review Chapter 2.

4.4 Specification of the Natural Numbers

STT provides a fair collection of logical operations but comes without the natural numbers. We will now show that STT can specify (i.e., express) the natural numbers for standard interpretations. The specification will use ideas that first appeared in 1889 in Peano's axiomatization of the natural numbers [53]. Read the article *Peano axioms* in Wikipedia to know more.

We start from infinite graphs of the form $\circ\to\circ\to\circ\to\cdots$ which we call chains. Each chain is a faithful representation of the natural numbers: The initial node is 0, the successor of the initial node is 1, and so on. We formalize chains as follows.

A **chain** is a triple (N, o, S) such that N is an infinite set, $o \in N$, $S \in N \to N$, and $N = \{o, So, S(So), \dots\}$. We call o the **origin** and S the **successor function** of the chain.

Here are two examples for how we can construct a chain:

1. $N = \mathbb{N}$, $o = 0$, $S = \lambda n{\in}N.\ n + 1$.
2. $N = \{\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \dots\}$, $o = \emptyset$, $S = \lambda n{\in}N.\ \{n\}$.

The definition of chains is not in a form that can be expressed in simple type theory. Thus we need an equivalent characterization of chains that can be expressed in simple type theory. Given a function $f \in X \to X$, we call a set $P \subseteq X$ **f-closed** if $fx \in P$ whenever $x \in P$. Given a chain, a set of nodes is S-closed if and only if it is closed under reachability.

Proposition 4.4.1 Let $o \in N$ and $S \in N \to N$. Then (N, o, S) is a chain if and only if $o \notin \operatorname{Ran} S$, S is injective, and $P = N$ for every S-closed set $P \subseteq N$ that contains o.

Proof We can picture (N, o, S) as a graph. Since S is a function $N \to N$, every node has exactly one outgoing edge. The condition $o \notin \operatorname{Ran} S$ means that o has no incoming edge, and injectivity of S means that every node has at most one incoming edge. Hence we know that the nodes reachable from o yield a chain.

The third condition makes sure that there are no nodes that are not reachable from o. ∎

The new characterization of chains translates directly into STT. We fix a sort N and names $o : N$ and $S : NN$ and obtain the formula chain as the conjuction of three formulas:

1. $\forall x.\ Sx \neq o$
2. $\forall xy.\ Sx = Sy \to x = y$
3. $\forall p.\ po \to (\forall x.\ px \to p(Sx)) \to (\forall x.px)$

We say that the formula chain specifies chains and refer to the three subformulas of chain as the **axioms of the specification**.

Proposition 4.4.2 A logical standard interpretation $\mathcal{I}$ satisfies the formula chain if and only if $(\mathcal{I}N, \mathcal{I}o, \mathcal{I}S)$ is a chain.

The third axiom of chain is known as the **induction axiom** since it justifies natural induction: To show that a property p holds for all numbers, show that it holds for 0 and that it holds for $x + 1$ whenever it holds for $x \in \mathbb{N}$.

Given o and S, it is easy to specify addition. We choose a name $+ : NNN$ and specify addition by the conjunction of two formulas:

1. $\forall y.\ o + y = y$
2. $\forall xy.\ Sx + y = x + Sy$

The formulas specify addition by induction on the first argument. The specification exploits that every natural number has either the form o or Sx, and that we can express the sum $Sx + y$ as the sum $x + Sy$ whose first argument is smaller (i.e., closer to the origin). From the perspective of programming, one would say that one has defined a procedure for addition that computes by recursion on its first argument. Inductive definitions have been used in mathematics for a long time. They are also fundamental for programming, where one speaks of recursive definitions (i.e., recursive procedures, recursive data types). In case you don't feel comfortable with inductive definitions, get yourself acquainted with functional programming. (There are plenty of textbooks, ML and Haskell are the most popular languages.)

Given addition, we can write a term leq : NNo that evaluates to a predicate that tests whether its first argument is less or equal than its second argument:

$$\mathsf{leq} := \lambda xy.\ \exists z.\ x + z = y$$

Now we can write a term finite : $(No)o$ that evaluates to a predicate that tests whether a set of natural numbers is finite:

$$\mathsf{finite} := \lambda p.\ \exists x\ \forall y.\ py \to \mathsf{leq}\, y\, x$$

Exercise 4.4.3 (Multiplication) Specify multiplication $\cdot : NNN$ using addition.

Exercise 4.4.4 (Order) Specify the less-or-equal predicate $\leq$: NNo for natural numbers in three different ways:

a) Inductively using $\forall_N$.

b) With $\exists_N$ using addition.

c) With $\forall_{No}$ quantifying over S-closed sets.

Exercise 4.4.5 (Induction Axiom) From the proof of Proposition 4.4.1 you know that the induction axiom in the specification of chains is needed to exclude nodes that are not reachable from the origin (so-called junk nodes).

a) Draw examples of graphs that satisfisfy the first and second axiom for chains but contain junk nodes.

b) The induction axiom can be replaced by other formulas. One possibility is $\forall p.\, \exists p \to (\forall x.\, p(Sx) \to px) \to p0$. Draw pictures to understand what sets of nodes this formula excludes.

c) Can you think of other formulas that can replace the induction axiom?

d) Explain why $\forall p.\, (\forall x.\, p(Sx) \to px) \to p0$ cannot replace the induction axiom.

Exercise 4.4.6 (Finiteness) Let $f : \sigma\sigma$ be a variable.

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

b) Find a term $\mathsf{surjective} : (\sigma\sigma)o$ such that a logical standard 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 standard interpretation if and only if it interprets the type σ as a finite set. Hint: A set X is finite if and only if every injective function $X \to X$ is surjective.

Exercise 4.4.7 Find an infinite set A of formulas such that every finite subset of A has a standard model but A does not have a standard model.

Exercise 4.4.8 (Transitive Closure) Let $R \subseteq X \times X$. The *transitive closure of* R is $R^+ := \{\, (x,y) \in X \times X \mid \forall R' \subseteq X \times X\colon\ R' \text{ transitive} \ \wedge\ R \subseteq R' \implies (x,y) \in R' \,\}$. In words we can say that R^+ is defined as the intersection of all transitive relations that contain R. One can show that R^+ is the least transitive relation that contains R.

In STT we represent relations as functions $\iota\iota o$. Give a term $TC : (\iota\iota o)\iota\iota o$ that in logical standard interpretations evaluates to a function that yields the transitive closure of a relation. Use the variables $x, y, z : \iota$ and $r, r' : \iota\iota o$.

Exercise 4.4.9 (Termination) A binary relation R is *terminating* if there exists no infinite sequence $x_0, x_1, x_2, \ldots$ such that $(x_n, x_{n+1}) \in R$ for all $n \in \mathbb{N}$. Let $r : \sigma\sigma o$ be a variable. Find a term $\mathsf{terminates} : (\sigma\sigma o)o$ such that a logical standard interpretation satisfies the formula $\mathsf{terminates}\, r$ if and only if it interprets r as the characteristic function of a terminating relation. Do not use the natural numbers. Hint: A relation R does not terminate if and only if there exists a nonempty R-closed subset of $\mathrm{Dom}\, R$.

Exercise 4.4.10 (Pairs) Let variables $\mathsf{pair} : \sigma\tau\alpha$, $\mathsf{fst} : \alpha\sigma$, and $\mathsf{snd} : \alpha\tau$ be given. Find a formula that is satisfied by a logical standard interpretation $\mathcal{I}$ if and only if $\mathcal{I}\alpha$ is in bijection with the set $\mathcal{I}\sigma \times \mathcal{I}\tau$ and pair, fst, and snd are interpreted as the pairing and projection functions.

Exercise 4.4.11 (Lists) Let variables $\mathsf{nil} : \alpha$, $\mathsf{cons} : \sigma\alpha\alpha$, $\mathsf{hd} : \alpha\sigma$, and $\mathsf{tl} : \alpha\alpha$ be given. Find a formula that is satisfied by a logical standard interpretation if and only if α represents all lists over σ and nil, cons, hd, and tl represent the list operations. Make sure α contains only elements that are reachable from nil by finitely many applications of cons.

4.5 Validity, Satisfiability, Logical Equivalence

A formula is **valid** if it is true in every logical interpretation, and **satisfiable** if it is true is some logical interpretation. A formula is **unsatisfiable** if it is not satisfiable. Valid formulas express logical laws (e.g. $\neg\neg x = x$). A satisfiable formula expresses a fact that is true in at least one logical interpretation. An unsatisfiable formula expresses a statement that is false in every logical interpretation (e.g., $\neg x = x$).

Proposition 4.5.1 A formula s is valid iff its negation $\neg s$ is unsatisfiable.

Valid formulas express properties of the logical operations. Since we can specify domains like, for instance, the natural numbers with formulas, we can express properties of specifiable domains as validity properties. For instance, take the formula chain, which specifies the natural numbers, and a formula s that expresses some property that involves the natural numbers. We know the following:

1. If the formula $\mathsf{chain} \to s$ is valid, s is true in every standard model of chain.
2. If the formula $\mathsf{chain} \to \neg s$ is valid, s is false in every standard model of chain.
3. Up to representation a standard model of chain interprets N as the set of natural numbers, o as zero, and S as the successor function $\lambda n{\in}\mathbb{N}.\ n + 1$.

Thus, if we have an algorithm that verifies the validity of formulas, we can use this algorithm to verify properties of the natural numbers.

A formula is **weakly valid** if it is true in every standard model. Clearly, every valid formula is weakly valid. On the other hand, we will show in § 10.2 that there are weakly valid formulas that are not valid.

Two terms are **logically equivalent** if they have the same type and evaluate to the same value for every logical interpretation. Note that semantically equivalent terms are always logically equivalent. Logic equivalennce reduces to validity.

Proposition 4.5.2 Let s and t be terms of the same type. Then s and t are logically equivalent iff the formula $s = t$ is valid.

Proposition 4.5.3 Logic equivalence is a compatible and stable relation on terms.

Exercise 4.5.4 Find an unsatisfiable formula sx such that $\exists s$ is satisfiable.

Exercise 4.5.5 Find an unsatisfiable formula $sx \neq tx$ such that $s \neq t$ is satisfiable.

Exercise 4.5.6 Find a satisfiable formula that has no free variable and is not valid.

Exercise 4.5.7 Find terms that are logically equivalent but not semantically equivalent.

Exercise 4.5.8 Argue the semantic equivalence of the formulas $\forall f$ and $\forall x.fx$.

Exercise 4.5.9 Determine for each of the following statements whether it is true or false. If the statement is false, give a counterexample.

a) Either s is satisfiable or $\neg s$ is satisfiable.

b) Either s is valid or $\neg s$ is valid.

c) Either s is valid or $\neg s$ is satisfiable.

d) If s and t are valid, then $s \wedge t$ is valid.

e) If s and t are satisfiable, then $s \wedge t$ is satisfiable.

f) If $s \wedge t$ is satisfiable, then both s and t are satisfiable.

g) If $s \vee t$ is satisfiable, then s is satisfiable or t is satisfiable.

h) If $s \vee t$ is valid, then s is valid or t is valid.

i) If $s \to t$ is satisfiable and s is satisfiable, then t is satisfiable.

j) If $\exists s$ is satisfiable, then sx is satisfiable if x not free in s.

k) If $\forall s$ is satisfiable, then st is satisfiable.

l) If $\forall s$ is valid, then st is valid.

m) If $st \neq su$ is satisfiable, then $t \neq u$ is satisfiable.

n) If $s \not\equiv t$ is satisfiable, then s and $\neg t$ are satisfiable or t and $\neg s$ are satisfiable.

o) If $s \neq t$ is satisfiable, then $sx \neq tx$ is satisfiable if x that is not free in $s \neq t$.

Exercise 4.5.10 Express validity in terms of logic equivalence and logic equivalence in terms of validity.

4.6 Propositional Logic

Propositional logic is a straigthforward logic that has numerous applications in Computer Science. We will obtain propositional logic as a subsystem of simple type theory. The formulas of propositional logic are obtained with the propositional constants $\bot$, $\top$, $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\equiv$ and variables of type o. With propositional formulas one can express the de Morgan Law $\neg(x \wedge y) = \neg x \vee \neg y$ for conjunction and the Contraposition Law $x \rightarrow y \equiv \neg y \rightarrow \neg x$ for implication. We give a rigorous definition of propositional formulas as formulas of simple type theory.

Definition. A **propositional variable** is a variable of type o. A **propositional formula** is a λ-free formula where every free name is either a propositional variable or a propositional constant. A **tautology** is a valid propositional formula.

The de Morgan Law and the Contraposition Law stated above are examples of equational tautologies. Peirce's Law $((x \rightarrow y) \rightarrow x) \rightarrow x$ is an example of a nonequational tautology.

Proposition 4.6.1 A formula is propositional if and only if it can be obtained with the grammar $s ::= x \mid \bot \mid \top \mid \neg s \mid s \wedge s \mid s \vee s \mid s \rightarrow s \mid s \equiv s$ where x ranges over propositional variables.

When we evaluate a propositional formula, we only need to know the values of the propositional variables occurring in the formula. Thus to determine the satisfiability and validity of a formula s, there is no need to consider full-blown logical interpretations. Rather it suffices to consider the functions $\mathcal{V}s \rightarrow \{0,1\}$. Since $\mathcal{V}s$ is finite, there are only finitely many such functions, and thus satisfiability and validity of propositional formulas are decidable. In fact, satisfiability of propositional formulas is the canonical NP-complete problem.

Proposition 4.6.2 Satisfiability of propositional formulas is NP-complete.

4.7 First-Order Predicate Logic

First-order predicate logic extends propositional logic by adding the quantifiers $\forall_\alpha$ and $\exists_\alpha$, the identities $=_\alpha$, and variables of type $\alpha_1 \dots \alpha_n\beta$ where $n \geq 0$. Recall that β ranges over base types and α ranges overs sorts (all base types but o). What makes first-order predicate logic first-order is the fact that quantifiers and identities are only available for sorts, and that functional variables can only have sorts as argument types. The first and the second axiom of the specification of chains are formulas of first-order predicate logic. The third axiom is not since it quantifies over a function type αo.

Definition. A **first-order variable** is a variable whose type has the form $\alpha_1 \dots \alpha_n\beta$ where $n \geq 0$, $\alpha_1, \dots, \alpha_n$ are sorts and β is a base type. A **first-order formula** is a formula s that can be obtained with the grammar

$$x ::= \text{first-order variable}$$
$$t ::= xt \dots t$$
$$s ::= t \mid \bot \mid \top \mid \neg s \mid s \wedge s \mid s \vee s \mid s \to s \mid s \equiv s \mid t =_\alpha t \mid \forall_\alpha x.s \mid \exists_\alpha x.s$$

where α ranges over sorts. Note that the pattern $xt \dots t$ subsumes x, and hence first-order variables are terms of category t. Also note that the grammar only generates typed terms. Every first-order formula is β-normal since abstractions can only appear as arguments of quantifications.

An example of a first-order formula is $(\neg\forall x.px(fx)) \equiv \exists x.\neg px(fx)$ where $x : \iota$, $f : \iota\iota$, and $p : \iota\iota o$ are first-order variables. The induction axiom of § 4.4 is not first-order since it contains the higher-order quantifier $\forall_{No}$. There are formulas that contain neither quantifiers nor identities and are not first-order. An example is $x \wedge px$.

A first-order formula is **pure** if it is relational and identity-free. Note that a first-order formula is relational if and only if it contains no variables of a type $\alpha_1 \dots \alpha_n\alpha$ where $n \geq 1$. A **first-order constant** is either a propositional constant or one of the constants $=_\alpha$, $\forall_\alpha$, $\exists_\alpha$ where α ranges over sorts. A **first-order name** is a first-order constant or a first-order variable. A **first-order term** is a λ-free term s such that the type of s is a sort and every name that occurs in s is a first-order variable.

Exercise 4.7.1 Give a β-normal formula that contains only first-order names but is not first-order.

Exercise 4.7.2 The following formulas are all valid. For each formula determine the types of the occurring variables. Moreover, decide for each of the formulas whether it is propositional and whether it is first-order. For first-order formulas also decide whether they are pure.

a) $f(f(fx)) \to fx$

b) $\forall_\alpha xy.\ x{=}y \to y{=}x$

c) $(\forall_\alpha x.\ fx \wedge gx) \equiv (\forall_\alpha x.\ fx) \wedge (\forall_\alpha y.\ gy)$

d) $(\forall_\alpha x.\ fx \wedge gx) \equiv (\forall_\alpha f) \wedge (\forall_\alpha g)$

e) $x{=}_\alpha y \equiv \forall f.\ fx \to fy$

f) $\exists x.\ x$

Exercise 4.7.3 Find a first-order formula that is logically equivalent to $f =_{\alpha\beta} g$.

Exercise 4.7.4 Find a propositional formula that is logically equivalent to the higher-order formula $\forall p.\ pxy \vee \neg pxy$.

4.8 Remarks

Simple type theory originated in 1940 with Church [20]. Ten years later Henkin [35] introduced nonstandard interpretations and showed that Church's proof system can prove exactly the valid formulas. A textbook covering simple type theory is Andrews [3]. Simple type theory is a prominent example of a **higher-order logic**. There are higher-order logics with more expressive type systems [8]. Simple type theory provides the logical base of the proof assistants Isabelle [52] and HOL [34].

The usual accounts of first-order logic and type theory (e.g., Andrews [3]) distinguish between logical constants, nonlogical constants, and variables. In contrast, we just have logical constants and variables. Our variables act both as nonlogical constants and as variables in the proper sense. Given a term, its free variables act as nonlogical constants and its bound variables act as variables in the proper sense. The unification of nonlogical constants and variables simplifies the semantic definitions.

5 Propositional Tableaux

In Chapter 3 we learned how to write (simply typed) terms. In Chapter 4 we learned how to give these terms meaning using logical interpretations. In this chapter we will begin learning to reason deductively. The process of reasoning we introduce will be mechanical and will apply only to syntax (terms). On the other hand, we must take care that the reasoning process respects semantics (logical interpretations).

Formulas (terms of type o) are interpreted either as 0 (false) or 1 (true) in any logical interpretation. A formula can either be valid (1 in all logical interpretations), satisfiable (1 in some logical interpretation), or unsatisfiable (1 in no logical interpretations). Suppose we want to determine a formula s is valid. We could introduce a proof system for validity. On the other hand, s is valid if and only if $\neg s$ is unsatisfiable. For this reason we can instead give a proof system for unsatisfiability. Accordingly we will define a tableau refutation calculus for finite sets of β-normal formulas A. In order for the calculus to correspond to the semantics, we must establish two facts:

- **Refutation Soundness:** If A is refutable, then A is unsatisfiable. (In particular, if $\neg s$ is refutable, then s is valid.)
- **Completeness:** If A is unsatisfiable, then A is refutable. (In particular, if s is valid, then $\neg s$ is refutable.)

Refutation soundness will be easy to justify since it will follow from soundness of each rule. Completeness will be more challenging.

In this chapter we will focus on tableaux for the propositional fragment of simple type theory. Propositional formulas are formulas where the only free names are propositional constants ($\bot$, $\top$, $\neg$, $\vee$, $\wedge$ and $\equiv$) and variables of type o. There is an easy naive algorithm for deciding satisfiability of a propositional formula: Consider each possible assignment of values $0, 1$ to each variable $p : o$ that occurs in the formula. The formula is satisfiable if and only if one of these assignments leads to evaluating the formula as 1. Satisfiability of propositional formulas (SAT) is a well-known example of an NP-complete problem. In fact, SAT was the first problem shown to be NP-complete [21]. Since satisfiability is decidable, validity and unsatisfiability are decidable as well.

Tableaux will consider a different algorithm for deciding unsatisfiability of finite sets of propositional formulas. Tableaux will scale to larger fragments of

simple type theory.

5.1 An Example: Peirce's Law

Let us consider an interesting formula called **Peirce's Law**:

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

Here we assume $p, q : o$ 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

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

for $a, b \in \mathbb{B}$. The following proposition is obvious.

Proposition 5.1.1 For any formulas s and t and any logical interpretation $\mathcal{I}$:

- If $\mathcal{I} \models s \to t$, then $\mathcal{I} \models t$ or $\mathcal{I} \models \neg s$.
- If $\mathcal{I} \models \neg(s \to t)$, then $\mathcal{I} \models s$ and $\mathcal{I} \models \neg t$.

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

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

Consequently,

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

and

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

Hence either $\mathcal{I} \models p$ or $\mathcal{I} \models \neg(p \to q)$. We cannot have $\mathcal{I} \models p$ since this contradicts $\mathcal{I} \models \neg p$. So we must have $\mathcal{I} \models \neg(p \to q)$. This means $\mathcal{I} \models \neg q$ and $\mathcal{I} \models p$, which again contradicts $\mathcal{I} \models \neg p$. Therefore, there can be no such logical interpretation $\mathcal{I}$. In other words, Peirce's Law is valid. We summarize this argument in the form of a tree in Figure 5.1. This tree is our first example 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.

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

Figure 5.1: Tableau for Peirce's Law

$$\mathcal{T}_{\to}\ \frac{s \to t}{t \mid \neg s} \qquad \mathcal{T}_{\neg\to}\ \frac{\neg(s \to t)}{s, \neg t} \qquad \mathcal{T}_{\neg}\ \frac{\neg s, s}{}$$

Figure 5.2: Tableau rules for implication

5.2 Tableau Rules: Implication

A **branch** is a finite set of β-normal formulas. We sometimes refer to a single formula s as a branch, by which we mean the singleton branch $\{s\}$. Note that Figure 5.1 contains two branches. The construction of the tableau in Figure 5.1 starting from the branch containing only the negation of Peirce's Law requires the three rule schemas shown in Figure 5.2. These are the first three rule schemas of what will be our tableau refutation calculus TaS for simple type theory. The remaining rules will be given throughout the chapter.

Each branch in Figure 5.1 is clearly unsatisfiable because each branch contains both p and $\neg p$.

In general, a **tableau rule** (or **rule**) is a tuple $\langle A, A_1, \ldots, A_n\rangle$ of branches with $n \geq 0$ such that $A \subsetneq A_i$ for each $i \in \{1, \ldots, n\}$. We can also write this tuple in the form

$$\dfrac{A}{\begin{array}{c|c|c} A_1 & \cdots & A_n \end{array}}$$

We refer to A as the **head** of this tableau rule and refer to each A_i as an **alternative** of the rule. If $n \geq 2$ we say the rule is **branching**.

We usually indicate a certain set of tableau rules by giving a rule schema. For example $\mathcal{T}_{\to}$ in Figure 5.2 is the set of rules $\langle A, A_1, A_2\rangle$ where for some $s, t : o$ we have $s \to t \in A$, $t \notin A$, $\neg s \notin A$, $A_1 = A \cup \{t\}$ and $A_2 = A \cup \{\neg s\}$. We say a rule **applies to** A if A is the head of the rule.

From an operational point of view, the tableau rule $\mathcal{T}_{\to}$ can be applied to A

whenever $s \to t$ is in A, $t \notin A$ and $\neg s \notin A$. Applying $\mathcal{T}_\to$ in such a situation yields two branches $A \cup \{t\}$ and $A \cup \{\neg s\}$.

While schemas like $\mathcal{T}_\to$ and $\mathcal{T}_{\neg\to}$ are technically sets of rules, we will often refer to them simply as rules.

A tableau calculus is given by a set of tableau rules. Relative to a tableau calculus, we say a branch A is **closed** if $\langle A \rangle$ is a rule in the calculus. $\mathcal{T}_\neg$ is an example of such a rule and explains technically why we can say both branches of the tableau in Figure 5.1 are closed. If a branch is not closed, we say it is **open**. A branch is **maximal** if it is either closed or no rule applies to it.

Given a tableau calculus $\mathcal{T}$ we can define the set of $\mathcal{T}$-refutable branches inductively.

- If $\langle A, A_1, \ldots, A_n \rangle$ is a rule in $\mathcal{T}$ and A_i is $\mathcal{T}$-refutable for all $i \in \{1, \ldots, n\}$, then A is $\mathcal{T}$-refutable.

(The base case of this inductive definition is when $n = 0$.) We will simply say **refutable** when the tableau calculus $\mathcal{T}$ is clear in context.

We will often consider fragments of the full language. A **fragment** $\mathcal{F}$ is simply a set of β-normal terms. An $\mathcal{F}$-branch is a branch that only contains formulas from $\mathcal{F}$. We refer to the set of all β-normal terms as the **full fragment**.

We can now define soundness and completeness of a tableau calculus $\mathcal{T}$ relative to a fragment $\mathcal{F}$.

- **Refutation Soundness:** $\mathcal{T}$ is **refutation sound** with respect to $\mathcal{F}$ if every refutable $\mathcal{F}$-branch is unsatisfiable.
- **Completeness:** $\mathcal{T}$ is **complete** with respect to $\mathcal{F}$ if every unsatisfiable $\mathcal{F}$-branch is refutable.

We can also define a related notion.

- **Verification Soundness:** $\mathcal{T}$ is **verification sound** with respect to $\mathcal{F}$ if every maximal open $\mathcal{F}$-branch is satisfiable.

A tableau rule is **sound** if either A is unsatisfiable or A_i is satisfiable for some $i \in \{1, \ldots, n\}$. (When $n = 0$, this simply means that A is unsatisfiable.) We will only consider sound tableau rules. Most tableau rules will satisfy a stronger property: If $\mathcal{I} \vDash A$, then $\mathcal{I} \vDash A_i$ for some $i \in \{1, \ldots, n\}$.

Proposition 5.2.1 Let R be a tableau rule

$$\frac{A}{A_1 \mid \cdots \mid A_n}$$

Suppose for any logical interpretation $\mathcal{I}$, if $\mathcal{I} \vDash A$, then $\mathcal{I} \vDash A_i$ for some $i \in \{1, \ldots, n\}$. Then the rule R is sound.

Exercise 5.2.2 Prove Proposition 5.2.1.

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

Figure 5.3: Tableau with a maximal open branch

Exercise 5.2.3 a) Is the tableau rule $\langle \emptyset \rangle$ sound?

b) Let A be a branch. Let $s : o$ be such that $s \notin A$ and $\neg s \notin A$. Is the tableau rule $\langle A, A \cup \{s\}, A \cup \{\neg s\} \rangle$ sound?

Exercise 5.2.4 Let $\langle A, A_1, \ldots, A_n \rangle$ be any tableau rule. Argue that if A_i is satisfiable for some $i \in \{1, \ldots, n\}$ then A is satisfiable. (Hint: It is extremely easy.)

A set of rules is **sound** if every rule in the set is sound. Soundess of the rule sets $\mathcal{T}_{\to}$ and $\mathcal{T}_{\neg\to}$ follow from Propositions 5.1.1 and 5.2.1.

Exercise 5.2.5 Prove soundness of the rule set $\mathcal{T}_{\neg}$ using Proposition 5.2.1.

Proposition 5.2.6 Let $\mathcal{T}$ be a tableau system. If every rule in $\mathcal{T}$ is sound, then $\mathcal{T}$ is refutation sound with respect to the full fragment.

Proof Suppose sound tableau rules have been used to extend a branch A to closed branches $A_1, \ldots, A_n$. If A were satisfiable, then there would be some $i \in \{1, \ldots, n\}$ such that A_i is satisfiable, which is impossible since the rules defining closedness are also sound. ∎

Since we will only include sound rules in our tableau calculi, all of our tableau calculi will be refutation sound with respect to the full fragment. Hence our tableau calculi will be refutation sound with respect to any fragment. Consequently, it was not necessary to define refutation soundness with respect to fragments. On the other hand, completeness and verification soundness will depend strongly on the fragment in question.

We will primarily be interested in using tableaux to show unsatisfiability of a branch A. In some special cases we will be able to use tableau to show satisfiability of a branch A. Consider the formula $(q \to p) \to p$. To show that this formula is not valid, we must find a logical interpretation $\mathcal{I}$ such that $\mathcal{I} \vDash \neg((q \to p) \to p)$. Consider the tableau in Figure 5.3. The right branch is a maximal open branch. Note that any interpretation $\mathcal{I}$ such that $\mathcal{I}p = 0$ and $\mathcal{I}q = 0$ will satisfy all the formulas on the right branch, including $\neg((q \to p) \to p)$.

$$\mathcal{T}_{\bot}\ \frac{\bot}{}\qquad \mathcal{T}_{\neg\top}\ \frac{\neg\top}{}\qquad \mathcal{T}_{\neg}\ \frac{\neg s, s}{}\qquad \mathcal{T}_{\neg\neg}\ \frac{\neg\neg s}{s}\qquad \mathcal{T}_{\wedge}\ \frac{s \wedge t}{s, t}\qquad \mathcal{T}_{\neg\wedge}\ \frac{\neg(s \wedge t)}{\neg s \mid \neg t}$$

$$\mathcal{T}_{\vee}\ \frac{s \vee t}{s \mid t}\qquad \mathcal{T}_{\neg\vee}\ \frac{\neg(s \vee t)}{\neg s, \neg t}\qquad \mathcal{T}_{\rightarrow}\ \frac{s \rightarrow t}{t \mid \neg s}\qquad \mathcal{T}_{\neg\rightarrow}\ \frac{\neg(s \rightarrow t)}{s, \neg t}$$

$$\mathcal{T}_{\mathbf{BQ}}\ \frac{s \equiv t}{s, t \mid \neg s, \neg t}\qquad \mathcal{T}_{\mathbf{BE}}\ \frac{\neg(s \equiv t)}{s, \neg t \mid t, \neg s}$$

Figure 5.4: Propositional tableau rules (TaP)

5.3 Tableau Rules: Propositional Connectives

We have given enough rules to reason about formulas that only involve implication and variables of type o. In this section we give rules which suffice to reason about formulas using the propositional constants $\bot$, $\top$, $\neg$, $\vee$, $\wedge$ and $\equiv$. The rules for $\bot$ are given by

$$\mathcal{T}_{\bot}\ \frac{\bot}{}$$

That is, any branch containing $\bot$ is closed.

We have already given a rule schema $\mathcal{T}_{\neg}$ involving negation. This is not yet enough to completely reason about negation. For example, we cannot yet refute the unsatisfiable formula $\neg\neg\bot$. Just as $\mathcal{T}_{\neg\rightarrow}$ applies to negated implications, we will need rules for reasoning about negations in front of each logical constant. In particular, we need a rule for a negation in front of a negation.

$$\mathcal{T}_{\neg\neg}\ \frac{\neg\neg s}{s}$$

For the remaining propositional connectives, there is one rule for the connective and one rule for a negation in front of the connective. These rules (along with all previous rules) are summarized in Figure 5.4. These rules define our tableau system TaP for propositional logic. We will soon show this tableau system terminates, is verification sound and is complete. This implies TaP gives a decision procedure for propositional logic.

Exercise 5.3.1 For each of the rules in Figure 5.4 give true statements analogous to the statements in Proposition 5.1.1 so that soundness of each rule schema

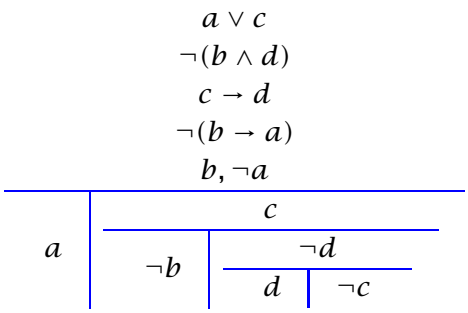


Figure 5.5: Tableau refutation

follows from Proposition 5.2.1. For example, the statement corresponding to the $\mathcal{T}_\wedge$ rule is:

- If $\mathfrak{I} \models s \wedge t$, then $\mathfrak{I} \models s$ and $\mathfrak{I} \models t$.

Consider the following example. Fix distinct variables $a, b, c, d : o$. We informally associate these variables with four sentences.

- a means "Alice is happy."
- b means "Bob is happy."
- c means "Carol is happy."
- d means "Dave is happy."

Suppose either Alice or Carol is happy: $a \vee c$. Suppose Bob and Dave cannot both be happy: $\neg(b \wedge d)$. Finally suppose Dave is happy whenever Carol is happy: $c \to d$. The tableau refutation in Figure 5.5 demonstrates that Alice is happy whenever Bob is happy: $b \to a$. That is, the branch with $a \vee c$, $\neg(b \wedge d)$, $c \to d$ and $\neg(b \to a)$ is unsatisfiable.

Exercise 5.3.2 Let $a, b, c : o$ be variables. Give tableau refutations of the following branches.

a) $a \not\equiv a$

b) $\{a \equiv b, b \not\equiv a\}$

c) $\{a \equiv b, b \equiv c, a \not\equiv c\}$

d) $(a \wedge b \to c) \not\equiv (a \to b \to c)$

e) $(a \to b) \not\equiv (\neg b \to \neg a)$

f) $\neg(a \wedge b) \not\equiv \neg a \vee \neg b$

g) $\neg(a \vee b) \not\equiv \neg a \wedge \neg b$

h) $\neg((a \to b) \vee (b \to a))$

5.4 Termination

Given a tableau system $\mathcal{T}$ and a fragment $\mathcal{F}$, a **$\mathcal{T}$-$\mathcal{F}$-chain** is a finite or infinite sequence of $\mathcal{F}$-branches

$$A_0 \subsetneq A_1 \subsetneq A_2 \subsetneq \cdots$$

where for each i there is some rule in $\mathcal{T}$ with A_i as its head and A_{i+1} as one of its alternatives. We will simply say **chain** when $\mathcal{T}$ and $\mathcal{F}$ are clear in context. The notion of a chain allows us to define termination.

- **Termination:** $\mathcal{T}$ is **terminating** with respect to $\mathcal{F}$ if there is no infinite chain.

  Completeness follows from termination and verification soundness.

Lemma 5.4.1 Suppose $\mathcal{T}$ is a tableau system that is terminating and verification sound. If a branch A is unsatisfiable and not refutable, then there is some branch A' where A' is not refutable and some rule in $\mathcal{T}$ with A as head and A' as an alternative.

Proof A is not closed since it is not refutable. A is not a maximal open branch since it is unsatisfiable and $\mathcal{T}$ is verification sound. Hence some rule in $\mathcal{T}$ has the form $\langle A, A_1, \ldots, A_n \rangle$ with $n \geq 1$. If A_i were refutable for every $i \in \{1, \ldots, n\}$, then A would be refutable. Hence there is some $i \in \{1, \ldots, n\}$ such that A_i is not refutable. Choose A' to be this A_i. ∎

Proposition 5.4.2 Suppose $\mathcal{T}$ is a tableau system that is terminating and verification sound. Then $\mathcal{T}$ is complete.

Proof Let A be an unsatisfiable branch. Assume A is not refutable. We will inductively construct an infinite chain

$$A_0 \subsetneq A_1 \subsetneq A_2 \subsetneq \cdots$$

contradicting termination. We will further ensure that each A_n is unsatisfiable and not refutable. Let A_0 be A. Assume we have

$$A_0 \subsetneq A_1 \subsetneq A_2 \subsetneq \cdots \subsetneq A_n$$

Since A_n is unsatisfiable and not refutable, we can apply Lemma 5.4.1 to obtain A_{n+1} which is unsatisfiable (since $A_n \subsetneq A_{n+1}$) and not refutable and such that

$$A_0 \subsetneq A_1 \subsetneq A_2 \subsetneq \cdots \subsetneq A_n \subsetneq A_{n+1}$$

is a chain. We inductively obtain an infinite chain, contradicting termination. ∎

We will now show TaP terminates (on the full fragment). For the termination proof it helps to define a finite superset of a branch A that none of the TaP rules can escape. For this reason, we define the **subterm universe** $SU(A)$ of a branch A to be the set

$$\{s | s : o \text{ is a subterm of some } t \in A\} \cup \{\neg s | s : o \text{ is a subterm of some } t \in A\}$$

Clearly $A \subseteq SU(A)$.

Exercise 5.4.3 Let A be a branch. Argue that each rule $\langle A', A_1, \ldots, A_n \rangle$ in TaP is such that if $A' \subseteq SU(A)$, then $A_i \subseteq SU(A)$ for all $i \in \{1, \ldots, n\}$.

We now show TaP terminates.

Proposition 5.4.4 TaP terminates.

Proof Suppose there were an infinite chain

$$A_0 \subsetneq A_1 \subsetneq A_2 \subsetneq \cdots$$

Clearly $A_0 \subseteq SU(A_0)$. By Exercise 5.4.3 we know $A_i \subseteq SU(A_0)$. Hence we have

$$A_0 \subsetneq A_1 \subsetneq A_2 \subsetneq \cdots \subseteq SU(A_0)$$

This is impossible since $SU(A_0)$ is finite. ∎

5.5 Propositional Examples

In this section we give the construction of several tableau refutations in full detail. The reader should understand how the final tableau in each example is constructed and why every branch in the final tableau is closed.

Example 5.5.1 We refute the following branch:

$$\begin{array}{c} a \to c \wedge d \\ \neg c \\ a \end{array}$$

The rule $\mathcal{T}_\to$ applied to $a \to c \wedge d$ gives

$$\begin{array}{c|c} a \to c \wedge d & \\ \neg c & \\ a & \\ \hline c \wedge d & \neg a \end{array}$$

We apply $\mathcal{T}_\wedge$ to $c \wedge d$ and obtain

$$\begin{array}{c|c} a \to c \wedge d & \\ \neg c & \\ a & \\ \hline c \wedge d & \\ c & \neg a \\ d & \end{array}$$

Example 5.5.2 We refute the following branch:

$$\begin{array}{c} a \wedge b \to c \wedge d \\ \neg(\neg c \to \neg a \vee \neg b) \end{array}$$

The rule $\mathcal{T}_{\neg\to}$ applied to $\neg(\neg c \to \neg a \vee \neg b)$ gives

$$\begin{array}{c} a \wedge b \to c \wedge d \\ \neg(\neg c \to \neg a \vee \neg b) \\ \neg c \\ \neg(\neg a \vee \neg b) \end{array}$$

We apply $\mathcal{T}_{\neg\vee}$ to $\neg(\neg a \vee \neg b)$ and obtain

$$\begin{array}{c} a \wedge b \to c \wedge d \\ \neg(\neg c \to \neg a \vee \neg b) \\ \neg c \\ \neg(\neg a \vee \neg b) \\ \neg\neg a \\ \neg\neg b \end{array}$$

Applying $\mathcal{T}_\to$ to $a \wedge b \to c \wedge d$ we obtain

$$\begin{array}{c|c} a \wedge b \to c \wedge d & \\ \neg(\neg c \to \neg a \vee \neg b) & \\ \neg c & \\ \neg(\neg a \vee \neg b) & \\ \neg\neg a & \\ \neg\neg b & \\ \hline c \wedge d & \neg(a \wedge b) \end{array}$$

The rule $\mathcal{T}_\wedge$ applied to $c \wedge d$ gives

$$\begin{array}{c}
a \wedge b \to c \wedge d \\
\neg(\neg c \to \neg a \vee \neg b) \\
\neg c \\
\neg(\neg a \vee \neg b) \\
\neg\neg a \\
\neg\neg b \\
\hline
\begin{array}{c|c}
c \wedge d & \\
c & \neg(a \wedge b) \\
d &
\end{array}
\end{array}$$

We apply $\mathcal{T}_{\neg\wedge}$ to $\neg(a \wedge b)$ and obtain

$$\begin{array}{c}
a \wedge b \to c \wedge d \\
\neg(\neg c \to \neg a \vee \neg b) \\
\neg c \\
\neg(\neg a \vee \neg b) \\
\neg\neg a \\
\neg\neg b \\
\hline
\begin{array}{c|c}
c \wedge d & \neg(a \wedge b) \\
\cline{2-2}
c & \begin{array}{c|c} \neg a & \neg b \end{array} \\
d &
\end{array}
\end{array}$$

Example 5.5.3 We refute the following branch:

$$a \wedge (a \to b) \not\equiv a \wedge b$$

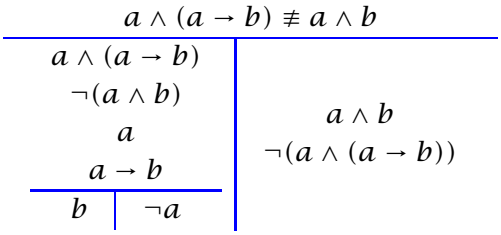
The rule $\mathcal{T}_{\text{BE}}$ applied to $a \wedge (a \to b) \not\equiv a \wedge b$ gives

$$\begin{array}{c}
a \wedge (a \to b) \not\equiv a \wedge b \\
\hline
\begin{array}{c|c}
a \wedge (a \to b) & a \wedge b \\
\neg(a \wedge b) & \neg(a \wedge (a \to b))
\end{array}
\end{array}$$

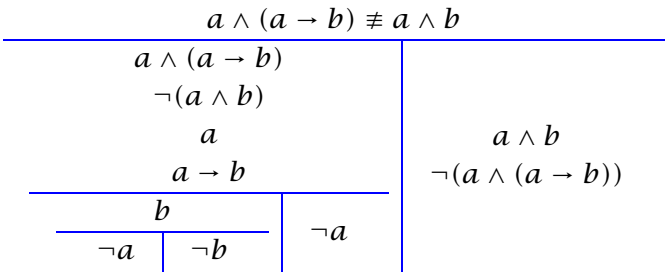
We apply $\mathcal{T}_\wedge$ to $a \wedge (a \to b)$ and obtain

$$\begin{array}{c}
a \wedge (a \to b) \not\equiv a \wedge b \\
\hline
\begin{array}{c|c}
a \wedge (a \to b) & \\
\neg(a \wedge b) & a \wedge b \\
a & \neg(a \wedge (a \to b)) \\
a \to b &
\end{array}
\end{array}$$

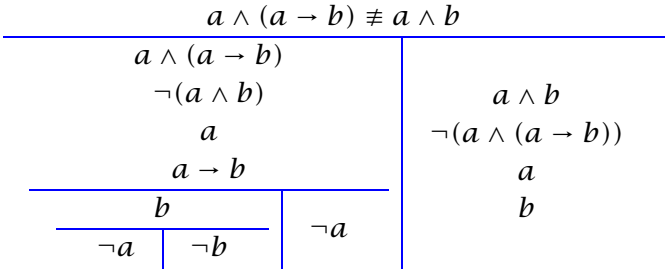
Applying $\mathcal{T}_{\rightarrow}$ to $a \rightarrow b$ we obtain



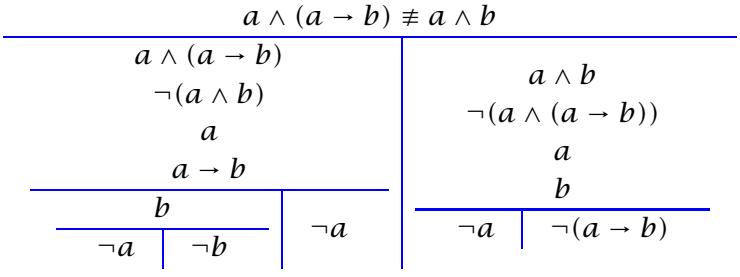
The rule $\mathcal{T}_{\neg\wedge}$ applied to $\neg(a \wedge b)$ gives



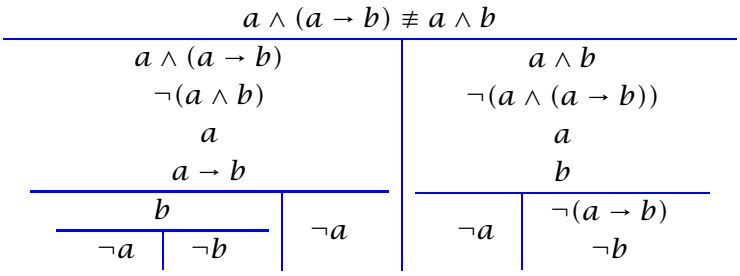
We apply $\mathcal{T}_{\wedge}$ to $a \wedge b$ and obtain



Applying $\mathcal{T}_{\neg\wedge}$ to $\neg(a \wedge (a \rightarrow b))$ we obtain



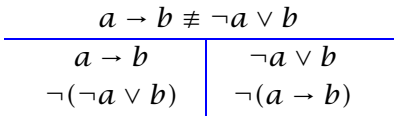
The rule $\mathcal{T}_{\neg\to}$ applied to $\neg(a \to b)$ gives



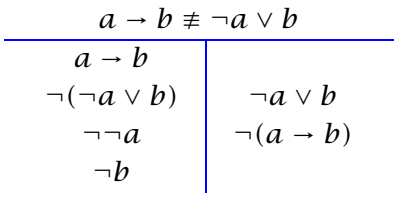
Example 5.5.4 We refute the following branch:

$$a \to b \not\equiv \neg a \vee b$$

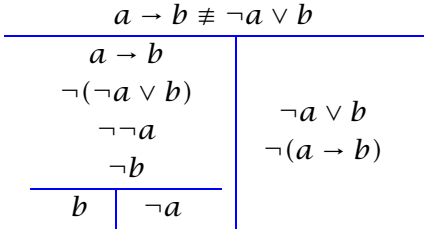
The rule $\mathcal{T}_{\text{BE}}$ applied to $a \to b \not\equiv \neg a \vee b$ gives



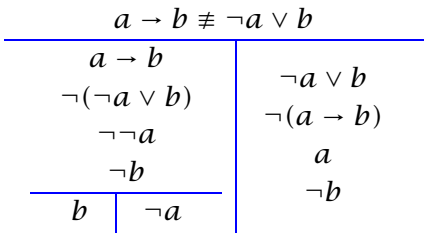
We apply $\mathcal{T}_{\neg\vee}$ to $\neg(\neg a \vee b)$ and obtain



Applying $\mathcal{T}_{\to}$ to $a \to b$ we obtain



The rule $\mathcal{T}_{\neg\to}$ applied to $\neg(a \to b)$ gives



We apply $\mathcal{T}_\vee$ to $\neg a \vee b$ and obtain

$$\dfrac{a \to b \not\equiv \neg a \vee b}{\begin{array}{c|c} a \to b & \neg a \vee b \\ \neg(\neg a \vee b) & \neg(a \to b) \\ \neg\neg a & a \\ \neg b & \neg b \\ \hline b \mid \neg a & \neg a \mid b \end{array}}$$

Example 5.5.5 We refute the following branch:

$$\begin{array}{l} a \equiv \neg b \\ a \equiv b \end{array}$$

The rule $\mathcal{T}_{\text{BQ}}$ applied to $a \equiv b$ gives

$$\dfrac{\begin{array}{l} a \equiv \neg b \\ a \equiv b \end{array}}{\begin{array}{l|l} a & \neg a \\ b & \neg b \end{array}}$$

We apply $\mathcal{T}_{\text{BQ}}$ to $a \equiv \neg b$ and obtain

$$\dfrac{\begin{array}{l} a \equiv \neg b \\ a \equiv b \end{array}}{\begin{array}{ll|l} a & & \\ b & & \neg a \\ \hline \neg b & \neg a & \neg b \\ & \neg\neg b & \end{array}}$$

Applying $\mathcal{T}_{\text{BQ}}$ to $a \equiv \neg b$ we obtain

$$\dfrac{\begin{array}{l} a \equiv \neg b \\ a \equiv b \end{array}}{\begin{array}{ll|ll} a & & \neg a & \\ b & & \neg b & \\ \hline \neg b & \neg a & a & \neg\neg b \\ & \neg\neg b & & \end{array}}$$

5.6 Completeness of the Implication Fragment

Let $\mathtt{TaP}_{\to}$ be the tableau system given by the rules $\mathcal{T}_{\to}$, $\mathcal{T}_{\neg\to}$, $\mathcal{T}_{\neg\neg}$ and $\mathcal{T}_{\neg}$. We define the set **PropImp** of **propositional implication formulas** by the grammar

$s ::= p \mid \neg s \mid s \to s$

$p : o$ is a variable

In this section we will prove the following completeness result.

Theorem 5.6.1 (Restricted Propositional Completeness)

1. $\mathtt{TaP}_{\to}$ is verification sound with respect to **PropImp**.
2. $\mathtt{TaP}_{\to}$ is complete with respect to **PropImp**.

By Proposition 5.4.4 we know $\mathtt{TaP}_{\to}$ terminates. Hence by Proposition 5.4.2 part (2) of Theorem 5.6.1 follows from part (1). We dedicate the rest of this section to proving part (1).

To prove the verification soundness we must construct an interpretation. Recall our attempt to refute $\neg((q \to p) \to p$ ending with the tableau in Figure 5.3. The right branch

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

is open and maximal. Is A satisfiable? In other words, is there a logical interpretation $\mathcal{I}$ such that $\mathcal{I} \models A$? Yes, as mentioned above, any logical interpretation $\mathcal{I}$ such that $\mathcal{I}p = 0$ and $\mathcal{I}q = 0$ will satisfy A. We can use a similar idea to prove the following general result.

For any set A of formulas and logical interpretation $\mathcal{I}$ we say $\mathcal{I}$ **respects** A if

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

for all variables $p : o$. The following proposition is obvious.

Proposition 5.6.2 For any set A there is a logical interpretation $\mathcal{I}$ which respects A.

Suppose $\mathcal{I}$ respects A. Clearly $\mathcal{I} \models p$ for names $p \in A$. We cannot in general conclude $\mathcal{I} \models A$ (i.e., $\mathcal{I} \models s$ for all $s \in A$). However, we will soon define a property of sets A of formulas which will guarantee $\mathcal{I} \models A$ if A contains only **PropImp**-formulas.

We define the following **evidence conditions** of a set A of formulas:

$\mathcal{E}_{\neg}$ If $\neg s \in A$, then $s \notin A$.

$\mathcal{E}_{\neg\neg}$ If $\neg\neg s \in A$, then $s \in A$.

$\mathcal{E}_{\to}$ If $s \to t \in A$, then $\neg s \in A$ or $t \in A$.

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

We say a set A of formulas is **PropImp-evident** if it satisfies these four evidence conditions. (For the rest of this section we will simply say **evident** for **PropImp**-evident.)

We will prove two lemmas.

Lemma 5.6.3 Let A be a branch. If A is open and maximal with respect to $\mathtt{TaP}_{\to}$, then A is evident.

Lemma 5.6.4 Let A be an evident set of formulas and $\mathcal{I}$ be a logical interpretation which respects A. For all $s \in$ **PropImp**, we have the following:

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

The first part of Theorem 5.6.1 follows from these two lemmas. Suppose A is open and maximal. Let $\mathcal{I}$ be a logical interpretation which respects A. By Lemma 5.6.3 it is evident. By Lemma 5.6.4 $\mathcal{I} \models A$. Hence A is satisfiable.

We now prove the two lemmas. The proof of the first lemma is very easy.

Proof (Proof of Lemma 5.6.3) We verify each of the four properties.

$\mathcal{E}_{\neg}$ Assume $\mathcal{E}_{\neg}$ does not hold. Then $\{s, \neg s\} \subseteq A$ for some formula s. In this case $\mathcal{T}_{\neg}$ applies and A is closed, contradicting our assumption.

$\mathcal{E}_{\neg\neg}$ Assume $\mathcal{E}_{\neg\neg}$ does not hold. We must have $\neg\neg s \in A$ and $s \notin A$. In this case $\mathcal{T}_{\neg\neg}$ applies, contradicting our assumption that A is maximal.

$\mathcal{E}_{\to}$ Assume $\mathcal{E}_{\to}$ does not hold. We must have $s \to t \in A$, $\neg s \notin A$ and $t \notin A$. In this case $\mathcal{T}_{\to}$ applies, contradicting our assumption that A is maximal.

$\mathcal{E}_{\neg\to}$ Assume $\mathcal{E}_{\neg\to}$ does not hold. We must have $\neg(s \to t) \in A$ and $\{s, \neg t\} \notin A$. In this case $\mathcal{T}_{\neg\to}$ applies, contradicting our assumption that A is maximal. ∎

The proof of the second lemma is by mutual induction on $s \in$ **PropImp**. That is, we will prove both parts of the lemma at the same time, and we are allowed to use the induction hypothesis from either part if the formula is smaller.

Proof (Proof of Lemma 5.6.4) We need to prove for every $s \in$ **PropImp** the two properties hold:

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

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

There are three kinds of formulas in **PropImp**: 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 $p \in$ **PropImp**.

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

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

Next we consider $\neg s \in$ **PropImp**. Our inductive hypothesis is that both (1_s) and (2_s) hold.

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

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

Finally we consider $s \to t \in$ **PropImp**. 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 $\hat{\mathcal{I}}(s \to t) = 1$. By $\mathcal{E}_\to$ either $\neg s \in A$ or $t \in A$. If $\neg s \in A$ then $\hat{\mathcal{I}}s = 0$ by (2_s) and so $\hat{\mathcal{I}}(s \to t) = 1$. If $t \in A$ then $\hat{\mathcal{I}}t = 1$ by (1_t) and so $\hat{\mathcal{I}}(s \to t) = 1$. In either case we have the desired result.

($2_{s\to t}$) Assume $\neg(s \to t) \in A$. We must prove $\hat{\mathcal{I}}(s \to t) = 0$. By $\mathcal{E}_{\neg\to}$ both $s \in A$ and $\neg t \in A$. By (1_s) and (2_t) we have $\hat{\mathcal{I}}s = 1$ and $\hat{\mathcal{I}}t = 0$. Hence $\hat{\mathcal{I}}(s \to t) = 0$. ∎

5.7 Completeness of the Propositional Fragment

We define the set **Prop** of **propositional formulas** by the grammar

$$s ::= p \mid \bot \mid \top \mid \neg s \mid s \to s \mid s \wedge s \mid s \vee s \mid s \equiv s$$

$p : o$ is a variable

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

Theorem 5.7.1 (Propositional Completeness)

1. TaP is verification sound with respect to **Prop**.
2. TaP is complete with respect to **Prop**.

We will use the an interpretation $\mathcal{I}$ which respects A. We will also use two lemmas. Since there are more tableau rules we need to expand our notion of evident to include more evidence conditions.

We will say A is **Prop-evident** if the following 12 **evidence conditions** hold:

$\mathcal{E}_{\neg}$ For all formulas s, $s \notin A$ or $\neg s \notin A$.

$\mathcal{E}_{\bot}$ $\bot \notin A$.

$\mathcal{E}_{\neg\top}$ $\neg\top \notin A$.

$\mathcal{E}_{\neg\neg}$ If $\neg\neg s \in A$, then $s \in A$.

$\mathcal{E}_{\to}$ If $s \to t \in A$, then $\neg s \in A$ or $t \in A$.

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

$\mathcal{E}_{\vee}$ If $s \vee t \in A$, then $s \in A$ or $t \in A$.

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

$\mathcal{E}_{\wedge}$ If $s \wedge t \in A$, then $s \in A$ and $t \in A$.

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

$\mathcal{E}_{\mathbf{BQ}}$ If $s \equiv t \in A$, then $\{s, t\} \subseteq A$ or $\{\neg s, \neg t\} \subseteq A$.

$\mathcal{E}_{\mathbf{BE}}$ 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 evidence conditions from the previous section. For the remainder of this section we will simply say **evident** for **Prop**-evident.

Lemma 5.7.2 Let A be a branch. If A is open and maximal with respect to TaP, then A is evident.

Lemma 5.7.3 Let A be an evident set of formulas and $\mathcal{I}$ be a logical interpretation which respects A. For all propositional formulas s, we have the following:

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

We can prove Lemmas 5.7.2 and 5.7.3 using the same techniques as Lemmas 5.6.3 and 5.6.4. We sketch the proofs and leave the reader to check the details.

Proof (Proof of Lemma 5.7.2) $\mathcal{E}_{\neg}$,$\mathcal{E}_{\bot}$,$\mathcal{E}_{\neg\top}$ since A is open.

$\mathcal{E}_{\neg\neg}$ since $\mathcal{T}_{\neg\neg}$ does not apply.

$\mathcal{E}_{\to}$ since $\mathcal{T}_{\to}$ does not apply.

$\mathcal{E}_{\neg\to}$ since $\mathcal{T}_{\neg\to}$ does not apply.

$\mathcal{E}_{\vee}$ since $\mathcal{T}_{\vee}$ does not apply.

$\mathcal{E}_{\neg\vee}$ since $\mathcal{T}_{\neg\vee}$ does not apply.

$\mathcal{E}_{\wedge}$ since $\mathcal{T}_{\wedge}$ does not apply.

$\mathcal{E}_{\neg\wedge}$ since $\mathcal{T}_{\neg\wedge}$ does not apply.

$\mathcal{E}_{\mathrm{BQ}}$ since $\mathcal{T}_{\mathrm{BQ}}$ does not apply.

$\mathcal{E}_{\mathrm{BE}}$ since $\mathcal{T}_{\mathrm{BE}}$ does not apply. ∎

Proof (Proof of Lemma 5.7.3) The proof is by induction on the propositional formula s:

- For variables p use $\mathcal{E}_{\neg}$.
- For $\top$ use $\mathcal{E}_{\neg\top}$.
- For $\bot$ use $\mathcal{E}_{\bot}$.
- For $\neg s$ use $\mathcal{E}_{\neg\neg}$.
- For $s \to t$ use $\mathcal{E}_{\to}$ and $\mathcal{E}_{\neg\to}$.
- For $s \vee t$ use $\mathcal{E}_{\vee}$ and $\mathcal{E}_{\neg\vee}$.
- For $s \wedge t$ use $\mathcal{E}_{\wedge}$ and $\mathcal{E}_{\neg\wedge}$.
- For $s \equiv t$ use $\mathcal{E}_{\mathrm{BQ}}$ and $\mathcal{E}_{\mathrm{BQ}}$. ∎

Finally, we can prove propositional completeness.

Proof (Proof of Theorem 5.7.1) The first part of Theorem 5.7.1 follows from Lemmas 5.7.2 and 5.7.3. Suppose A is open and maximal. Let $\mathcal{I}$ be a logical interpretation which respects A. By Lemma 5.7.2 it is evident. By Lemma 5.7.3 $\mathcal{I} \models A$. Hence A is satisfiable.

As with Theorem 5.6.1, the second part of Theorem 5.7.1 follows from the first part and termination via Propositions 5.4.2 and 5.4.4. ∎

Implicit in the proof above is the fact that any search for a refutation 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.

5.8 Remarks

6 Tableaux with Quantifiers

In Chapter 5 we gave rules for the propositional logical constants and proved the corresponding calculus is complete for propositional logic. In this chapter we will give tableau rules for quantifiers $\forall$ and $\exists$. The given calculus will be complete for pure first-order formulas. In the next chapter we will give the final tableau rules for handling equality. Note, however, that we can already reason to some degree with equality once we have quantifiers by using Leibniz equality.

6.1 Tableau Rules for Quantifiers

We will give a tableau system TaQ for quantifiers by including all of the tableau rules from TaP (see Figure 5.4) and adding rules for quantifiers. In particular, a branch A will be closed (with respect to TaQ) if and only if $\bot \in A$, $\neg\top \in A$, or $\{s, \neg s\} \subseteq A$ for some s.

Let $p : \tau o$, $f : \sigma\tau$, $x : \sigma$ and $y : \tau$ be distinct variables. Consider the branch A_0 with two formulas: $\exists x.p(fx)$ and $\forall y.\neg py$. Clearly A_0 is unsatisfiable: Suppose $\mathcal{I}$ were a model of A_0. Let P be $\mathcal{I}p$ and F be $\mathcal{I}f$. Since $\mathcal{I} \models \exists x.p(fx)$, there must be some $X \in \mathcal{I}\sigma$ such that $P(FX) = 1$. On the other hand, since $\mathcal{I} \models \forall y.\neg py$ we must have $PY = 0$ for all $Y \in \mathcal{I}\sigma$. Choosing Y to be FX we have a contradition.

How can we model this semantic argument using tableau rules? Suppose we add $p(fx)$ to the branch A_0 to obtain

$$A_1 = A_0 \cup \{p(fx)\}$$

Next suppose we add $\neg p(fx)$ to A_1 to obtain

$$A_2 = A_1 \cup \{\neg p(fx)\}$$

Since $p(fx)$ and $\neg p(fx)$ are in A_2, A_2 will be closed. Our quantifier rules will be such that $\langle A_0, A_1\rangle$ and $\langle A_1, A_2\rangle$ are rules of TaQ. Hence Figure 6.1 will be a TaQ-refutation of A_0.

Let us first consider the rule $\langle A_1, A_2\rangle$ and the following schema.

$$\frac{\forall_\sigma s}{st}\ t : \sigma$$

$$\exists x.p(fx)$$
$$\forall y.\neg py$$
$$p(fx)$$
$$\neg p(fx)$$

Figure 6.1: Tableau refutation with quantifiers

This schema indicates a set of rules including $\langle A, A' \rangle$ whenever $\forall_\sigma s \in A$, $t : \sigma$, $st \notin A$ and A' is $A \cup \{st\}$. Is the rule $\langle A_1, A_2 \rangle$ a member of this set of rules? Recall that $\forall y.\neg py$ is notation for $\forall_\tau(\lambda y.\neg py)$. Hence the rule schema would allow us to add $(\lambda y.\neg py)(fx)$ to the branch A_1. This would not yield A_2, as we desired. We instead would like to obtain the β-normal form $\neg p(fx)$.

Let $[s]$ denote $\mathcal{B}\imath s$. Recall from Chapter 3 that $\mathcal{B}\imath s$ yields a β-normal form for s which is unique up to $\sim_\alpha$. Now we can give the appropriate rule schema $\mathcal{T}_\forall$:

$$\mathcal{T}_\forall \quad \frac{\forall_\sigma s}{[st]} \quad t : \sigma$$

The reader should check that $\langle A_1, A_2 \rangle$ in our example is in the set of rules given by this schema. (Hint: The particular type σ will be τ.) Soundness of each rule in $\mathcal{T}_\forall$ follows from Proposition 5.2.1 and the following fact:

- For any logical interpretation, if $\mathcal{I} \vDash \forall_\sigma s$, then $\mathcal{I} \vDash [st]$.

Next we turn to the rule $\langle A_0, A_1 \rangle$. Here we used the formula $\exists x.p(fx)$ to justify adding $p(fx)$ to the branch. The idea is that if there is some element satisfying a property, then we can use the variable x to name such an element. The correct rule schema $\mathcal{T}_\exists$ justifying this step is

$$\mathcal{T}_\exists \quad \frac{\exists_\sigma s}{[sx]} \quad x : \sigma \text{ fresh}$$

We say a variable x is **fresh** for a set A if there is no term $t \in A$ such that x is free in t. Note that x is fresh in A_0 so that $\langle A_0, A_1 \rangle$ is a rule generated by $\mathcal{T}_\exists$. We say a variable x is **free** in a set A if there is a term $t \in A$ such that x is free in t.

Consider the following simpler (and incorrect) rule schema without the freshness condition:

$$\frac{\exists_\sigma s}{[sx]} \quad x : \sigma$$

Using this rule, we could refute the branch $\{\exists x.px,\ \neg px\}$ even though this branch is clearly satisfiable. That is, one rule in the incorrect schema is $\langle A, A' \rangle$

where A is $\{\exists x.px,\ \neg px\}$ and A' is $\{\exists x.px,\ \neg px,\ px\}$. Since A is satisfiable and A' is unsatisfiable, we know that the rule $\langle A, A'\rangle$ is not sound. Since we only want sound rules in our tableau system, we require the freshness restriction.

Is every rule in the schema $\mathcal{T}_\exists$ sound?

For all previous rules we have argued soundness by showing the stronger property that a model of the head of the rule is also a model of one of the alternatives and applied Proposition 5.2.1. In this case we do not have this stronger form of soundness. Consider the branch $A = \{\exists_\iota p\}$ where $p : \iota o$ and $x : \iota$ are variables. Let $\mathcal{I}$ be a logical interpretation with $\mathcal{I}\iota = \{0, 1\}$, $\mathcal{I}x = 0$, $\mathcal{I}p0 = 0$ and $\mathcal{I}p1 = 1$. Clearly $\mathcal{I} \vDash A$. On the other hand, $\mathcal{I} \nvDash px$.

Every rule given by $\mathcal{T}_\exists$ is sound, but this must be proven by returning to the definition of soundness. A rule is sound if either the head is unsatisfiable or one of the alternatives is satisfiable. Let $\langle A, A \cup \{[sx]\}\rangle$ be a rule given by $\mathcal{T}_\exists$ where $\exists_\sigma s \in A$ and x is fresh for A. Suppose the head A is satisfiable. Let $\mathcal{I}$ be a logical interpretation with $\mathcal{I} \vDash A$. In particular, $\mathcal{I} \vDash \exists s$. Consequently, there is some $a \in \mathcal{I}\sigma$ such that $\hat{\mathcal{I}}sa = 1$. Since x is fresh for A, we can apply Coincidence (Proposition 4.1.2) to conclude $\widehat{\mathcal{I}^x_a}t = \hat{\mathcal{I}}t = 1$ for all $t \in A$. Coincidence also means that $\widehat{\mathcal{I}^x_a}(sx) = \hat{\mathcal{I}}sa = 1$. Therefore, $\mathcal{I}^x_a \vDash A \cup \{[sx]\}$ and the alternative is satisfiable, as desired.

As usual, we also have tableau rules for the negated version of the logical constants.

$$\mathcal{T}_{\neg\forall}\ \frac{\neg\forall_\sigma s}{\neg[sx]}\ x : \sigma \text{ fresh} \qquad\qquad \mathcal{T}_{\neg\exists}\ \frac{\neg\exists_\sigma s}{\neg[st]}\ t : \sigma$$

Note the similarity between the schemas $\mathcal{T}_\forall$ and $\mathcal{T}_{\neg\exists}$ as well as between the schemas $\mathcal{T}_\exists$ and $\mathcal{T}_{\neg\forall}$. Combining these rules with the ones defining TaP, we obtain a tableau calculus TaQ. All the rules defining TaQ are generated by the schemas in Figure 6.2.

Example 6.1.1 We often must apply $\mathcal{T}_\forall$ several times to the same formula with different terms t. (The same is true for $\mathcal{T}_\exists$.) For a simple example of this, we show the formula $(\forall x.px) \to pa \wedge pb$ is valid (where $p : \sigma o$ and $x, a, b : \sigma$ are variables) by refuting its negation. After applying $\mathcal{T}_{\neg\to}$ and $\mathcal{T}_{\neg\wedge}$ we have

$$\frac{\neg((\forall x.px) \to pa \wedge pb),\ \forall x.px,\ \neg(pa \wedge pb)}{\neg pa \mid \neg pb}$$

We close the left branch by applying $\mathcal{T}_\forall$ to $\forall x.px$ with a.

$$\frac{\neg((\forall x.px) \to pa \wedge pb),\ \forall x.px,\ \neg(pa \wedge pb)}{\neg pa,\ pa \mid \neg pb}$$

$$\mathcal{T}_{\bot}\ \frac{\bot}{}\qquad \mathcal{T}_{\neg\top}\ \frac{\neg\top}{}\qquad \mathcal{T}_{\neg}\ \frac{\neg s, s}{}\qquad \mathcal{T}_{\neg\neg}\ \frac{\neg\neg s}{s}\qquad \mathcal{T}_{\wedge}\ \frac{s \wedge t}{s, t}\qquad \mathcal{T}_{\neg\wedge}\ \frac{\neg(s \wedge t)}{\neg s \mid \neg t}$$

$$\mathcal{T}_{\vee}\ \frac{s \vee t}{s \mid t}\qquad \mathcal{T}_{\neg\vee}\ \frac{\neg(s \vee t)}{\neg s, \neg t}\qquad \mathcal{T}_{\rightarrow}\ \frac{s \rightarrow t}{t \mid \neg s}\qquad \mathcal{T}_{\neg\rightarrow}\ \frac{\neg(s \rightarrow t)}{s, \neg t}$$

$$\mathcal{T}_{\mathbf{BQ}}\ \frac{s \equiv t}{s, t \mid \neg s, \neg t}\qquad \mathcal{T}_{\mathbf{BE}}\ \frac{\neg(s \equiv t)}{s, \neg t \mid t, \neg s}\qquad \mathcal{T}_{\forall}\ \frac{\forall_\sigma s}{[st]}\ t : \sigma$$

$$\mathcal{T}_{\exists}\ \frac{\exists_\sigma s}{[sx]}\ x : \sigma \text{ fresh}\qquad \mathcal{T}_{\neg\forall}\ \frac{\neg\forall_\sigma s}{\neg[sx]}\ x : \sigma \text{ fresh}\qquad \mathcal{T}_{\neg\exists}\ \frac{\neg\exists_\sigma s}{\neg[st]}\ t : \sigma$$

Figure 6.2: Tableau rules for TaQ

We close the right branch using $\mathcal{T}_{\forall}$ applied to $\forall x.px$ with b.

$$\frac{\neg((\forall x.px) \rightarrow pa \wedge pb),\ \forall x.px,\ \neg(pa \wedge pb)}{\neg pa,\ pa \mid \neg pb,\ pb}$$

Clearly TaQ does not terminate. For a simple example let $x : \sigma$ be a variable and $x_0, x_1, x_2, \ldots$ be an infinite sequence of distinct variables of type σ. Consider the infinite chain

$$A_0 \subsetneq A_1 \subsetneq A_2 \subsetneq \cdots$$

where A_0 is $\{\forall x.px\}$ and each A_{i+1} is $A_i \cup \{px_i\}$.

Proposition 6.1.2 TaQ does not terminate (on the full fragment).

The example above is somewhat unsatisfying. Technically we have given a chain and proven nontermination. However, there was no real motivation for applying $\mathcal{T}_{\forall}$ to $\forall x.px$ with each x_i. Likewise, we could have applied $\mathcal{T}_{\exists}$ to $\exists x.px$ infinitely many times (with fresh x_i) to obtain a chain. In the examples above we obtained a refutation by applying $\mathcal{T}_{\exists}$ at most once and $\mathcal{T}_{\forall}$ only with terms that only contain variables that occur free in the branch. Such restrictions would rule out the infinite chain above. However, such a restriction on $\mathcal{T}_{\forall}$ would be a bit too strong, as the next example demonstrates.

Example 6.1.3 [Drinker's Paradox] Consider the valid formula

$$\exists x.dx \rightarrow \forall y.dy$$

where $d : \iota o$ and $x, y : \iota$ are variables. This is sometimes called the "Drinker's Paradox" (see Wikipedia), though it is not really a paradox. One can translate the formula into natural language as follows: *In every bar, there is someone who, if he drinks, then everyone drinks.* We show the Drinker's Paradox is valid by refuting it's negation.

$$\neg\exists x.dx \to \forall y.dy$$

Note that there are no variables of type ι free in the branch. If we were only allowed to apply $\mathcal{T}_{\neg\exists}$ to terms containing free names in the branch, then we would be stuck. However, if we apply $\mathcal{T}_{\neg\exists}$ using x, then we have

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

Applying a few more rules we obtain

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

We now apply $\mathcal{T}_{\neg\exists}$ to $\neg\exists x.dx \to \forall y.dy$ again, this time using y.

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

Applying $\mathcal{T}_{\neg\to}$ to $\neg(dy \to \forall y.dy)$ adds dy to the branch completing the refutation.

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

We will revisit restricting the $\mathcal{T}_\forall$ and $\mathcal{T}_\exists$ to only use variables free in the branch later. Examples such as Example 6.1.3 will be dealt with by allowing the use of a fixed default variable of the right type if no variable of the right type occurs free in the branch. For now we will informally describe why the tableau system would still not terminate even with such a restriction.

Consider the formula

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

The only free variable is r, which is clearly of a different type than x. Suppose x_0 is the default variable we are allowed to use in such a situation. Applying $\mathcal{T}_\forall$ with x_0 we add $\exists y.rx_0y$ to the branch. Applying $\mathcal{T}_\exists$ with a fresh variable x_1 we add rx_0x_1 to the branch. Now since x_1 is free in the branch we can apply $\mathcal{T}_\forall$ and obtain $\exists y.rx_1y$. Clearly we can keep applying rules in this manner forever.

6.2 First-Order Examples

In this section we give a number of examples of tableau refutations restricting ourselves to first-order formulas (without equality). Since the formulas are first-order, all quantifiers will be over a sort α. For simplicity assume all bound variables are of a particular sort ι.

Example 6.2.1 We refute the following branch:

$$\begin{array}{c} \forall x \exists y.rxy \\ \neg\forall x \exists yz.rxy \wedge ryz \end{array}$$

$\mathcal{T}_{\neg\forall}$ applied to $\neg\forall x \exists yz.rxy \wedge ryz$ using x gives

$$\begin{array}{c} \forall x \exists y.rxy \\ \neg\forall x \exists yz.rxy \wedge ryz \\ \neg\exists yz.rxy \wedge ryz \end{array}$$

We apply $\mathcal{T}_{\forall}$ to $\forall x \exists y.rxy$ with x and then $\mathcal{T}_{\exists}$ to $\exists y.rxy$ with y to obtain

$$\begin{array}{c} \forall x \exists y.rxy \\ \neg\forall x \exists yz.rxy \wedge ryz \\ \neg\exists yz.rxy \wedge ryz,\ \exists y.rxy,\ rxy \end{array}$$

$\mathcal{T}_{\neg\exists}$ applied to $\neg\exists yz.rxy \wedge ryz$ using y gives

$$\begin{array}{c} \forall x \exists y.rxy \\ \neg\forall x \exists yz.rxy \wedge ryz \\ \neg\exists yz.rxy \wedge ryz,\ \exists y.rxy,\ rxy \\ \neg\exists z.rxy \wedge ryz \end{array}$$

We now revisit $\forall x \exists y.rxy$ this time instantiating with y to add $\exists y'.ryy'$ to the branch. Notice that substituting y for x in $\exists y.rxy$ required renaming the bound variable y to be y'. Applying $\mathcal{T}_{\exists}$ to $\exists y'.ryy'$ with z we obtain

$$\begin{array}{c} \forall x \exists y.rxy \\ \neg\forall x \exists yz.rxy \wedge ryz \\ \neg\exists yz.rxy \wedge ryz,\ \exists y.rxy,\ rxy \\ \neg\exists z.rxy \wedge ryz,\ \exists y'.ryy',\ ryz \end{array}$$

We complete the refutation by applying $\mathcal{T}_{\neg\exists}$ to $\neg\exists z.rxy \wedge ryz$ using z followed by $\mathcal{T}_{\neg\wedge}$.

$$\begin{array}{c} \forall x \exists y.rxy \\ \neg\forall x \exists yz.rxy \wedge ryz \\ \neg\exists yz.rxy \wedge ryz,\ \exists y.rxy,\ rxy \\ \neg\exists z.rxy \wedge ryz,\ \exists y'.ryy',\ ryz \\ \neg(rxy \wedge ryz) \\ \hline \neg rxy \quad\vert\quad \neg ryz \end{array}$$

Example 6.2.2 We prove $(\forall x.px \wedge qx) \to \forall x.px$ by refuting its negation using the following tableau.

$$\begin{array}{c} \neg((\forall x.px \wedge qx) \to \forall x.px) \\ \forall x.px \wedge qx,\ \neg\forall x.px,\ \neg px,\ px \wedge qx,\ px,\ qx \end{array}$$

Example 6.2.3 We refute the following branch:

$$\forall x.px \to qx,\ \forall x.px,\ \neg\forall x.qx$$

Applying $\mathcal{T}_{\neg\forall}$ with x and $\mathcal{T}_{\forall}$ to both universally quantified formulas with x we obtain the refutation.

$$\frac{\forall x.px \to qx,\ \forall x.px,\ \neg\forall x.qx,\ \neg qx,\ px,\ px \to qx}{\begin{array}{c|c} qx & \neg px \end{array}}$$

Example 6.2.4 The equation $(\exists x.px \wedge q) \equiv (\exists x.px) \wedge q$ is valid. Below we show the refutation of one direction, leaving the full equivalence as an exercise.

$$\frac{\begin{array}{c} \exists x.px \wedge q \\ \neg((\exists x.px) \wedge q),\ px \wedge q,\ px,\ q \end{array}}{\begin{array}{c|c} \neg\exists x.px,\ \neg px & \neg q \end{array}}$$

Example 6.2.5 (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 y if and only if y does not halt on the representation of y?
3. Does there exist a set that contains a set y as element if and only if $y \notin y$? □

The answer to all 3 questions is no, and the reason is purely logical.

Russell's or Turing's Law: Let $f : \iota\iota o$ be a name and $x, y : \iota$ be names. We can prove $\neg\exists x\forall y.fxy \equiv \neg fyy$ by refuting its negation.

$$\frac{\begin{array}{c} \exists x\forall y.fxy \equiv \neg fyy \\ \forall y.fxy \equiv \neg fyy \\ fxx \equiv \neg fxx \end{array}}{\begin{array}{c|c} fxx & \neg fxx \\ \neg fxx & \neg\neg fxx \end{array}}$$

Exercise 6.2.6 Prove the validity of each of the following formulas by giving a TaQ tableau refutation of its negation.

a) $(\forall x.px \wedge qx) \equiv (\forall x.px) \wedge \forall x.qx$

b) $(\forall x.px) \vee (\forall x.qx) \to \forall x.px \vee qx$

c) $(\exists x.px \vee qx) \equiv (\exists x.px) \vee \exists x.qx$

d) $(\exists x.px \wedge qx) \to (\exists x.px) \wedge \exists x.qx$

e) $(\exists x.px \wedge q) \equiv (\exists x.px) \wedge q$

f) $(\forall x.px \vee q) \equiv (\forall x.px) \vee q$

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

Exercise 6.2.7 Give a TaQ tableau refutation of the branch with the three formulas

$$\forall xy.rxy \to ryx$$
$$\forall xyz.rxy \to ryz \to rxz$$
$$\neg\forall x.(\exists y.rxy) \to rxx$$

6.3 Higher-Order Examples

We now consider a few higher-order examples. Many of these examples make use of Leibniz equality.

Example 6.3.1 Let $f : \sigma\sigma$, $p : \sigma o$ and $x : \sigma$ be variables. We prove $\exists f.\forall xp.p(fx) \to px$ by refuting its negation. Essentially we want to prove that there is a function f such that for all x, fx is equal to x (in the sense of Leibniz). In the first step we must give an instantiation term for the $\mathcal{T}_{\neg\exists}$ rule. We use $\lambda x.x$ as this term. In this case, normalization will reduce

$$(\lambda f.\forall xp.p(fx) \to px)(\lambda x.x)$$

to

$$\forall xp.px \to px$$

Hence we obtain the following branch:

$$\neg\exists f\forall xp.p(fx) \to px$$
$$\neg\forall xp.px \to px$$

Using the obvious rules we complete the refutation.

$$\neg\exists f\forall xp.p(fx) \to px$$
$$\neg\forall xp.px \to px$$
$$\neg\forall p.px \to px$$
$$\neg(px \to px)$$
$$px$$
$$\neg px$$

Example 6.3.2 Next we prove that $\bot$ is not Leibniz equal to $\top$. We start with a formula asserting that $\bot$ and $\top$ are Leibniz equal.

$$\forall p.p\bot \to p\top$$

The rule $\mathcal{T}_\forall$ applied to $\forall p.p\bot \to p\top$ using $\neg$ gives

$$\begin{array}{c}\forall p.p\bot \to p\top\\ \neg\bot \to \neg\top\end{array}$$

We complete the refutation using propositional rules.

$$\dfrac{\begin{array}{c}\forall p.p\bot \to p\top\\ \neg\bot \to \neg\top\end{array}}{\neg\top \mid \neg\neg\bot, \bot}$$

We next give two tableau refutations which prove Leibniz equality is symmetric. In both cases, we will refute the following branch:

$$\begin{array}{c}\forall p.pa \to pb\\ \neg\forall p.pb \to pa\end{array}$$

That is, we begin by assuming a is Leibniz equal to b, but b is not Leibniz equal to a.

Example 6.3.3 (Symmetry of Leibniz, Proof I) The rule $\mathcal{T}_{\neg\forall}$ applied to $\neg\forall p.pb \to pa$ using p followed by $\mathcal{T}_{\neg\to}$ gives

$$\begin{array}{c}\forall p.pa \to pb\\ \neg\forall p.pb \to pa\\ \neg(pb \to pa),\ pb,\ \neg pa\end{array}$$

Applying $\mathcal{T}_\forall$ with $\lambda x.\neg px$ followed by $\mathcal{T}_\to$ we obtain the refutation

$$\dfrac{\begin{array}{c}\forall p.pa \to pb\\ \neg\forall p.pb \to pa\\ \neg(pb \to pa),\ pb,\ \neg pa,\ \neg pa \to \neg pb\end{array}}{\neg pb \mid \neg\neg pa}$$

Example 6.3.4 (Symmetry of Leibniz, Proof II) We begin again with the branch

$$\begin{array}{c}\forall p.pa \to pb\\ \neg\forall p.pb \to pa\end{array}$$

This time instead of applying $\mathcal{T}_{\neg\forall}$ we start by applying $\mathcal{T}_{\forall}$ using $\lambda x.\forall p.px \to pa$ gives

$$\begin{array}{c}\forall p.pa \to pb\\ \neg\forall p.pb \to pa\\ (\forall p.pa \to pa) \to \forall p.pb \to pa\end{array}$$

We apply $\mathcal{T}_{\to}$ to $(\forall p.pa \to pa) \to \forall p.pb \to pa$ and obtain

$$\begin{array}{c}\forall p.pa \to pb\\ \neg\forall p.pb \to pa\\ (\forall p.pa \to pa) \to \forall p.pb \to pa\\ \hline \forall p.pb \to pa \mid \neg\forall p.pa \to pa\end{array}$$

Note that the left branch is closed already. Two more rule applications close the right branch.

$$\begin{array}{c}\forall p.pa \to pb\\ \neg\forall p.pb \to pa\\ (\forall p.pa \to pa) \to \forall p.pb \to pa\\ \hline \forall p.pb \to pa \mid \begin{array}{c}\neg\forall p.pa \to pa\\ \neg(pa \to pa),\ pa,\ \neg pa\end{array}\end{array}$$

Our final higher-order example of this section is a version of **Cantor's Theorem**. Consider a type σ. The type σo corresponds to sets of elements of type σ. Cantor's Theorem states that there are always more elements of σo than there are elements of σ. One way to express this precisely is to say there is no surjection from σ onto σo. The following formula

$$\neg\exists f\forall g\exists x.fx = g$$

where $f : \sigma\sigma o$, $g : \sigma o$ and $x : \sigma$ are variables expresses Cantor's Theorem. However, this formula contains an identity $=_{\sigma o}$ at a type other than o. Consequently, we cannot currently prove this version of Cantor's Theorem. On the other hand, we know that two functions are equal if they give the same result for any inputs. Hence we can express Cantor's Theorem as the formula

$$\neg\exists f.\forall g.\exists x.\forall y.fxy \equiv gy$$

where $y : \sigma$ is a variable distinct from x. We refute the negation of this version of Cantor's Theorem.

Example 6.3.5 (Cantor's Theorem, Version I) We refute the following branch:

$$\exists f\forall g\exists x\forall y.fxy \equiv gy$$

The rule $\mathcal{T}_\exists$ applied to $\exists f \forall g \exists x \forall y. fxy \equiv gy$ using f gives

$$\begin{array}{c} \exists f \forall g \exists x \forall y. fxy \equiv gy \\ \forall g \exists x \forall y. fxy \equiv gy \end{array}$$

We now want to apply $\mathcal{T}_\forall$ with a term of type σo. We use the term $\lambda x. \neg fxx$. We then apply $\mathcal{T}_\exists$ to obtain

$$\begin{array}{c} \exists f \forall g \exists x \forall y. fxy \equiv gy \\ \forall g \exists x. \forall y. fxy \equiv gy \\ \exists x \forall y. fxy \equiv \neg fyy \\ \forall y. fxy \equiv \neg fyy \end{array}$$

We complete the refutation by applying $\mathcal{T}_\forall$ to $\forall y. fxy \equiv \neg fyy$ using x followed by $\mathcal{T}_{\text{BQ}}$.

$$\begin{array}{c} \exists f \forall g \exists x \forall y. fxy \equiv gy \\ \forall g \exists x \forall y. fxy \equiv gy \\ \exists x \forall y. fxy \equiv \neg fyy \\ \forall y. fxy \equiv \neg fyy \\ fxx \equiv \neg fxx \\ \hline \begin{array}{c|c} fxx & \neg fxx \\ \neg fxx & \neg\neg fxx \end{array} \end{array}$$

Exercise 6.3.6 (Cantor's Theorem, Version II) Prove the validity of the following version of Cantor's Theorem by refuting its negation.

$$\neg \exists f \forall g \exists x \forall p. p(fx) \to pg$$

where $f : \iota\iota o$, $g : \iota o$, $p : (\iota o)o$ and $x : \iota$ are variables.

Exercise 6.3.7 Let $f : \iota\iota o$, $g : \iota o$, $x : \iota$ and $h : (\iota o)o$ be variables. Consider the formula

$$\exists g \forall x \exists h. h(fx) \wedge \neg hg$$

Determine if this formula is valid, unsatisfiable or neither. If it is unsatisfiable, give a TaQ tableau refutation of it. If it is valid, give a TaQ tableau refutation of its negation. If it is neither valid nor unsatisfiable, go to the next problem.

Exercise 6.3.8 Recall the Peano axiom of induction (where N is a sort and $o : N$, $S : NN$, $p : No$ and $x : N$ are variables):

$$\forall p.\ \ po \to (\forall x.\ px \to p(Sx)) \to (\forall x. px)$$

Let induction refer to the Peano axiom of induction. The following formula says that any nonempty set that is closed under predecessor must contain zero:

$$\forall p.(\exists x.px) \wedge (\forall x.\, p(Sx) \rightarrow px) \rightarrow po$$

Let induction2 refer to this formula. Prove induction and induction2 are equivalent by giving a TaQ refutation of

$$\{\text{induction}, \neg\text{induction2}\}$$

and a TaQ refutation of

$$\{\neg\text{induction}, \text{induction2}\}$$

Exercise 6.3.9 Give TaQ tableau refutations of the following formulas.

a) $\forall p.(p\neg) \rightarrow p(\lambda x.\top)$ where $p : (oo)o$ and $x : o$ are variables.

b) $\forall p.(p\neg) \rightarrow p(\lambda x.\bot)$ where $p : (oo)o$ and $x : o$ are variables.

6.4 Completeness for Pure First-Order Formulas

Recall that pure first-order formulas are first-order formulas which contain no equations and no variables of a type $\alpha_1 \ldots \alpha_n\alpha$ where $n \geq 1$. For simplicity we will restrict our attention to one sort ι and the constants $\neg$, $\wedge$ and $\forall_\iota$. We can directly describe this fragment of pure first-order formulas by the grammar

$$s ::= px \cdots x \mid \neg s \mid s \wedge s \mid \forall_\iota x.s$$

where $p : \iota \cdots \iota o$ and $x : \iota$ are variables. We say A is a **pure first-order branch** if it is a branch containing only formulas in this fragment.

Our goal in this section is to prove completeness of TaQ with respect to the pure first-order fragment. We will actually prove completeness of a more restricted tableau system TaQ^f where the applicability of $\mathcal{T}_\forall$ and $\mathcal{T}_{\neg\forall}$ are restricted.

Gödel is generally credited as the first to prove a completeness theorem for first-order logic [31]. One could argue that Skolem was the first to prove completeness in [60]. Henkin later proved a completeness theorem for simple type theory [35] and realized he could use some of the ideas to give a simpler completeness theorem for first-order logic [37]. Smullyan generalized Henkin's techniques to give a very modular proof of completeness. We use Smullyan's abstract consistency method for proving completeness of TaQ^f for pure first-order logic. We will later use this same method to prove completeness of a tableau system for simple type theory.

$$\mathcal{T}_{\neg}\ \frac{\neg s, s}{} \qquad \mathcal{T}_{\neg\neg}\ \frac{\neg\neg s}{s} \qquad \mathcal{T}_{\wedge}\ \frac{s \wedge t}{s, t} \qquad \mathcal{T}_{\neg\wedge}\ \frac{\neg(s \wedge t)}{\neg s \mid \neg t}$$

$$\mathcal{T}^{f}_{\forall}\ \frac{\forall_\iota s}{[s\mathcal{Y}]}\ \mathcal{Y} \in \mathcal{H}(A),\ A \text{ branch}$$

$$\mathcal{T}^{f}_{\neg\forall}\ \frac{\neg\forall_\iota s}{\neg[s\mathcal{Y}]}\ \mathcal{Y} : \iota \text{ fresh},\ (\neg\forall x.s) \text{ not evident on branch}$$

Figure 6.3: Tableau rules for TaQ^f

Fix a variable x_0 of type ι. This variable x_0 will act as a **default variable** for applying the $\mathcal{T}_\forall$ rule when there is no variable of type ι free in the branch.

Let A be a set of formulas. We make the following definitions.

- Let $\mathcal{V}_\iota A$ denote the set of variables $x : \iota$ free in A.
- For any set A of formulas, let $\mathcal{H}(A)$ be defined as follows:

$$\mathcal{H}(A) := \begin{cases} \mathcal{V}_\iota A & \text{if } \mathcal{V}_\iota A \neq \emptyset \\ \{x_0\} & \text{otherwise.} \end{cases}$$

  We call $\mathcal{H}(A)$ the **Herbrand universe** of A. Note that the Herbrand universe of A is always nonempty by construction.
- We say a formula $\neg\forall_\iota x.s$ is **evident** in A if $\neg s^x_{\mathcal{Y}} \in A$ for some variable $\mathcal{Y} : \iota$.

We now define TaQ^f to be the tableau calculus given by Figure 6.3. The following proposition is obvious.

Proposition 6.4.1 If a branch A is TaQ^f-refutable, then it is also TaQ-refutable.

Proof Every rule in TaQ^f is also a rule in TaQ. (The reader should verify this.) ∎

6.4.1 Termination of the Bernays-Schönfinkel Fragment

Clearly TaQ^f does not terminate on the pure first-order fragment. Consider the formula

$$\forall x.\neg\forall y.rxy$$

We can apply $\mathcal{T}^f_\forall$ and $\mathcal{T}^f_{\neg\forall}$ in sequence to add formulas of the form $\neg rx_0x_1$, $\neg rx_1x_2$, and so on.

On the other hand, TaQ^f does terminate if A is a branch only containing pure first-order formulas of the form

$$\forall x_1 \cdots \forall x_n . s$$

where s is quantifier-free. This is easy to see: The Herbrand universe of A is finite and will not become bigger as we apply rules. Hence the $\mathcal{T}^f_\forall$ rule can only be applied finitely many times. It is also easy to see that we could use the same technique if A only contained formulas of the form

$$\exists y_1 \cdots \exists y_m \forall x_1 \cdots \forall x_n . s$$

where s is quantifier-free (assuming we include a rule like $\mathcal{T}_\exists$ but restricted like $\mathcal{T}^f_{\neg\forall}$).

This fragment is known as the Bernays-Schönfinkel fragment of first-order logic. For this fragment we can show completeness simply by showing verification soundness. As before, verification soundness follows by proving a Model Existence Theorem for evident sets.

6.4.2 Evident Sets and Model Existence

As in the propositional case, we will describe **evidence conditions** that a set E of formulas may satisfy.

$\mathcal{E}_\neg$ If $\neg s \in E$, then $s \notin E$.

$\mathcal{E}_{\neg\neg}$ If $\neg\neg s \in E$, then $s \in E$.

$\mathcal{E}_\wedge$ If $s \wedge t \in E$, then $s \in E$ and $t \in E$.

$\mathcal{E}_{\neg\wedge}$ If $\neg(s \wedge t) \in E$, then $\neg s \in E$ or $\neg t \in E$.

$\mathcal{E}^f_\forall$ If $\forall_\iota s \in E$, then $[s\mathcal{Y}] \in E$ for all $\mathcal{Y} \in \mathcal{H}(E)$.

$\mathcal{E}^f_{\neg\forall}$ If $\neg\forall_\iota s \in E$, then $\neg[s\mathcal{Y}] \in E$ for some $\mathcal{Y} : \iota$.

We say E is **pure first-order evident** (or simply **evident**) if it satisfies these evidence conditions.

In analogy with the propositional case, we can prove the existence of a model of any evident set.

Theorem 6.4.2 (Pure First-Order Model Existence) Let E be a set of pure first-order formulas. If E is pure first-order evident, then there is a (standard) model of E.

Proof Let $\mathcal{I}$ be a standard interpretation with

$$\mathcal{I}\iota = \mathcal{H}E$$

$$\mathcal{I}x = x$$

for each $x \in \mathcal{H}E$ and

$$\mathcal{I}px_1 \cdots x_n = \begin{cases} 1 & \text{if } px_1 \cdots x_n \in E \\ 0 & \text{otherwise} \end{cases}$$

for each variable $p : \iota \cdots \iota o$ and $x_1, \ldots, x_n \in \mathcal{H}E$. We will prove $\mathcal{I} \vDash E$ by inductively proving for every pure first-order s we have

(1_s) If $s \in E$, then $\mathcal{I} \vDash s$.

(2_s) If $\neg s \in E$, then $\mathcal{I} \nvDash s$.

Since every $s \in E$ is pure first-order, the first fact ensures $\mathcal{I} \vDash E$.

We first consider s of the form $px_1 \cdots x_n$.

$(1_{px_1 \cdots x_n})$ Assume $px_1 \cdots x_n \in E$. By the choice of $\mathcal{I}p$ we know $\mathcal{I} \vDash px_1 \cdots x_n$.

$(2_{px_1 \cdots x_n})$ Assume $\neg px_1 \cdots x_n \in E$. By $\mathcal{E}_\neg$ we have $px_1 \cdots x_n \notin E$. By the choice of $\mathcal{I}p$ we know $\mathcal{I} \nvDash px_1 \cdots x_n$.

We next consider s of the form $\neg t$.

$(1_{\neg t})$ Assume $\neg t \in E$. The inductive hypothesis (2_t) implies $\mathcal{I} \nvDash t$ and so $\mathcal{I} \vDash \neg t$.

$(2_{\neg t})$ Assume $\neg\neg t \in E$. By $\mathcal{E}_{\neg\neg}$ we have $t \in E$ and so $\mathcal{I} \vDash t$ by the inductive hypothesis (1_t). Hence $\mathcal{I} \nvDash \neg t$.

We now consider s of the form $t \wedge u$.

$(1_{t \wedge u})$ Assume $(t \wedge u) \in E$. By $\mathcal{E}_\wedge$ we have $t \in E$ and $u \in E$. By inductive hypotheses (1_t) and (1_u) we have $\mathcal{I} \vDash t$ and $\mathcal{I} \vDash u$ and so $\mathcal{I} \vDash t \wedge u$.

$(2_{t \wedge u})$ Assume $\neg(t \wedge u) \in E$. By $\mathcal{E}_{\neg\wedge}$ we have $\neg t \in E$ or $\neg t \in E$. If $\neg t \in E$, then $\mathcal{I} \nvDash t$ by inductive hypothesis (2_t) and so $\mathcal{I} \nvDash t \wedge u$. If $\neg u \in E$, then $\mathcal{I} \nvDash u$ by inductive hypothesis (2_u) and so $\mathcal{I} \nvDash t \wedge u$.

We finally consider s of the form $\forall_\iota x.t$.

$(1_{\forall x.t})$ Assume $(\forall x.t) \in E$. In order to prove $\mathcal{I} \vDash \forall x.t$ we must prove $(\mathcal{I})^x_y \vDash t$ for every $y \in \mathcal{I}\iota$. Let $y \in \mathcal{I}\iota$ be given. That is, $y \in \mathcal{H}(E)$. By $\mathcal{E}^f_\forall$ we know $t^x_y \in E$. By inductive hypothesis $(1_{t^x_y})$ we have $\mathcal{I} \vDash t^x_y$. This implies $(\mathcal{I})^x_y \vDash t$ as desired.

$(2_{\forall x.t})$ Assume $\neg(\forall x.t) \in E$. We first argue that there is some $z \in \mathcal{H}(E)$ such that $\neg t^x_z \in E$. By $\mathcal{E}^f_{\neg\forall}$ there is some y such that $\neg t^x_y \in E$. If $y \in \mathcal{H}(E)$, then we take z to be this y. Otherwise, x is not free in t and we take any $z \in \mathcal{H}(E)$. (If x is not free in t, t^x_y is the same as t^x_z for any z.) By inductive hypothesis $(2_{t^x_z})$ we have $\mathcal{I} \nvDash t^x_z$. Hence $\mathcal{I}^x_z \nvDash t$ and so $\mathcal{I} \nvDash \forall x.t$. ∎

Verification soundness follows immediately from the Model Existence Theorem and the fact that open maximal branches are evident. Since we do not have

$C_\neg$	If $\neg s$ is in A, then s is not in A.
$C_{\neg\neg}$	If $\neg\neg s$ is in A, then $A \cup \{s\}$ is in Γ.
$C_\wedge$	If $s \wedge t$ is in A, then $A \cup \{s, t\}$ is in Γ.
$C_{\neg\wedge}$	If $\neg(s \wedge t)$ is in A, then $A \cup \{\neg s\}$ or $A \cup \{\neg t\}$ is in Γ.
$C^f_\forall$	If $\forall_\iota s$ is in A, then $A \cup \{[s\mathcal{Y}]\}$ is in Γ for all $\mathcal{Y} \in \mathcal{H}(A)$.
$C^f_{\neg\forall}$	If $\neg\forall_\iota s$ is in A, then $A \cup \{\neg[s\mathcal{Y}]\}$ is in Γ for some $\mathcal{Y}$.

Figure 6.4: Pure first-order abstract consistency conditions (must hold for every $A \in \Gamma$)

termination, we cannot yet conclude completeness. A new idea (abstract consistency) is required.

6.4.3 Abstract Consistency and the Extension Lemma

We say a branch A is **TaQ^f-consistent** (or **consistent**) if it is not TaQ^f-refutable. Ultimately we will show completeness by showing every consistent branch is satisfiable. Due to the Model Existence Theorem we know that it is enough to show every consistent branch can be extended to an evident set. In particular, we are interested in ways consistent sets can be extended while remaining consistent.

A **(pure first-order) abstract consistency class** is a set Γ of branches such that every branch $A \in \Gamma$ satisfies the conditions in Figure 6.4.

Lemma 6.4.3 (Extension Lemma) Let Γ be a pure first-order abstract consistency class and $A \in \Gamma$. Then there exists a pure first-order evident set E such that $A \subseteq E$.

Proof Let $u_0, u_1, u_2, \ldots$ be an enumeration of all pure first-order formulas. We construct a sequence $A_0 \subseteq A_1 \subseteq A_2 \subseteq \cdots$ of branches such that every $A_n \in \Gamma$. Let $A_0 := A$. We define A_{n+1} by cases. If there is no $B \in \Gamma$ such that $A_n \cup \{u_n\} \subseteq B$, then let $A_{n+1} := A_n$. Otherwise, choose some $B \in \Gamma$ such that $A_n \cup \{u_n\} \subseteq B$. We consider two subcases.

1. If u_n is of the form $\neg\forall_\iota x.s$, then choose A_{n+1} to be $B \cup \{\neg s^x_\mathcal{Y}\} \in \Gamma$ for some variable $\mathcal{Y} : \iota$. This is possible since Γ satisfies $C^f_{\neg\forall}$.
2. Otherwise, then let A_{n+1} be B.

Let $E := \bigcup_{n\in\mathbb{N}} A_n$. Note that $\mathcal{H}E \subseteq \bigcup_{n\in\mathbb{N}} \mathcal{H}A_n$. (It is possible that $\mathcal{H}(A_n) = \{x_0\}$ for some n but that $x_0 \notin \mathcal{H}E$.) We show that E is evident.

$\mathcal{E}_\neg$ If $\neg s$ and s are in E, then $\neg s$ and s are in A_n for some n, contradicting $C_\neg$.

$\mathcal{E}_{\neg\neg}$ Assume $\neg\neg s$ is in E. Let n be such that $u_n = s$ and $r \geq n$ be such that $\neg\neg s$ is in A_r. Since $A_n \cup \{s\} \subseteq A_r \cup \{s\} \in \Gamma$ (using $C_{\neg\neg}$), we have $s \in A_{n+1} \subseteq E$.

$\mathcal{E}_\wedge$ Assume $s \wedge t$ is in E. Let n, m be such that $u_n = s$ and $u_m = t$. Let $r \geq n, m$ be such that $s \wedge t$ is in A_r. By $C_\wedge$, $A_r \cup \{s, t\} \in \Gamma$. Since $A_n \cup \{s\} \subseteq A_r \cup \{s, t\}$, we have $s \in A_{n+1} \subseteq E$. Since $A_m \cup \{t\} \subseteq A_r \cup \{s, t\}$, we have $t \in A_{m+1} \subseteq E$.

$\mathcal{E}_{\neg\wedge}$ Assume $\neg(s \wedge t)$ is in E. Let n, m be such that $u_n = s$ and $u_m = t$. Let $r \geq n, m$ be such that $\neg(s \wedge t)$ is in A_r. By $C_{\neg\wedge}$, $A_r \cup \{\neg s\} \in \Gamma$ or $A_r \cup \{\neg t\} \in \Gamma$. In the first case, $A_n \cup \{\neg s\} \subseteq A_r \cup \{\neg s\} \in \Gamma$, and so $\neg s \in A_{n+1} \subseteq E$. In the second case, $A_m \cup \{\neg t\} \subseteq A_r \cup \{\neg t\} \in \Gamma$, and so $\neg t \in A_{m+1} \subseteq E$. Hence either $\neg s$ or $\neg t$ is in E.

$\mathcal{E}_\forall$ Assume $\forall_\iota x.s$ is in E. Let $y \in \mathcal{H}(E)$ be given. Let n be such that $u_n = s^x_y$. Let $r \geq n$ be such that $\forall x.s$ is in A_r and $y \in \mathcal{H}(A_r)$. By $C_\forall$ we have $A_r \cup \{s^x_y\} \in \Gamma$. Since $A_n \cup \{u_n\} \subseteq A_r \cup \{s^x_y\}$, we have $s^x_y = u_n \in A_{n+1} \subseteq E$.

$\mathcal{E}_{\neg\forall}$ Assume $\neg\forall_\iota x.s$ is in E. Let n be such that $u_n = \neg\forall x.s$. Let $r \geq n$ be such that $\neg\forall x.s$ is in A_r. Since $A_n \cup \{\neg\forall x.s\} \subseteq A_r \in \Gamma$, we have $s^x_y \in A_{n+1} \subseteq E$ for some y by construction. ∎

6.4.4 Completeness

Let Γ^f_{TaQ} be the set of all branches A which are not TaQ^f-refutable. We will prove Γ^f_{TaQ} is a pure first-order abstract consistency class and use this to prove TaQ^f is complete with respect to pure first-order branches.

Lemma 6.4.4 Γ^f_{TaQ} is a pure first-order abstract consistency class.

Proof We must check six conditions.

$C_\neg$ Suppose $\neg s, s \in A \in \Gamma^f_{\mathsf{TaQ}}$. Then we can refute A using $\mathcal{T}_\neg$. Contradiction.

$C_{\neg\neg}$ Let $\neg\neg s \in A \in \Gamma^f_{\mathsf{TaQ}}$. Suppose $A \cup \{s\}$ is not in Γ^f_{TaQ}. Then $A \cup \{s\}$ is refutable and so A is refutable by $\mathcal{T}_{\neg\neg}$, a contradiction.

$C_\wedge$ Let $(s \wedge t) \in A \in \Gamma^f_{\mathsf{TaQ}}$. Suppose $A \cup \{s, t\}$ is not in Γ^f_{TaQ}. Then $A \cup \{s, t\}$ is refutable. Hence A can be refuted using $\mathcal{T}_\wedge$. Contradiction.

$C_{\neg\wedge}$ Let $\neg(s \wedge t) \in A \in \Gamma^f_{\mathsf{TaQ}}$. Suppose $A \cup \{\neg s\}$ and $A \cup \{\neg t\}$ are not in Γ^f_{TaQ}. Then $A \cup \{\neg s\}$ and $A \cup \{\neg t\}$ are refutable. Hence A can be refuted using $\mathcal{T}_{\neg\wedge}$. Contradiction.

$C^f_\forall$ Let $\forall_\iota x.s \in A \in \Gamma^f_{\mathsf{TaQ}}$. Suppose $A \cup \{s^x_y\} \notin \Gamma^f_{\mathsf{TaQ}}$ for some $y \in \mathcal{H}(A)$. Then $A \cup \{s^x_y\}$ is refutable. Hence A can be refuted using $\mathcal{T}^f_\forall$.

$C^f_{\neg\forall}$ Let $\neg\forall_\iota x.s \in A \in \Gamma^f_{\mathsf{TaQ}}$. Suppose $A \cup \{\neg s^x_y\} \notin \Gamma^f_{\mathsf{TaQ}}$ for every $y : \iota$. Then $A \cup \{\neg s^x_y\}$ is refutable for every $y : \iota$. Hence A is refutable using $\mathcal{T}^f_{\neg\forall}$ and the finiteness of A. Contradiction. ∎

Theorem 6.4.5 (Pure First-Order Completeness) TaQ^f is complete with respect to pure first-order branches.

Proof Assume A is a pure first-order branch which is not TaQ^f-refutable. Then $A \in \Gamma^f_{\mathsf{TaQ}}$. By Lemma 6.4.4 Γ^f_{TaQ} is a pure first-order abstract consistency class. By the Extension Lemma (Lemma 6.4.3) there is a pure first-order evident set E such that $A \subseteq E$. By the Model Existence Theorem (Theorem 6.4.2) E is satisfiable. Hence A is satisfiable. ∎

6.5 Remarks

In Exercise 6.2.6 the validity of certain quantifier laws such as

$$(\exists x.px \wedge q) \equiv (\exists x.px) \wedge q$$

and

$$(\forall x.px \vee q) \equiv (\forall x.px) \vee q$$

have been established. Using quantifier laws such as these it is easy to transform any first-order formula into a *prenex* form:

$$Q_1 x_1 \cdots Q_m x_m.s$$

where each Q_i is $\forall$ or $\exists$ and s is quantifier-free. Note that the prenex formula

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

is satisfiable if and only if

$$\forall x.rx(fx)$$

is satisfiable. In automated theorem proving, the new variable f is called a **Skolem function** since Skolem considered such formulas in [60]. In this way, we can reduce satisfiability of sets of first-order formulas to satisfiability of first order formulas of the form

$$\forall x_1 \cdots \forall x_m.s$$

where s is quantifier-free.

Given a set A of first-order formulas, one may naturally consider the terms generated by the grammar

$$t ::= ft \cdots t$$

where $f : \alpha_1 \cdots \alpha_m \alpha$ (with $m \geq 0$) is free in A. This is the natural generalization of the **Herbrand universe** $\mathcal{H}(A)$ from the pure first-order case to the general first-order case. For simplicity, let us assume that for each sort α there is some $x : \alpha$ free in A. In the general first-order case it makes sense to refine the Herbrand universe to be $\mathcal{H}_n(A)$ by iteratively defining

$$\mathcal{H}_0(A) := \{x : \alpha | x \text{ is free in } A\}$$

and

$$\mathcal{H}_{n+1}(A) := \{f t_1 \cdots t_m | t_1, \ldots, t_m \in \mathcal{H}_n(A), f \text{ is free in } A, m \geq 0\}.$$

Both Skolem [60] and Herbrand [38] considered such terms, so we could just as well call it the Skolem universe.

In [60] Skolem showed that if A is a set of first-order formulas of the form

$$\forall x_1 \cdots \forall x_m . s$$

where s is quantifier-free, then there is some n such that unsatisfiability of A becomes clear by considering terms from $\mathcal{H}_n(A)$. Herbrand showed related results in [38]. Taken together, the results in [60] and [38] can be taken as a proof of completeness of first-order logic. (As noted earlier, the first proof of completeness was given by Gödel [31].) Certainly the techniques of Skolem and Herbrand in [60] and [38] had a significant impact years later in the field of automated deduction [25]. The Herbrand Award for Distinguished Contributions to Automated Deduction is named after Herbrand.

Henkin's proof of completeness [37] primarily consisted of extending a consistent set to a maximally consistent set. This is the essence of the proof of the Extension Lemma (Lemma 6.4.3). A set is maximally consistent if adding any new closed formula to the set would make the set inconsistent. Once one has a maximally consistent set, obtaining a model of this set is easy.

In [43] Hintikka showed how to give a model of a set which may not be maximally consistent. It is enough to have a set that satisfies certain closure conditions. Hintikka called such a set a **model set**. We have called such a set **evident**. The fact that evident sets are satisfiable is sometimes called Hintikka's Lemma (e.g., in [61]). We have called it the Model Existence Theorem (Theorem 6.4.2).

While Henkin argued using consistency, Smullyan recognized that the technique only relied on certain properties of sets of formulas. Smullyan made these properties explicit giving a notion of abstract consistency [62] (see also [61]). Given some notion of abstract consistency and some abstractly consistent set, it can be extended to be an evident set (a Hintikka set in Smullyan's terminology; a

model set in Hintikka's terminology). This is what we have proven in our Extension Lemma (Lemma 6.4.3). Combining this with the Model Existence Theorem we conclude completeness.

In the context of simple type theory the completeness result (relative to standard models) can be extended to fragments that extend first-order logic by allowing λ-abstractions and embedded formulas [15]. The construction of a model is significantly different in this extended first-order (EFO) fragment.

7 Tableaux with Equality

In this chapter we give the final tableau rules defining the refutation calculus TaS which is complete for STT. The final rules allow us to reason about equations and disequations. They also allow us to reason about embedded formulas via disequations.

7.1 Functional Equality

We must give rules for equations and disequations at function type. The rules are similar to the quantifier rules.

$$\mathcal{T}_{\mathbf{FQ}}\ \frac{s =_{\sigma\tau} t}{[su] = [tu]} \qquad \mathcal{T}_{\mathbf{FE}}\ \frac{s \neq_{\sigma\tau} t}{[sx] \neq [tx]}\ x \text{ fresh}$$

As a simple example, we refute the branch with $f = g$ and $g \neq f$ where $f, g : \iota o$. The rule $\mathcal{T}_{\text{FE}}$ applied to $g \neq f$ using x (which is fresh) allows us to add $gx \not\equiv fx$ to the branch. We apply $\mathcal{T}_{\text{FQ}}$ to $f = g$ with x and add $fx \equiv gx$ to the branch. The refutation is completed using $\mathcal{T}_{\text{BE}}$ and $\mathcal{T}_{\text{BQ}}$.

$$\begin{array}{c|c|c} f = g,\ g \neq f,\ gx \not\equiv fx,\ fx \equiv gx & & \\ gx,\ \neg fx & fx,\ \neg gx & \\ fx \mid \neg gx & gx \mid \neg fx & \end{array}$$

Using $\mathcal{T}_{\text{FQ}}$ we can prove the following version of Cantor's Theorem:

$$\neg\exists f \forall g \exists x. fx = g$$

where $f : \sigma\sigma o$, $g : \sigma o$ and $x : \sigma$ are variables. We begin the refutation using the $\mathcal{T}_\exists$ and $\mathcal{T}_\forall$ with the instantiation term $\lambda x.\neg fxx$. This results in the branch

$$\begin{array}{c} \exists f \forall g \exists x. fx = g \\ \forall g \exists x. fx = g \\ \exists x. fx = \lambda x.\neg fxx \\ fx = \lambda x.\neg fxx \end{array}$$

The rule $\mathcal{T}_{\text{FQ}}$ applied to $fx = \lambda x.\neg fxx$ using x allows us to add $fxx \equiv \neg fxx$ to the branch. The refutation is finished using $\mathcal{T}_{\text{BQ}}$ as usual.

$$\begin{array}{c}
\exists f \forall g \exists x.fx = g \\
\forall g \exists x.fx = g \\
\exists x.fx = \lambda x.\neg fxx \\
fx = \lambda x.\neg fxx \\
fxx \equiv \neg fxx \\
\hline
fxx,\ \neg fxx \quad \Big| \quad \begin{array}{c} \neg fxx \\ \neg\neg fxx \end{array}
\end{array}$$

7.2 Mating, Decomposition and Confrontation

Although we have now given rules for all of the logical constants, we do not yet have a complete system. Consider the following example.

Let $p : oo$, $x : o$ and $y : o$ be variables. Consider the branch A with the formulas x, y, px and $\neg py$. The set A is unsatisfiable. Suppose $\mathcal{I}$ were a model of A. We must have $\mathcal{I}x = 1$ and $\mathcal{I}y = 1$. Hence $\mathcal{I}x = \mathcal{I}y$. Thus $\hat{\mathcal{I}}(px) = \hat{\mathcal{I}}(py)$, which is impossible since $\hat{\mathcal{I}}(px) = 1$ and $\hat{\mathcal{I}}(py) = 0$.

We have not yet given any rule which applies to this A. The rule we give will be, in some ways, a generalization of the rule $\mathcal{T}_{\neg}$. Consider the formulas px and $\neg py$. If x and y were the same variable, then $\mathcal{T}_{\neg}$ would apply and close the branch. In other words, we can conclude from px and $\neg py$ that x and y must be different. We need a rule which will allow us to add the disequation $x \neq y$ to the branch. If $x \neq y$ is on the branch, then the rest of the refutation is clear:

$$\begin{array}{c}
x,\ y,\ px,\ \neg py,\ x \not\equiv y \\
\hline
\neg y \quad \Big| \quad \neg x
\end{array}$$

In general a **mating rule** $\mathcal{T}_{\text{MAT}}$ is given by the scheme

$$\mathcal{T}_{\text{MAT}}\ \frac{xs_1 \ldots s_n\,,\ \neg xt_1 \ldots t_n}{s_1 \neq t_1 \mid \cdots \mid s_n \neq t_n}$$

Here we have two formulas $xs_1 \cdots s_n$ and $\neg xt_1 \cdots t_n$ on the branch where $x : \sigma_1 \cdots \sigma_n o$ is a variable. In such a case we know there must be some $i \in \{1, \cdots, n\}$ such that s_i and t_i must be different. For each i we create a branch with the disequation $s_i \neq t_i$.

Note that we include the possibility that $n = 0$ as a mating rule. In such a case we have x and $\neg x$ on the branch and no alternatives. Hence a mating rule with $n = 0$ is a rule that closes a branch. The mating rule when $n = 0$ is a special case of $\mathcal{T}_{\neg}$. While we will keep $\mathcal{T}_{\neg}$ in our calculus, it turns out that one only needs

the special case where there is a variable x such that x and $\neg x$ is on the branch. That is, $\mathcal{T}_\neg$ can be replaced by mating (in the presence of all the other rules of TaS.

Exercise 7.2.1 Let $a, b : o$ and $p : oo$ be given. Refute the following branches using the mating rule and TaQ rules.

- $a \equiv b,\ pa,\ \neg pb$
- $pa,\ pb,\ \neg p(a \wedge b)$
- $p(a \wedge b),\ \neg p(b \wedge a)$

Exercise 7.2.2 Let $f : oo$ and $x : o$ be variables. Prove $f(f(fx)) \equiv fx$ by refuting its negation using only $\mathcal{T}_{\text{MAT}}$ (mating) and $\mathcal{T}_{\text{BQ}}$.

Until now, we have no rules for equations and disequations at a sort. In particular, we cannot refute $x \neq x$ when $x : \alpha$. A **decomposition rule** is generated by the following scheme

$$\mathcal{T}_{\text{DEC}}\ \frac{xs_1 \ldots s_n \neq_\alpha xt_1 \ldots t_n}{s_1 \neq t_1 \mid \cdots \mid s_n \neq t_n}$$

Here we have a disequation between $xs_1 \cdots s_n$ and $xt_1 \cdots t_n$ on the branch where $x : \sigma_1 \cdots \sigma_n \alpha$. As in the mating rule we create a branch with the disequation $s_i \neq t_i$ for each $i \in \{1, \ldots, n\}$. If $n = 0$, this means the branch is closed. In particular, any branch with a formula $x \neq x$ for a variable $x : \alpha$ is closed.

As a nontrivial example of the decomposition rule, consider a branch with $x,\ y,\ fx \neq fy$ where $x, y : o$ and $f : o\alpha$ are variables. Using $\mathcal{T}_{\text{DEC}}$ we add $x \not\equiv y$ to the branch. The refutation is finished with $\mathcal{T}_{\text{BE}}$.

$$\begin{array}{c} x,\ y,\ fx \neq fy,\ x \not\equiv y \\ \hline \neg y \quad \Big| \quad \neg x \end{array}$$

Exercise 7.2.3 Using only decomposition and $\mathcal{T}_{\text{FE}}$, refute $f \neq f$ where $f : \iota(\iota\iota)\iota$ is a variable.

Our final rules are **confrontation** which applies when there is both an equation and a disequation of a sort α on the branch. The schema for **confrontation** is

$$\mathcal{T}_{\text{CON}}\ \frac{s =_\alpha t\,,\ u \neq_\alpha v}{s \neq u\,,\ t \neq u \mid s \neq v\,,\ t \neq v}$$

As a single example of confrontation consider a branch with $x = y$ and $y \neq x$ with $x, y : \alpha$. Confronting $x = y$ against $y \neq x$ gives

$$\begin{array}{c} x = y,\ y \neq x \\ \hline x \neq y,\ y \neq y \quad \Big| \quad x \neq x \end{array}$$

$$\mathcal{T}_{\bot}\ \frac{\bot}{}\qquad \mathcal{T}_{\neg\top}\ \frac{\neg\top}{}\qquad \mathcal{T}_{\neg}\ \frac{\neg s, s}{}\qquad \mathcal{T}_{\neg\neg}\ \frac{\neg\neg s}{s}\qquad \mathcal{T}_{\wedge}\ \frac{s \wedge t}{s, t}\qquad \mathcal{T}_{\neg\wedge}\ \frac{\neg(s \wedge t)}{\neg s \mid \neg t}$$

$$\mathcal{T}_{\vee}\ \frac{s \vee t}{s \mid t}\qquad \mathcal{T}_{\neg\vee}\ \frac{\neg(s \vee t)}{\neg s, \neg t}\qquad \mathcal{T}_{\rightarrow}\ \frac{s \rightarrow t}{t \mid \neg s}\qquad \mathcal{T}_{\neg\rightarrow}\ \frac{\neg(s \rightarrow t)}{s, \neg t}\qquad \mathcal{T}_{\mathbf{BQ}}\ \frac{s \equiv t}{s, t \mid \neg s, \neg t}$$

$$\mathcal{T}_{\mathbf{BE}}\ \frac{\neg(s \equiv t)}{s, \neg t \mid t, \neg s}\qquad \mathcal{T}_{\forall}\ \frac{\forall_\sigma s}{[st]}\ t : \sigma\qquad \mathcal{T}_{\exists}\ \frac{\exists_\sigma s}{[sx]}\ x : \sigma \text{ fresh}$$

$$\mathcal{T}_{\neg\forall}\ \frac{\neg\forall_\sigma s}{\neg[sx]}\ x : \sigma \text{ fresh}\qquad \mathcal{T}_{\neg\exists}\ \frac{\neg\exists_\sigma s}{\neg[st]}\ t : \sigma\qquad \mathcal{T}_{\mathbf{FQ}}\ \frac{s =_{\sigma\tau} t}{[su] = [tu]}$$

$$\mathcal{T}_{\mathbf{FE}}\ \frac{s \neq_{\sigma\tau} t}{[sx] \neq [tx]}\ x : \sigma \text{ fresh}\qquad \mathcal{T}_{\mathbf{MAT}}\ \frac{xs_1 \ldots s_n\,,\ \neg x t_1 \ldots t_n}{s_1 \neq t_1 \mid \cdots \mid s_n \neq t_n}$$

$$\mathcal{T}_{\mathbf{DEC}}\ \frac{xs_1 \ldots s_n \neq_\alpha x t_1 \ldots t_n}{s_1 \neq t_1 \mid \cdots \mid s_n \neq t_n}\qquad \mathcal{T}_{\mathbf{CON}}\ \frac{s =_\alpha t\,,\ u \neq_\alpha v}{s \neq u\,,\ t \neq u \mid s \neq v\,,\ t \neq v}$$

Figure 7.1: Tableau system TaS

Both branches are closed by decomposition (with $n = 0$).

Exercise 7.2.4 Using only mating, decomposition and confrontation, refute the branch with the formulas $c = d$, qc and $\neg qd$ where $c, d : \iota$ and $q : \iota o$ are variables.

7.3 The Full Tableau System

Figure 7.1 gives all the rules of our tableau system TaS.

The following example uses mating, confrontation and decomposition.

Example 7.3.1 Let $D : (\iota o)\iota$, $x, y, z : \iota$ and $p : \iota o$ be variables. We will refute the following branch.

$$\forall z.D(\lambda y.y = z) = z$$
$$px$$
$$\forall y.py \rightarrow y = x$$
$$\neg p(Dp)$$

Mating ($\mathcal{T}_{\text{MAT}}$) px with $\neg p(Dp)$ gives

$$\begin{array}{c} \forall z.D(\lambda y.y = z) = z,\ px \\ \forall y.py \to y = x,\ \neg p(Dp),\ x \neq Dp \end{array}$$

We apply $\mathcal{T}_{\forall}$ to $\forall z.D(\lambda y.y = z) = z$ with x and obtain

$$\begin{array}{c} \forall z.D(\lambda y.y = z) = z,\ px \\ \forall y.py \to y = x,\ \neg p(Dp),\ x \neq Dp \\ D(\lambda y.y = x) = x \end{array}$$

Applying $\mathcal{T}_{\text{CON}}$ (confrontation) to $D(\lambda y.y = x) = x$ and $x \neq Dp$ yields

$$\begin{array}{c} \forall z.D(\lambda y.y = z) = z,\ px \\ \forall y.py \to y = x,\ \neg p(Dp),\ x \neq Dp \\ D(\lambda y.y = x) = x \\ \hline D(\lambda y.y = x) \neq x,\ x \neq x \mid D(\lambda y.y = x) \neq Dp \end{array}$$

Decomposing ($\mathcal{T}_{\text{DEC}}$) $D(\lambda y.y = x) \neq Dp$ yields

$$\begin{array}{c} \forall z.D(\lambda y.y = z) = z,\ px \\ \forall y.py \to y = x,\ \neg p(Dp),\ x \neq Dp \\ D(\lambda y.y = x) = x \\ \hline D(\lambda y.y = x) \neq x,\ x \neq x \mid \begin{array}{c} D(\lambda y.y = x) \neq Dp \\ (\lambda y.y = x) \neq p \end{array} \end{array}$$

We apply $\mathcal{T}_{\text{FE}}$ to $(\lambda y.y = x) \neq p$ with y and obtain

$$\begin{array}{c} \forall z.D(\lambda y.y = z) = z,\ px \\ \forall y.py \to y = x,\ \neg p(Dp),\ x \neq Dp \\ D(\lambda y.y = x) = x \\ \hline D(\lambda y.y = x) \neq x,\ x \neq x \mid \begin{array}{c} D(\lambda y.y = x) \neq Dp \\ (\lambda y.y = x) \neq p \\ y = x \not\equiv py \end{array} \end{array}$$

Applying $\mathcal{T}_{\text{BE}}$ to $y = x \not\equiv py$ we obtain

$$\begin{array}{c} \forall z.D(\lambda y.y = z) = z,\ px \\ \forall y.py \to y = x,\ \neg p(Dp),\ x \neq Dp \\ D(\lambda y.y = x) = x \\ \hline \begin{array}{c} D(\lambda y.y = x) \neq x \\ x \neq x \end{array} \mid \begin{array}{c} D(\lambda y.y = x) \neq Dp \\ (\lambda y.y = x) \neq p \\ y = x \not\equiv py \\ \hline y = x \mid py \\ \neg py \mid y \neq x \end{array} \end{array}$$

Mating $\neg py$ with px gives

$$\dfrac{\begin{array}{c}\forall z.D(\lambda y.y = z) = z,\ px\\ \forall y.py \to y = x,\ \neg p(Dp),\ x \neq Dp\\ D(\lambda y.y = x) = x\end{array}}{\begin{array}{c|c}D(\lambda y.y = x) \neq x\\ x \neq x\end{array}\quad\begin{array}{c}D(\lambda y.y = x) \neq Dp\\ (\lambda y.y = x) \neq p\\ y = x \not\equiv py\end{array}\quad\dfrac{\begin{array}{c}y = x\\ \neg py\\ y \neq x\end{array}\quad\begin{array}{c}py\\ y \neq x\end{array}}{}}$$

We apply $\mathcal{T}_\forall$ to $\forall y.py \to y = x$ with y followed by $\mathcal{T}_\to$ to finish the refutation.

$$\dfrac{\begin{array}{c}\forall z.D(\lambda y.y = z) = z,\ px\\ \forall y.py \to y = x,\ \neg p(Dp),\ x \neq Dp\\ D(\lambda y.y = x) = x\end{array}}{\begin{array}{c|c}D(\lambda y.y = x) \neq x\\ x \neq x\end{array}\quad\begin{array}{c}D(\lambda y.y = x) \neq Dp\\ (\lambda y.y = x) \neq p\\ y = x \not\equiv py\end{array}\quad\dfrac{\begin{array}{c}y = x\\ \neg py\\ y \neq x\end{array}\quad\dfrac{\begin{array}{c}py,\ y \neq x\\ py \to y = x\end{array}}{\begin{array}{c|c}y = x & \neg py\end{array}}}{}}$$

Exercise 7.3.2 Let $h : o\iota$ be given. Prove $h(h\top = h\bot) = h\bot$ is valid by giving a TaS-refutation of its negation.

Exercise 7.3.3 Use TaS to refute the branch

$$x,\ y,\ p(fx),\ \neg p(fy)$$

where $x, y : o$, $f : o\iota$ and $p : \iota o$ are variables.

Exercise 7.3.4 Let $p : \iota o$, $C, D : (\iota o)\iota$ and $x, y : \iota$ be variables. Use TaS to refute the branch

$$\forall p.(\exists x.px) \to p(Cp)$$
$$\neg\exists D\forall p.(\exists x.px \wedge \forall y.py \to x = y) \to p(Dp)$$

Exercise 7.3.5 Let $f : \iota\iota$, $r : \iota\iota o$ and $x, y, z : \iota$ be variables. Use TaS to refute the branch

$$\forall r.(\forall x\exists y.rxy) \to \exists f\forall x.rx(fx)$$
$$\neg\forall r.(\forall x\exists y.rxy \wedge \forall z.rxz \to y = z) \to \exists f\forall x.rx(fx)$$

Exercise 7.3.6 Let $C : (\iota o)\iota$, $p : \iota o$, $r : \iota\iota o$, $f : \iota\iota$ and $x, y : \iota$ be variables. Use TaS to refute the branch

$$\exists C \forall p.(\exists x.px) \to p(Cp)$$
$$\neg \forall r.(\forall x \exists y.rxy) \to \exists f \forall x.rx(fx)$$

Exercise 7.3.7 Prove with TaS that $px \equiv \forall y.\, y{=}x \to py$ with $x : \iota$ is valid.

8 Tableaux Examples

In this chapter we give a number of examples showing how to represent concepts in STT and prove things about them using tableau. We will also give an extended tableau system TaS^+ which makes proving more practical.

8.1 Example: A Formal Proof by Induction

Here we give a formal proof by induction on the natural numbers. First we give the informal proof. For every natural number n, let Sn denote the successor of n (i.e., $n+1$). The following is the first inductive proof given in Landau [48].

Proposition 8.1.1 For every natural number n, $Sn \neq n$.

Proof The proof is by induction on n.

Base case: $S0 \neq 0$ since no successor of a natural number is 0.

Inductive case: Assume the inductive hypothesis: $Sn \neq n$. We will prove $S(Sn) \neq (Sn)$. Assume $S(Sn) = Sn$. Since the successor function is injective, $Sn = n$, contradicting the inductive hypothesis. ∎

We now repeat this informal proof in the formal setting of simple type theory.

Example 8.1.2 Recall the formula **chain** from Chapter 4. This was the conjunction of the three formulas

1. $\forall x.\ Sx{=}o$
2. $\forall xy.\ Sx{=}Sy \to x{=}y$
3. $\forall p.\ po \to (\forall x.\ px \to p(Sx)) \to (\forall x.px)$

where N is a sort and $o : N$, $S : NN$, $x, y : N$ and $p : No$ are variables. (These formulas come from Peano's axioms for the natural numbers in 1889.) We prove the desired result by including the three formulas of **chain** and the negated formula $\neg\forall x.Sx \neq x$. That is, we refute the branch

$$\forall x.Sx \neq o$$
$$\forall xy.Sx = Sy \to x = y$$
$$\forall p.po \wedge (\forall x.px \to p(Sx)) \to \forall x.px$$
$$\neg\forall x.Sx \neq x$$

The fact that we will prove the result by induction corresponds to applying the rule $\mathcal{T}_\forall$ to $\forall p.po \wedge (\forall x.px \to p(Sx)) \to \forall x.px$ using the instantiation term $\lambda x.Sx \neq x$. The result is the branch.

$$\begin{array}{c}
\forall x.Sx \neq o \\
\forall xy.Sx = Sy \to x = y \\
\forall p.po \wedge (\forall x.px \to p(Sx)) \to \forall x.px \\
\neg\forall x.Sx \neq x \\
So \neq o \wedge (\forall x.Sx \neq x \to S(Sx) \neq Sx) \to \forall x.Sx \neq x
\end{array}$$

We apply $\mathcal{T}_\to$ to the new formula. Note that the left branch is now closed.

$$\begin{array}{c|c}
\forall x.Sx \neq o & \\
\forall xy.Sx = Sy \to x = y & \\
\forall p.po \wedge (\forall x.px \to p(Sx)) \to \forall x.px & \\
\neg\forall x.Sx \neq x & \\
So \neq o \wedge (\forall x.Sx \neq x \to S(Sx) \neq Sx) \to \forall x.Sx \neq x & \\
\forall x.Sx \neq x & \neg(So \neq o \wedge \forall x.Sx \neq x \to S(Sx) \neq Sx)
\end{array}$$

Applying $\mathcal{T}_{\neg\wedge}$ gives us two open branch corresponding to the base case and the inductive case.

$$\begin{array}{c|c|c}
\forall x.Sx \neq o & & \\
\forall xy.Sx = Sy \to x = y & & \\
\forall p.po \wedge (\forall x.px \to p(Sx)) \to \forall x.px & & \\
\neg\forall x.Sx \neq x & & \\
So \neq o \wedge (\forall x.Sx \neq x \to S(Sx) \neq Sx) \to \forall x.Sx \neq x & & \\
\forall x.Sx \neq x & \neg(So \neq o \wedge \forall x.Sx \neq x \to S(Sx) \neq Sx) & \\
 & \neg So \neq o & \neg\forall x.Sx \neq x \to S(Sx) \neq Sx
\end{array}$$

The rule $\mathcal{T}_\forall$ applied to $\forall x.Sx \neq o$ using o results in a closed branch. This corresponds to the base case.

$$\begin{array}{c|c|c}
\forall x.Sx \neq o & & \\
\forall xy.Sx = Sy \to x = y & & \\
\forall p.po \wedge (\forall x.px \to p(Sx)) \to \forall x.px & & \\
\neg\forall x.Sx \neq x & & \\
So \neq o \wedge (\forall x.Sx \neq x \to S(Sx) \neq Sx) \to \forall x.Sx \neq x & & \\
\forall x.Sx \neq x & \neg(So \neq o \wedge \forall x.Sx \neq x \to S(Sx) \neq Sx) & \\
 & \neg So \neq o & \neg\forall x.Sx \neq x \to S(Sx) \neq Sx \\
 & So \neq o &
\end{array}$$

We next prove the inductive case. Let y be a natural number for which we assume $Sy \neq y$. We must prove $S(Sy) \neq Sy$. Technically, we apply $\mathcal{T}_{\neg\forall}$ to $\neg\forall x.Sx \neq$

$x \to S(Sx) \neq Sx$ followed by $\mathcal{T}_{\neg\to}$.

$$
\begin{array}{c}
\forall x.Sx \neq o \\
\forall xy.Sx = Sy \to x = y \\
\forall p.po \wedge (\forall x.px \to p(Sx)) \to \forall x.px \\
\neg\forall x.Sx \neq x \\
So \neq o \wedge (\forall x.Sx \neq x \to S(Sx) \neq Sx) \to \forall x.Sx \neq x \\
\left|\begin{array}{c}
\neg(So \neq o \wedge \forall x.Sx \neq x \to S(Sx) \neq Sx) \\
\left|\begin{array}{c}
\neg\forall x.Sx \neq x \to S(Sx) \neq Sx \\
\neg(Sy \neq y \to S(Sy) \neq Sy) \\
Sy \neq y \\
\neg S(Sy) \neq Sy
\end{array}\right. \\
\begin{array}{c}
\forall x.Sx \neq x \\
\neg So \neq o \\
So \neq o
\end{array}
\end{array}\right.
\end{array}
$$

The rule $\mathcal{T}_\forall$ applied to $\forall xy.Sx = Sy \to x = y$ (injectivity of successor) using Sy and then y gives

$$
\begin{array}{c}
\forall x.Sx \neq o \\
\forall xy.Sx = Sy \to x = y \\
\forall p.po \wedge (\forall x.px \to p(Sx)) \to \forall x.px \\
\neg\forall x.Sx \neq x \\
So \neq o \wedge (\forall x.Sx \neq x \to S(Sx) \neq Sx) \to \forall x.Sx \neq x \\
\left|\begin{array}{c}
\neg(So \neq o \wedge \forall x.Sx \neq x \to S(Sx) \neq Sx) \\
\left|\begin{array}{c}
\neg\forall x.Sx \neq x \to S(Sx) \neq Sx \\
\neg(Sy \neq y \to S(Sy) \neq Sy) \\
Sy \neq y \\
\neg S(Sy) \neq Sy \\
\forall y'.S(Sy) = Sy' \to Sy = y' \\
S(Sy) = Sy \to Sy = y
\end{array}\right. \\
\begin{array}{c}
\forall x.Sx \neq x \\
\neg So \neq o \\
So \neq o
\end{array}
\end{array}\right.
\end{array}
$$

Applying $\mathcal{T}_\to$ to we finish the tableau refutation.

$$
\begin{array}{c}
\forall x.Sx \neq o \\
\forall xy.Sx = Sy \to x = y \\
\forall p.po \wedge (\forall x.px \to p(Sx)) \to \forall x.px \\
\neg\forall x.Sx \neq x \\
So \neq o \wedge (\forall x.Sx \neq x \to S(Sx) \neq Sx) \to \forall x.Sx \neq x \\
\left|\begin{array}{c}
\neg(So \neq o \wedge \forall x.Sx \neq x \to S(Sx) \neq Sx) \\
\left|\begin{array}{c}
\neg\forall x.Sx \neq x \to S(Sx) \neq Sx \\
\neg(Sy \neq y \to S(Sy) \neq Sy),\ Sy \neq y \\
\neg S(Sy) \neq Sy \\
\forall y'.S(Sy) = Sy' \to Sy = y' \\
S(Sy) = Sy \to Sy = y \\
Sy = y \quad \left|\ S(Sy) \neq Sy\right.
\end{array}\right. \\
\begin{array}{c}
\forall x.Sx \neq x \\
\neg So \neq o \\
So \neq o
\end{array}
\end{array}\right.
\end{array}
$$

The attentive reader will note that although there are many identities $=_N$ we have not used any tableau rules that explicitly reason about identities. The same refutation would work if $=_N$ were uniformly replaced by a variable $r : NNo$. □

8.2 Skolem's Law and the Axiom of Choice

For types σ and τ let $\mathsf{Skolem}_{\sigma,\tau}$ be the formula

$$\forall r.(\forall x \exists y.rxy) \to \exists f \forall x.rx(fx)$$

where $r : \sigma\tau o$, $f : \sigma\tau$, $x : \sigma$ and $y : \tau$ are variables. When $\mathsf{Skolem}_{\sigma,\tau}$ holds we can commute universal and existential quantifiers. Skolem [59] used this technique to reduce satisfiability of first-order formulas to satisfiability of first-order formulas of the form $\forall x_1 \cdots x_n.s$ where s is quantifier-free (see § 9.9).

For every type σ let Choice_σ be the formula

$$\exists C \forall p.(\exists x.px) \to p(Cp)$$

where $C : (\sigma o)\sigma$, $p : \sigma o$ and $x : \sigma$ are variables. Choice_σ formulates the axiom of choice in simple type theory (see § 10.1). It turns out that a logical interpretation satisfies $\mathsf{Skolem}_{\sigma,\tau}$ for all types σ and τ iff it satisfies Choice_σ for every type σ. We prove this in two steps using tableau. For any types σ and τ we will show $\mathsf{Skolem}_{\sigma,\tau}$ holds if Choice_τ holds. For any type σ we will show Choice_σ holds if $\mathsf{Skolem}_{\sigma o,\sigma}$ holds.

Choice_σ and $\mathsf{Skolem}_{\sigma,\tau}$ are weakly valid but not valid (see § 10.1 and § 10.2).

Example 8.2.1 Let σ and τ be types. We refute the branch containing Choice_τ and the negation of $\mathsf{Skolem}_{\sigma,\tau}$. After applying $\mathcal{T}_{\neg\forall}$, $\mathcal{T}_{\neg\to}$ and $\mathcal{T}_\exists$ we have

$$\begin{array}{c}
\exists C \forall p.(\exists z.pz) \to p(Cp) \\
\neg\forall r.(\forall x \exists y.rxy) \to \exists f \forall x.rx(fx) \\
\neg((\forall x \exists y.rxy) \to \exists f \forall x.rx(fx)) \\
\forall x \exists y.rxy \\
\neg\exists f \forall x.rx(fx) \\
\forall p.(\exists z.pz) \to p(Cp)
\end{array}$$

The rule $\mathcal{T}_{\neg\exists}$ applied to $\neg\exists f \forall x.rx(fx)$ using $\lambda x.C(rx)$ gives

$$\begin{array}{c}
\exists C \forall p.(\exists z.pz) \to p(Cp) \\
\neg\forall r.(\forall x \exists y.rxy) \to \exists f \forall x.rx(fx) \\
\neg((\forall x \exists y.rxy) \to \exists f \forall x.rx(fx)) \\
\forall x \exists y.rxy \\
\neg\exists f \forall x.rx(fx) \\
\forall p.(\exists z.pz) \to p(Cp) \\
\neg\forall x.rx(C(rx))
\end{array}$$

Applying $\mathcal{T}_{\neg\forall}$ with x and $\mathcal{T}_{\forall}$ with rx we obtain

$$\begin{array}{c}
\exists C \forall p.(\exists z.pz) \to p(Cp) \\
\neg\forall r.(\forall x \exists y.rxy) \to \exists f \forall x.rx(fx) \\
\neg((\forall x \exists y.rxy) \to \exists f \forall x.rx(fx)) \\
\forall x \exists y.rxy \\
\neg\exists f \forall x.rx(fx) \\
\forall p.(\exists z.pz) \to p(Cp) \\
\neg\forall x.rx(C(rx)),\ \neg rx(C(rx)) \\
(\exists z.rxz) \to rx(C(rx))
\end{array}$$

The rest of the refutation is unsurprising.

$$\begin{array}{c}
\exists C \forall p.(\exists z.pz) \to p(Cp) \\
\neg\forall r.(\forall x \exists y.rxy) \to \exists f \forall x.rx(fx) \\
\neg((\forall x \exists y.rxy) \to \exists f \forall x.rx(fx)) \\
\forall x \exists y.rxy \\
\neg\exists f \forall x.rx(fx) \\
\forall p.(\exists z.pz) \to p(Cp) \\
\neg\forall x.rx(C(rx)),\ \neg rx(C(rx)) \\
(\exists z.rxz) \to rx(C(rx))
\end{array}$$

$$\begin{array}{c|c}
rx(C(rx)) & \begin{array}{c}\neg\exists z.rxz \\ \exists y.rxy,\ rxy,\ \neg rxy\end{array}
\end{array}$$

For the other direction we will make use of an extra (sound) rule in the schema $\mathcal{T}_{\neg\alpha}$:

$$\mathcal{T}_{\neg\alpha}\ \frac{\neg s, s'}{s \sim_\alpha s'}$$

We leave it to the reader to find a TaS-refutation (without making use of the extra rule).

Example 8.2.2 Let σ be a type. We refute the branch containing $\mathsf{Skolem}_{\sigma o,\sigma}$ and the negation of Choice_σ.

$$\begin{array}{c}
\forall r.(\forall x \exists y.rxy) \to \exists f \forall x.rx(fx) \\
\neg\exists C \forall p.(\exists z.pz) \to p(Cp)
\end{array}$$

The rule $\mathcal{T}_{\forall}$ applied using $\lambda xy.(\exists z.xz) \to xy$ gives

$$\begin{array}{c}
\forall r.(\forall x \exists y.rxy) \to \exists f \forall x.rx(fx) \\
\neg\exists C \forall p.(\exists z.pz) \to p(Cp) \\
(\forall x \exists y.(\exists z.xz) \to xy) \to \exists f \forall x.(\exists z.xz) \to x(fx)
\end{array}$$

We apply $\mathcal{T}_{\to}$ to $(\forall x \exists y.(\exists z.xz) \to xy) \to \exists f \forall x.(\exists z.xz) \to x(fx)$ and obtain

$$\dfrac{\begin{array}{c}\forall r.(\forall x \exists y.rxy) \to \exists f \forall x.rx(fx) \\ \neg \exists C \forall p.(\exists z.pz) \to p(Cp) \\ (\forall x \exists y.(\exists z.xz) \to xy) \to \exists f \forall x.(\exists z.xz) \to x(fx)\end{array}}{\exists f \forall x.(\exists z.xz) \to x(fx) \mid \neg \forall x \exists y.(\exists z.xz) \to xy}$$

Note that the left branch is closed by $\mathcal{T}_{\neg\alpha}$. The remainder of the refutation is similar to the Drinker's Paradox (Example 6.1.3).

$$\dfrac{\begin{array}{c}\forall r.(\forall x \exists y.rxy) \to \exists f \forall x.rx(fx) \\ \neg \exists C \forall p.(\exists z.pz) \to p(Cp) \\ (\forall x \exists y.(\exists z.xz) \to xy) \to \exists f \forall x.(\exists z.xz) \to x(fx)\end{array}}{\begin{array}{c|c} & \neg \forall x \exists y.(\exists z.xz) \to xy \\ & \neg \exists y.(\exists z.xz) \to xy \\ \exists f \forall x.(\exists z.xz) \to x(fx) & \neg((\exists z.xz) \to xy) \\ & \exists z.xz,\ \neg xy,\ xz \\ & \neg((\exists z.xz) \to xz),\ \neg xz \end{array}}$$

8.3 An Extended Tableau System

The tableau system TaS is complete for STT. For now we will state this result and leave the proof for a future chapter.

Theorem 8.3.1 TaS is complete with respect to the full fragment of STT.

Of course, TaS is also refutation sound. Consequently, we have the following results.

Corollary 8.3.2 For any branch A, A is TaS-refutable iff A is unsatisfiable.

Corollary 8.3.3 Let $\mathcal{T}$ be any tableau system such that every rule of TaS is a rule of $\mathcal{T}$ and every rule of $\mathcal{T}$ is sound. For any branch A the following are equivalent.

1. A is $\mathcal{T}$-refutable.
2. Some subset A' of A is $\mathcal{T}$-refutable.
3. A is unsatisfiable.
4. A is TaS-refutable.

Corollary 8.3.3 motivates the definition of an extended tableau system TaS^+ for which proving is easier. We define TaS^+ by adding four sets of rules to the

rules of TaS. First the schema $\mathcal{T}_{\neg\alpha}$ is a generalization of $\mathcal{T}_{\neg}$ that allows us to close a branch when there are complementary formulas up to $\sim_\alpha$.

$$\mathcal{T}_{\neg\alpha}\ \frac{\neg s, s'}{s \sim_\alpha s'}$$

Next the schema $\mathcal{T}_{\neq\alpha}$ allows us to close a branch when a formula conflicts with reflexivity of equality (up to $\sim_\alpha$).

$$\mathcal{T}_{\neq\alpha}\ \frac{s \neq_\sigma s'}{s \sim_\alpha s'}$$

The schema $\mathcal{T}_{\mathrm{sym}}$ allows us to close a branch when two formulas are complementary equations up to symmetry of equality and $\sim_\alpha$.

$$\mathcal{T}_{\mathbf{sym}}\ \frac{s =_\sigma t, t' \neq_\sigma s'}{s \sim_\alpha s', t \sim_\alpha t'}$$

Finally, a cut rule $\mathcal{T}_{\mathrm{cut}}$ allows us to consider two cases s or $\neg s$ (for a formula s) at any point during a proof.

$$\mathcal{T}_{\mathbf{cut}}\ \frac{}{s \mid \neg s}$$

Now TaS^+ is the set of rules given by TaS, $\mathcal{T}_{\neg\alpha}$, $\mathcal{T}_{\neq\alpha}$, $\mathcal{T}_{\mathrm{sym}}$ and $\mathcal{T}_{\mathrm{cut}}$.

From Corollary 8.3.3 we have the following results.

Proposition 8.3.4 For any branch A the following are equivalent.

1. A is TaS^+-refutable.
2. A is unsatisfiable.
3. A is TaS-refutable.

Proposition 8.3.5 (Weakening) A branch A is TaS^+-refutable iff some subset of A is TaS^+-refutable.

In the remainder of this chapter we will use the word **refutable** to mean TaS^+-refutable and give examples of TaS^+-refutattions. By weakening (Proposition 8.3.5) we can conclude that a branch A is refutable whenever a subset A' has already been refuted. Consequently, we make the convention that a branch can be closed by a reference to an earlier refutation of a subset of the branch.

The following example shows how the new rules and conventions can simplify proving.

Example 8.3.6 Let $a, b : o$ and $q : o \cdots oo$ be variables where q expects n arguments. We can refute the branch with a, b, $qa \cdots a$ and $\neg qb \cdots b$ using only the rules from TaS. The tableau has $2n$ branches. If we first refute the branch with a, b and $a \neq b$ and then use this (via weakening) we only need to show n branches. Finally, if we use weakening and use $\mathcal{T}_{\text{cut}}$ with Leibniz equality of a and b, then we can refute the branch showing only 3 branches (independent of n). □

8.4 Transitive Closure

Suppose R is a binary relation on a set X. We can define the transitive closure R^+ inductively as follows:

1. If $(x, y) \in R$, then $(x, y) \in R^+$.
2. If $(x, y) \in R^+$ and $(y, z) \in R^+$, then $(x, z) \in R^+$.

We often describe R^+ as the least transitive relation which contains R. In this section we will show how to define the transitive closure of a relation in STT and prove that it is indeed the transitive closure. We will furthermore prove an inversion principle.

In simple type theory we represent binary relations using a type of the form $\sigma\tau o$. For any σ and τ, let $\subseteq_{\sigma,\tau}$ be the term

$$\lambda r r'.\forall x y.rxy \to r'xy$$

for some variables $r, r' : \sigma\tau o$, $x : \sigma$ and $y : \tau$. When the types σ and τ are clear in context, we write simply $\subseteq$. As usual, we use infix notation and write $s \subseteq t$ for $\subseteq$ applied to s and t.

To represent transitivity we must assume σ and τ are the same type. Given variables $r : \sigma\sigma o$ and $x, y, z : \sigma$ transitivity can be represented as

$$\lambda r.\forall x y z.rxy \to ryz \to rxz$$

Let trans_σ (or simply trans) be this term.

Given these conventions and variables $r, r' : \sigma\sigma o$ and $x, y : \sigma$ we can represent transitive closure as

$$\lambda r x y.\forall r'.r \subseteq r' \to \mathsf{trans}\, r' \to r'xy$$

For any term $s : \sigma\sigma o$ let s^+ stand for the normal term

$$[(\lambda r x y.\forall r'.r \subseteq r' \to \mathsf{trans}\, r' \to r'xy)s]$$

Now we consider a relation over the specific sort ι. Let $r, r', s : \iota\iota o$ and $x, y, z : \iota$ be variables. Note that r^+ is the term

$$\lambda xy.\forall r'.[r \subseteq r'] \to [\mathsf{trans}\, r'] \to r'xy$$

We will prove r^+ contains r and r^+ is transitive. Furthermore, we will prove that r^+ is the least such relation.

We first prove $[r \subseteq r^+]$.

Example 8.4.1 We show r^+ contains r as follows.

$$\dfrac{\begin{array}{c}\neg[r \subseteq r^+] \\ \neg\forall y.rxy \to [r^+xy] \\ \neg(rxy \to [r^+xy]),\ rxy,\ \neg[r^+xy] \\ \neg\forall r'.[r \subseteq r'] \to [\mathsf{trans}\, r'] \to r'xy \\ \neg([r \subseteq r'] \to [\mathsf{trans}\, r'] \to r'xy),\ [r \subseteq r'] \\ \neg([\mathsf{trans}\, r'] \to r'xy),\ [\mathsf{trans}\, r'],\ \neg r'xy \\ \forall y.rxy \to r'xy,\ rxy \to r'xy\end{array}}{r'xy \mid \neg rxy}$$

We now turn to proving r^+ is transitive.

Example 8.4.2 We show r^+ is transitive as follows.

$$\dfrac{\dfrac{\begin{array}{c}\neg[\mathsf{trans}\, r^+] \\ \neg\forall yz.[r^+xy] \to [r^+yz] \to [r^+xz] \\ \neg\forall z.[r^+xy] \to [r^+yz] \to [r^+xz] \\ \neg([r^+xy] \to [r^+yz] \to [r^+xz]),\ [r^+xy] \\ \neg([r^+yz] \to [r^+xz]),\ [r^+yz],\ \neg[r^+xz] \\ \neg\forall r'.[r \subseteq r'] \to [\mathsf{trans}\, r'] \to r'xz \\ \neg([r \subseteq r'] \to [\mathsf{trans}\, r'] \to r'xz),\ [r \subseteq r'] \\ \neg([\mathsf{trans}\, r'] \to r'xz),\ [\mathsf{trans}\, r'],\ \neg r'xz \\ \forall yz.r'xy \to r'yz \to r'xz \\ \forall z.r'xy \to r'yz \to r'xz \\ r'xy \to r'yz \to r'xz\end{array}}{\dfrac{r'yz \to r'xz}{r'xz \mid \neg r'yz} \mid \neg r'xy}}$$

Now we prove r^+ is the least transitive relation extending r.

Example 8.4.3 We prove r^+ is the least transitive relation extending r by refuting the branch with $[r \subseteq s]$, $[\mathsf{trans}\, s]$ and $\neg[r^+ \subseteq s]$.

$$\dfrac{\begin{array}{c}[r \subseteq s]\\ [\mathsf{trans}\, s]\\ \neg[r^+ \subseteq s]\\ \neg\forall y.[r^+xy] \to sxy\\ \neg([r^+xy] \to sxy),\ [r^+xy],\ \neg sxy\\ [r \subseteq s] \to [\mathsf{trans}\, s] \to sxy\end{array}}{\dfrac{[\mathsf{trans}\, s] \to sxy}{sxy \mid \neg[\mathsf{trans}\, s]} \mid \neg[r \subseteq s]}$$

Inductively defined sets and relations satisfy a property commonly called an **inversion principle** corresponding to their definition. In the case of transitive closure, the inversion principle corresponds to the fact that if $\mathcal{I} \vDash [r^+xy]$ then either $\mathcal{I} \vDash rxy$ or $\mathcal{I} \vDash \exists w.[r^+xw] \wedge [r^+wy]$ (where $w : \iota$ is a variable). Deductively, this means the branch with $[r^+xy]$ and $\neg(rxy \vee \exists w.[r^+xw] \wedge [r^+wy]$ is refutable.

Let inv stand for the formula

$$\lambda xy.rxy \vee \exists w.[r^+xw] \wedge [r^+wy]$$

We refute the branch with $[r^+xy]$ and $\neg[\mathsf{inv}\, xy]$ in a series of steps.

Note that inv corresponds to a binary relation. This relation contains r and is transitive. Once we have proven these facts we can conclude the inversion principle by applying the definition of r^+ to inv.

First we show inv contains r.

Example 8.4.4 $\neg[r \subseteq \mathsf{inv}]$ is refutable.

$$\begin{array}{c}\neg[r \subseteq \mathsf{inv}]\\ \neg\forall y.rxy \to [\mathsf{inv}\, xy]\\ \neg(rxy \to [\mathsf{inv}\, xy]),\ rxy\\ \neg(rxy \vee \exists z.[r^+xz] \wedge [r^+zy]),\ \neg rxy\\ \neg\exists z.[r^+xz] \wedge [r^+zy]\end{array}$$

We now turn to proving inv is transitive. We do this in the next four examples.

Example 8.4.5 We refute the branch with $\exists w.[r^+xw] \wedge [r^+wy]$ and $\neg[r^+xy]$

using $\mathcal{T}_{\text{cut}}$ with $[\mathsf{trans}\, r^+]$.

$$\dfrac{\exists w.[r^+xw] \wedge [r^+wy],\ \neg[r^+xy],\ [r^+xw] \wedge [r^+wy],\ [r^+xw],\ [r^+wy]}{\begin{array}{l|l} \begin{array}{c}[\mathsf{trans}\, r^+] \\ \forall yz.[r^+xy] \to [r^+yz] \to [r^+xz] \\ \forall z.[r^+xw] \to [r^+wz] \to [r^+xz] \\ [r^+xw] \to [r^+wy] \to [r^+xy]\end{array} & \neg[\mathsf{trans}\, r^+] \\ \quad \textbf{(8.4.2)} & \end{array}}$$

$$\dfrac{\dfrac{[r^+wy] \to [r^+xy]}{[r^+xy] \mid \neg[r^+wy]} \mid \neg[r^+xw]}{}$$

Example 8.4.6 We refute $\neg[\mathsf{inv} \subseteq r^+]$ using $\mathcal{T}_{\text{cut}}$ with $[r \subseteq r^+]$.

$$\dfrac{\begin{array}{c}\neg[\mathsf{inv} \subseteq r^+] \\ \neg\forall y.[\mathsf{inv}\, xy] \to [r^+xy] \\ \neg([\mathsf{inv}\, xy] \to [r^+xy]) \\ rxy \vee \exists z.[r^+xz] \wedge [r^+zy] \\ \neg[r^+xy]\end{array}}{\begin{array}{l|l} \begin{array}{c} rxy \\ \hline \begin{array}{l|l} \begin{array}{c}[r \subseteq r^+] \\ \forall y.rxy \to [r^+xy] \\ rxy \to [r^+xy]\end{array} & \neg[r \subseteq r^+] \\ [r^+xy] \mid \neg rxy & \quad \textbf{(8.4.1)} \end{array}\end{array} & \exists z.[r^+xz] \wedge [r^+zy] \\ & \quad \textbf{(8.4.5)} \end{array}}$$

Example 8.4.7 We refute the branch with $[r^+xy]$, $[r^+yz]$, and $\neg[\mathsf{inv}\, xz]$.

$$\dfrac{\begin{array}{c}[r^+xy],\ [r^+yz],\ \neg(rxz \vee \exists z'.[r^+xz'] \wedge [r^+z'z]) \\ \neg rxz \\ \neg\exists z'.[r^+xz'] \wedge [r^+z'z] \\ \neg([r^+xy] \wedge [r^+yz])\end{array}}{\neg[r^+xy] \mid \neg[r^+yz]}$$

Example 8.4.8 We refute $\neg[\mathsf{trans}\,\mathsf{inv}]$ using cut ($\mathcal{T}_{\mathrm{cut}}$) with $[\mathsf{inv} \subseteq r^+]$.

$$\dfrac{\begin{array}{c}\neg[\mathsf{trans}\,\mathsf{inv}]\\ \neg\forall xyz.[\mathsf{inv}\,xy] \to [\mathsf{inv}\,yz] \to [\mathsf{inv}\,xz]\\ \neg\forall yz.[\mathsf{inv}\,xy] \to [\mathsf{inv}\,yz] \to [\mathsf{inv}\,xz]\\ \neg\forall z.[\mathsf{inv}\,xy] \to [\mathsf{inv}\,yz] \to [\mathsf{inv}\,xz]\\ \neg([\mathsf{inv}\,xy] \to [\mathsf{inv}\,yz] \to [\mathsf{inv}\,xz]),\ [\mathsf{inv}\,xy]\\ \neg([\mathsf{inv}\,yz] \to [\mathsf{inv}\,xz]),\ [\mathsf{inv}\,yz],\ \neg[\mathsf{inv}\,xz]\end{array}}{\begin{array}{c}[\mathsf{inv} \subseteq r^+]\\ \forall xy.[\mathsf{inv}\,xy] \to [r^+xy]\\ \forall y.[\mathsf{inv}\,xy] \to [r^+xy]\\ [\mathsf{inv}\,xy] \to [r^+xy]\end{array} \;\Big|\; \neg[\mathsf{inv} \subseteq r^+]}$$

$$\dfrac{[r^+xy]\\ \forall y'.[\mathsf{inv}\,yy'] \to [r^+yy']\\ [\mathsf{inv}\,yz] \to [r^+yz]}{[r^+yz] \mid \neg[\mathsf{inv}\,yz]\ \textbf{(8.4.7)}} \;\Big|\; \neg[\mathsf{inv}\,xy]$$

We can now refute the original branch to prove the inversion principle.

Example 8.4.9 We prove the inversion principle by refuting the branch with $[r^+xy]$ and $\neg[\mathsf{inv}\,xy]$.

$$\dfrac{[r^+xy],\ \neg[\mathsf{inv}\,xy]\\ [r \subseteq \mathsf{inv}] \to [\mathsf{trans}\,\mathsf{inv}] \to [\mathsf{inv}\,xy]}{[\mathsf{trans}\,\mathsf{inv}] \to [\mathsf{inv}\,xy]} \;\Big|\; \neg[r \subseteq \mathsf{inv}]\ \textbf{(8.4.4)}$$

$$[\mathsf{inv}\,xy] \mid \neg[\mathsf{trans}\,\mathsf{inv}]\ \textbf{(8.4.8)}$$

9 Basic Deduction

In this chapter we look at deduction rules that derive valid formulas from valid formulas. A famous example is *modus ponens*

$$\frac{s \to t \qquad s}{t}$$

a deduction rule that appeared in Aristotle's philosophical writings more than two thousand years ago. We call rules that derive valid formulas from valid formulas *basic deduction rules* to distinguish them from tableau rules and other types of deduction rules. Proof systems obtained with basic deduction rules are known as *Hilbert systems*.

We will use basic deduction rules as a means to formulate basic reasoning principles. We will also list a variety of logical laws that express important properties of the logical operations. As a special form of basic deduction we will consider rewriting with equational laws and apply it to obtain prenex forms and negation normal forms.

9.1 Substitution Rule

We start with a deduction rule that does not depend on the presence of logical constants. If a formula s is valid, then every substitution instance $S\theta s$ of s is valid. We express this fact with the **substitution rule**

$$\textbf{Sub}\ \frac{s}{S\theta s}$$

where s ranges over formulas, S over substitution operators, and θ over substitutions. Here is an instance of the substitution rule:

$$\frac{x \vee \neg x}{(x \to y) \vee \neg(x \to y)}$$

The substitution rule is sound since if s is valid, then s evaluates to true no matter how the values of the free variables of s are chosen, and a substitution instance $S\theta s$ just takes away some of the freedom there is for choosing values.

Formally, a **basic rule** is a tuple $\langle s_1, \ldots, s_n, s\rangle$ of formulas where $n \geq 0$. A basic rule $\langle s_1, \ldots, s_n, s\rangle$ is **sound** if the **conclusion** s is valid whenever all of the **premises** $s_1, \ldots, s_n$ are valid. We specify basic rules through **schematic rules** like Sub and speak of the **instances** of a schematic rule. Rules without premises (i.e., $n = 0$) are called **initial**, and rules with premises are called **proper**. A **Hilbert system** is a set of basic rules. A Hilbert system is **sound** if all its rules are sound, and **complete** if it can derive all valid formulas.

Given a Hilbert system, it is common to call the schemes for the initial rules **logical axioms** and the schemes for the proper rules **inference rules**. We may also refer to the schemes for the initial rules as **laws**.

9.2 Replacement Rule

Equality is a basic primitive in mathematical reasoning. In STT equality appears through the identity predicates $=_\sigma : \sigma\sigma o$. The main deduction principle for equality is **replacement of equals with equals**. We formalize this principle with the schematic rule

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

One possible instance is

$$\frac{x \equiv x \vee x \qquad \forall x.\, x \equiv \neg\neg x}{\forall x.\, x \equiv \neg\neg(x \vee x)}$$

As the instance shows, obtaining $C[t]$ from $C[s]$ may involve capture. This is sound since the validity of $s = t$ means that s and t evaluate to the same value no matter which values their free variables take. We can combine Sub and Rep into a new rule

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

which give us the instance

$$\frac{x \equiv x \vee x \qquad \forall x.\, x \equiv \neg\neg x}{\forall x.\, x \equiv \neg\neg x \vee \neg\neg x}$$

We say that SR is a **derived rule** since it can be derived from Sub and Rep:

$$\dfrac{\dfrac{s = t}{S\theta s = S\theta t}\ \text{Sub} \qquad C[S\theta s]}{C[S\theta t]}\ \text{Rep}$$

Note that the soundness of a derived rule follows from the soundness of the rules in its derivation.

Here are rules that account for the reflexivity, symmetry, transitivity, and compatibility (see § 3.9) of equality:

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

Sym, Tra, and Com can be derived from Ref and Rep. The derivation of Sym is as follows:

$$\frac{s = t \qquad \dfrac{}{s = s}\ \text{Ref}}{t = s}\ \text{Rep}$$

Exercise 9.2.1

a) Derive Com from Ref and Rep.

b) Derive Tra from Rep. Convince yourself that every instance of Tra is an instance of Rep.

9.3 Beta and Eta

We formulate the β- and η-law with the following rules:

$$\beta\ \frac{}{(\lambda x.s)t = s_t^x} \qquad \eta\ \frac{}{(\lambda x.sx) = s}\ x \notin \mathcal{N}s$$

One can show that the deduction system given by the rules Rep, β and η can derive an equation $s = t$ if and only if the terms s and t are $\beta\eta$-equivalent. The subsystem consisting of Rep and β can derive Ref

$$\frac{\dfrac{}{(\lambda x.s)x = s}\ \beta \qquad \dfrac{}{(\lambda x.s)x = s}\ \beta}{s = s}\ \text{Rep}$$

and a special case of the substitution rule:

$$\textbf{Sub}_{=}\ \frac{s = t}{s_u^x = t_u^x}$$

The full system consisting of Rep, β, and η can derive the following rules:

$$\textbf{FE}\ \frac{sx = tx}{s = t}\ x \notin \mathcal{N}(s{=}t) \qquad \alpha\ \frac{}{(\lambda x.s) = \lambda y.s_y^x}\ y \notin \mathcal{N}s$$

Rule FE accounts for functional extensionality (see § 2.10), and Rule α formulates the α-law.

We can simplify the basic deduction system consisting of Rep, β and η by replacing η with a more constrained rule η'

$$\textbf{Rep}\ \frac{s = t \qquad C[s]}{C[t]} \qquad \beta\ \frac{}{(\lambda x.s)t = s^x_t} \qquad \eta'\ \frac{}{(\lambda x.fx) = f}$$

where x and f range over variables. This does not give away deductive power since $\text{Sub}_=$ can be derived from Rep and β and η can be derived from η' and $\text{Sub}_=$.

We may also have an intial rule known as **λ-conversion**:

$$\lambda\ \frac{}{s = t}\ s \sim_{\beta\eta} t$$

The soundness of λ-conversion follows from Theorem 4.2.1. Note that λ-conversion subsumes Ref, α, β, and η.

Note that the deduction system we have introduced so far can only prove equations since for all rules the conclusion is an equation whenever all the premises are equations.

Exercise 9.3.1

a) Derive $\text{Sub}_=$ with Com, β, Sym, and Tra.

b) Derive FE from Com, η, and Rep.

c) Derive α from η, β, Sym, and Rep. Hint: See § 3.8.

d) Derive η from η' and $\text{Sub}_=$.

9.4 Modus Ponens and Generalization

Hilbert systems for first-order predicate logic typically come with two proper rules called modus ponens and generalization:

$$\textbf{MP}\ \frac{s \to t \qquad s}{t} \qquad\qquad \textbf{Gen}\ \frac{s}{\forall x.s}$$

With modus ponens we can derive proper rules from initial rules. For instance, given the initial rule

$$\textbf{Tra}'\ \frac{}{s{=}t \to t{=}u \to s{=}u}$$

we can derive the proper rule Tra:

$$\dfrac{\dfrac{\dfrac{}{s=t \to t=u \to s=u}\,\mathrm{Tra}' \qquad s=t}{t=u \to s=u}\,\mathrm{MP} \qquad t=u}{s=u}\,\mathrm{MP}$$

For STT there are complete Hilbert systems whose only proper rule is Rep. Suppose we have a system with Rep and β and the following initial rules:

$$\top\equiv\ \frac{}{s=\top \equiv s} \qquad \top\to\ \frac{}{\top\to s \equiv s} \qquad \forall\top\ \frac{}{(\forall x.s) \equiv s}\ x \notin \mathcal{N}s$$

This system is quite powerful. It can derive MP and Gen. It can also derive Sub and a normalizing substitution rule NS:

$$\textbf{Sub}\ \frac{s}{S\theta s} \qquad\qquad \textbf{NS}\ \frac{s}{\mathcal{B}\theta s}$$

Below is a derivation of the modus ponens rule MP. We exploit that Sym is derivable from Rep and β.

$$\dfrac{\dfrac{}{\top\to t \equiv t}\,\top\to \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{s=\top \equiv s}\,\top\equiv}{s \equiv s=\top}\,\mathrm{Sym} \qquad s}{s \equiv \top}\,\mathrm{Rep} \qquad s \to t}{\top\to t}\,\mathrm{Rep}}{t}\,\mathrm{Rep}$$

Exercise 9.4.1 Derive Gen with $\top\equiv$, $\forall\top$, Sym, and Rep.

Exercise 9.4.2 Derive the substitution rule $\dfrac{s}{s^x_t}$ from $\top\equiv$, Sym, $\mathrm{Sub}_=$, and Rep.

Exercise 9.4.3 Recall that we can derive $\mathrm{Sub}_=$ with Rep and β. Hence weaker versions of $\top\equiv$, $\top\to$, and $\forall\top$ suffice where the metavariable s is replaced by some fixed variable $x_0 : o$. Give the formulations with x_0. For $\forall\top$ the side condition $x \neq x_0$ is necessary.

Exercise 9.4.4 The generalization rule Gen is sound. On the other hand, there are formulas s such that the formula $s \to \forall x.s$ is not valid. Give such a formula s and explain why Gen is sound.

9.5 Important Tautologies

We now look at equational tautologies that express important properties of the propositional constants. Recall that a tautology is a valid propositional formula.

A **Boolean equation** is a propositional formula $s \equiv t$ where s and t contain no other constants but $\bot$, $\top$, $\neg$, $\wedge$, and $\vee$. A **Boolean identity** is a valid Boolean equation. Figure 9.1 shows the most prominent *Boolean identities*. Note that every Boolean identity is a tautology.

The identities in Figure 9.1 exhibit a symmetry known as **duality**: If we take a boolean identity and apply the substitution

$$[\,\bot := \top,\ \top := \bot,\ (\wedge) := (\vee),\ (\vee) := (\wedge)\,]$$

to it, we obtain another Boolean identity. The duality between $\wedge$ and $\vee$ and $\top$ and $\bot$ also shows in the de Morgan laws.

It turns out that the Boolean identities are laws that also hold for the basic set operations. Take a set X and interpret $\bot$ as the empty set $\emptyset$, $\top$ as the full set X, $\neg$ as set complement with respect to X, $\wedge$ as intersection $\cap$, and $\vee$ as union $\cup$. Then the Boolean identities hold for all subsets of X. A general interpretation of the Boolean identities is known as *Boolean algebra*. A main result says that a Boolean equation $s \equiv t$ is valid in a nontrivial Boolean algebra if and only if it is a tautology. The study of Boolean algebras originated in 1847 with George Boole's book *An Investigation of the Laws of Thought* [12]. Whitesitt [68] is a more recent textbook on elementary Boolean algebra.

Figure 9.2 shows tautologies that express important properties of implication and equivalence.

9.6 Boolean Case Analysis and Boolean Expansion

Consider the deduction system given by Rep, β, η, and the initial rule

$$\textbf{Taut}\ \frac{}{s}\ s \text{ tautology}$$

that proves all tautologies.[1] One may hope that this system can prove all valid formulas that contain no non-propositional constants. This is not the case. Here are valid formulas the system cannot prove:

$f\bot \to f\top \to fx$	Boolean case analysis
$fx \equiv \neg x \wedge f\bot \vee x \wedge f\top$	Boolean expansion
$f(f(fx)) \equiv fx$	Kaminski's equation

[1] Recall that it is decidable whether a formula is a tautology.

$x \wedge (y \wedge z) \equiv (x \wedge y) \wedge z$	associativity
$x \vee (y \vee z) \equiv (x \vee y) \vee z$	
$x \wedge y \equiv y \wedge x$	commutativity
$x \vee y \equiv y \vee x$	
$x \wedge x \equiv x$	idempotence
$x \vee x \equiv x$	
$x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$	distributivity
$x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$	
$x \wedge (x \vee y) \equiv x$	absorption
$x \vee (x \wedge y) \equiv x$	
$x \wedge \top \equiv x$	identity
$x \vee \bot \equiv x$	
$x \wedge \bot \equiv \bot$	dominance
$x \vee \top \equiv \top$	
$x \wedge \neg x \equiv \bot$	complement
$x \vee \neg x \equiv \top$	
$\neg(x \wedge y) \equiv \neg x \vee \neg y$	de Morgan
$\neg(x \vee y) \equiv \neg x \wedge \neg y$	
$\neg\top \equiv \bot$	
$\neg\bot \equiv \top$	
$\neg\neg x \equiv x$	double negation

Figure 9.1: Boolean identities

$x \to y \equiv \neg y \to \neg x$	contraposition
$x \to y \to z \equiv x \wedge y \to z$	Schönfinkel
$x \to y \to z \equiv y \to x \to z$	exchange
$x \to x \equiv \top$	
$\bot \to x \equiv \top$	
$\top \to x \equiv x$	
$x \to \bot \equiv \neg x$	
$x \to \top \equiv \top$	
$x \to y \equiv \neg x \vee y$	
$x \to y \equiv (x \equiv x \wedge y)$	
$(x \equiv y) \equiv (y \equiv x)$	commutativity
$(x \equiv (y \equiv z)) \equiv ((x \equiv y) \equiv z)$	associativity
$(x \equiv y) \equiv (\neg x \equiv \neg y)$	contraposition
$(x \equiv x) \equiv \top$	
$(x \equiv \neg y) \equiv (x \not\equiv y)$	
$(x \equiv \bot) \equiv \neg x$	
$(x \equiv \top) \equiv x$	
$(x \equiv y) \equiv (x \to y) \wedge (y \to x)$	Boolean extensionality
$(x \equiv y) \equiv (x \wedge y) \vee (\neg x \wedge \neg y)$	
$(x \not\equiv y) \equiv (x \wedge \neg y) \vee (\neg x \wedge y)$	
$(x \equiv x \wedge y) \equiv (y \equiv x \vee y)$	golden rule
$(x \to y) \wedge x \equiv x \wedge y$	modus ponens
$(x \equiv y) \wedge x \equiv x \wedge y$	
$(x \vee y) \wedge (\neg x \vee z) \to (y \vee z)$	resolution
$(y \wedge z) \to (x \wedge y) \vee (\neg x \wedge z)$	

Figure 9.2: Tautologies for implication and equivalence

We can fix the problem by adding the initial rule

$$\textbf{BCA}\ \frac{}{f\bot \to f\top \to fx}$$

where $f : oo$ and $x : o$ are fixed variables. To demonstrate the power of BCA we derive the rule

$$\textbf{BCR}\ \frac{s^x_\bot \qquad s^x_\top}{s}$$

The derivation of BCR is as follows:

1. $f\bot \to f\top \to fx$ with BCA.
2. $(\lambda x.s)\bot \to (\lambda x.s)\top \to (\lambda x.s)x$ with Sub
   (Sub for $[f := (\lambda x.s)]$ is derivable with Taut, see Exercise 9.4.2).
3. $s^x_\bot \to s^x_\top \to s$ with β and Rep.
4. s from $s^x_\bot$ and $s^x_\top$ with MP (MP is derivable with Taut).

We will not investigate the system with Taut and BCA further. It can prove Boolean expansion and Kaminski's equation, but finding the proofs is tedious.

Exercise 9.6.1 Prove the validity of the following formulas with tableaux.

a) $f\bot \to f\top \to fx$

b) $fx \equiv \neg x \wedge f\bot \vee x \wedge f\top$

c) $(x \equiv y) \wedge fx \equiv (x \equiv y) \wedge fy$

9.7 Quantifier Laws

Figure 9.3 shows the most important equational quantifier laws. The laws are schemes that produce valid formulas. The schemes are necessary so that the laws can address quantification at all types. We can see a law as a scheme for sound and initial basic rules. In fact, laws and schemes for sound and initial basic rules are the same thing.

We call a law **monomorphic** if its logical content can be expressed with a single formula. All the laws for the propositional constants we have stated so far are monomorphic. A law is **polymorphic** if it is not monomorphic. Most laws in Figure 9.3 are polymorphic. An example is the instantiation law $\forall_\sigma f \to fx$. Polymorphism is needed so that the law can address quantification at different types σ. The elimination law $(\forall_o x.x) \equiv \bot$ is an example of a monomorphic law since only quantification at o is concerned.

$\forall f \to fx$		instantiation
$fx \to \exists f$		
$(\forall x.y) \equiv y$	if $x \neq y$	elimination
$(\exists x.y) \equiv y$	if $x \neq y$	
$(\forall x.x) \equiv \bot$		Boolean quantifiers
$(\exists x.x) \equiv \top$		
$(\forall x.fx) \equiv f\bot \wedge f\top$		
$(\exists x.fx) \equiv f\bot \vee f\top$		
$(\forall x.px) \equiv \forall x.p(\neg x)$		
$(\exists x.px) \equiv \exists x.p(\neg x)$		
$(\forall xy.fxy) \equiv \forall yx.fxy$		commutativity
$(\exists xy.fxy) \equiv \exists yx.fxy$		
$(\forall xx.fx) \equiv \forall x.fx$		idempotence
$(\exists xx.fx) \equiv \exists x.fx$		
$\neg(\forall x.fx) \equiv \exists x.\neg fx$		de Morgan
$\neg(\exists x.fx) \equiv \forall x.\neg fx$		
$(\forall x.fx) \wedge y \equiv \forall x.fx \wedge y$	if $x \neq y$	scope extension
$(\exists x.fx) \vee y \equiv \exists x.fx \vee y$	if $x \neq y$	
$(\forall x.fx) \vee y \equiv \forall x.fx \vee y$	if $x \neq y$	
$(\exists x.fx) \wedge y \equiv \exists x.fx \wedge y$	if $x \neq y$	
$(\forall x.fx) \to y \equiv \exists x.fx \to y$	if $x \neq y$	
$(\exists x.fx) \to y \equiv \forall x.fx \to y$	if $x \neq y$	
$y \to (\forall x.fx) \equiv \forall x.y \to fx$	if $x \neq y$	
$y \to (\exists x.fx) \equiv \exists x.y \to fx$	if $x \neq y$	
$(\forall x.fx) \wedge (\forall y.gy) \equiv \forall x.fx \wedge gx$		distributivity
$(\exists x.fx) \vee (\exists y.gy) \equiv \exists x.fx \vee gx$		
$(\forall x.fx) \to (\exists y.gy) \equiv \exists x.fx \to gx$		

Figure 9.3: Quantifier laws

For each σ, the quantifiers $\forall_\sigma$ and $\exists_\sigma$ are dual to each other. This shows in the de Morgan laws. It also shows in the fact that most laws come in pairs where the paired laws are dual to each other.

From the instantiation laws (see Figure 9.3) one can derive corresponding instantiation rules for the quantifiers:

$$\forall I\ \frac{\forall x.s}{s_t^x} \qquad\qquad \exists I\ \frac{s_t^x}{\exists x.s}$$

Here is a derivation of the universal instantiation rule $\forall I$:

$$\dfrac{\dfrac{}{(\lambda x.s)t = s_t^x}\ \beta \qquad \dfrac{\dfrac{\dfrac{}{\forall f \to fx}\ \text{Law}}{(\forall x.s) \to (\lambda x.s)t}\ \text{Sub} \qquad \forall x.s}{(\lambda x.s)t}\ \text{MP}}{s_t^x}\ \text{Rep}$$

Russell's law and Cantor's law are two nonequational quantifier laws whose validity was already established with tableaux:

$\neg\exists x \forall y.\ pxy \equiv \neg pyy$ **Russell's law**

$\neg\exists f \forall g \exists x.\ fx = g$ where $f : \sigma\sigma o$ **Cantor's law**

Both laws say that certain values do not exist. Russell's law yields the undecidability of the halting problem for Turing machines, and Cantor's law yields the uncountability of the power set of the natural numbers. Here are Skolem's law and the axiom of choice:

$(\forall x \exists y.\ pxy) \equiv \exists f \forall x.\ px(fx)$ **Skolem's law**

$\exists C \forall p.\ \exists p \to p(Cp)$ **Choice**

Both laws are weakly valid but not valid (see § 10.1 and § 10.2). Using tableaux, in § 8.2 we have shown that Skolem and Choice are equivalent in the sense that an interpretation satisfies all instances of Skolem iff it satisfies all instances of Choice. The dual of Skolem's law is also weakly valid:

$(\exists x \forall y.\ pxy) \equiv \forall f \exists x.\ px(fx)$ **dual of Skolem's law**

Exercise 9.7.1 Derive the existential instantiation rule $\exists I$.

Exercise 9.7.2 Derive the idempotence laws for quantifiers from the elimination laws for quantifiers.

Exercise 9.7.3 Identify all monomorphic laws in Figure 9.3. Hint: A law is monomorphic if the types of the quantifiers occurring in the law are fixed.

Exercise 9.7.4 Prove the quantifier laws with tableaux. If the laws are polymorphic, assume sorts for the type parameters. For instance, prove $\forall f \to fx$ for specific variables $f : \alpha o$ and $x : \alpha$.

Exercise 9.7.5 Give interpretations in which the following formulas are false:

a) $(\forall x. fx \vee gx) \to (\forall x. fx) \vee (\forall x. gx)$

b) $(\exists x. fx) \to (\exists x. gx) \to \exists x. fx \wedge gx$

c) $(\forall x. fx \to gx) \to (\exists x. fx) \to (\forall x. gx)$

Exercise 9.7.6 (Unique Existential Quantification ∃!) Let $\exists! := \lambda p. \exists x. p \equiv (=)x$. The predicate ∃! checks in every interpretation whether its argument is a singleton set. We can also see $\exists! x.s$ as a quantification that holds if and only if there is a unique x such that s holds. Show that ∃! is not commutative, that is, that the equivalence $(\exists! x \exists! y. pxy) \equiv \exists! y \exists! x. pxy$ is not weakly valid. Hint: Consider $\exists! : (oo)o$.

Exercise 9.7.7 Skolem's law and its dual are logically equivalent. To show this, assume $p : \alpha_1 \alpha_2 o$ and prove with tableaux that the formulas

$$\forall p. (\forall x \exists y. pxy) \equiv \exists f \forall x. px(fx)$$
$$\forall p. (\exists x \forall y. pxy) \equiv \forall f \exists x. px(fx)$$

are logically equivalent.

9.8 Normal Forms and Rewriting

To decide the validity of a formula s, we may first employ a normalization step that translates s into a logically equivalent formula s' and then decide the validity of s'. The decision procedure for validity can then assume that the formulas it works with are in a certain normal form. Here are properties we may require of such a normal form:

1. The formula does not contain a β-redex (β-normal form).
2. Every negated subterm $\neg s$ of the formula has the property that s has the form $xs_1 \dots s_n$ where x is a variable or an identity (**negation normal form**).
3. The formula has the form $Q_1 x_1 \dots Q_n x_n.s$ where $n \geq 0$, $Q_1, \dots, Q_n$ are quantifiers, and s is a quantifier-free formula (**prenex form**).

Often, the normal form can be obtained by rewriting the formula with equational laws. Here is an example that rewrites a formula to negation normal form:

$\neg\forall x\exists y.\ py \vee qxy$	initial formula
$\equiv \exists x.\neg\exists y.\ py \vee qxy$	rewrite with $\neg(\forall x.\, fx) \equiv \exists x.\neg fx$
$\equiv \exists x\forall y.\neg(py \vee qxy)$	rewrite with $\neg(\exists x.\, fx) \equiv \forall x.\neg fx$
$\equiv \exists x\forall y.\neg py \wedge \neg qxy$	rewrite with $\neg(x \vee y) \equiv \neg x \wedge \neg y$

The rewrite steps are obtained from the laws by first applying NS (normalizing substitution) and then possibly Com (compatibility). For the first rewrite step we have

$$\dfrac{\dfrac{}{\neg(\forall x.\, fx) \equiv \exists x.\neg fx}\ \text{Law}}{\neg(\forall x\exists y.\ py \vee qxy) \equiv \exists x.\neg\exists y.\ py \vee qxy}\ \text{NS, } f = \lambda x.\exists y.\ py \vee qxy$$

For the second rewrite step we have

$$\dfrac{\dfrac{\dfrac{}{\neg(\exists y.\, fy) \equiv \forall y.\neg fy}\ \text{Law}}{\neg(\exists y.\ py \vee qxy) \equiv \forall y.\neg(py \vee qxy)}\ \text{NS, } f = \lambda y.\ py \vee qxy}{(\exists x.\neg\exists y.\ py \vee qxy) \equiv \exists x\forall y.\neg(py \vee qxy)}\ \text{Com, } C = \exists x.\bullet$$

Finally, the third rewrite step can be obtained as follows:

$$\dfrac{\dfrac{\dfrac{}{\neg(x \vee y) \equiv \neg x \wedge \neg y}\ \text{Law}}{\neg(py \vee qxy) \equiv \neg py \wedge \neg qxy}\ \text{NS, } x := py,\ y := qxy}{(\exists x\forall y.\neg(py \vee qxy)) \equiv \exists x\forall y.\neg py \wedge \neg qxy}\ \text{Com, } C = \exists x\forall y.\bullet$$

To obtain the equivalence between the initial and the final formula, the three rewrite steps are combined with two applications of Tra (transitivity). We can obtain the rewrite steps by a new rewrite rule

$$\textbf{Rew}\ \dfrac{s = t}{C[\mathcal{B}\theta s] = C[\mathcal{B}\theta t]}$$

that combines NS and Com.

Here is an example that rewrites a formula to prenex form:

$\neg(\forall x.\, px) \vee \forall x\exists y.qxy$	initial formula
$\equiv (\exists x.\neg px) \vee \forall x\exists y.qxy$	rewrite with $\neg(\forall x.\, fx) \equiv \exists x.\neg fx$
$\equiv \forall x.(\exists x.\neg px) \vee \exists y.qxy$	rewrite with $y \vee \forall x.\, fx \equiv \forall x.\, y \vee fx$
$\equiv \forall x\exists y.\neg py \vee qxy$	rewrite with $(\exists x.\, fx) \vee (\exists y.\, gy) \equiv \exists y.\, fy \vee gy$

As it comes to the laws we use, we take the laws given in this section up to minor changes such as α-renaming and commutativity of $\wedge$ and $\vee$.

Following algebraic tradition, we call valid equations $s = t$ **identities**. Recall that we use the word identity also for the constants $=_\sigma$. Thus you have to understand the context to know what we mean.

A **BSR formula** is a relational first-order formula that is in $\exists^*\forall^*$-prenex form, that is, in prenex form where existential quantifiers do not occur below universal quantifiers. An example of a BSR formula is $\exists x \forall y.\, pxy \to x{=}y$. BSR formulas matter since their satisfiability is decidable, a fact that was discovered by Bernays, Schönfinkel, and Ramsey (hence the acronym BSR). A **Bernays-Schönfinkel formula** is an identity-free BSR formula. The satisfiability of BSR formulas can be decided with a terminating tableau system. For Bernays-Schönfinkel formulas such a system was discussed in § 6.4.1. If one has to decide the satisfiability of a formula that is not BSR, one may try to rewrite the formula into an equivalent BSR formula using the quantifier elimination and scope extension laws. If this succeeds, one can decide the satisfiability of the formula. See Exercise 9.8.6.

Exercise 9.8.1 Look at the rewrite steps of the derivation of a prenex form given above: $\neg(\forall x.\, px) \vee \forall x \exists y.qxy \equiv \cdots \equiv \forall x \exists y.\neg py \vee qxy$. For each rewrite step give the context C and the substitution θ to be used with the rule Rew.

Exercise 9.8.2 Give a set of identities so that every formula can be rewritten into negation normal form. Hint: You need one identity per logical constant.

Exercise 9.8.3 Give a set of identities so that every formula can be rewritten into a logically equivalent formula containing no other constants but $\to$ and the universal quantifiers $\forall_\sigma$. Hint: Recall Leibniz' law.

Exercise 9.8.4 Give a set of identities so that every first-order formula can be rewritten into prenex form. Assume the identities

$$x \wedge y \equiv y \wedge x$$
$$x \vee y \equiv y \vee x$$
$$x \to y \equiv \neg x \vee y$$
$$x = y \equiv x \wedge y \vee \neg x \wedge \neg y$$

Six additional rules suffice (for each of the constants $\neg$, $\wedge$, $\vee$ one rule for $\forall$ and one rule for $\exists$).

Exercise 9.8.5 Rewrite each of the following formulas to a logically equivalent formula that is in prenex and negation normal form. Annotate each step with the law you use. Keep the number of quantifiers as small as possible by prefering distributivity laws over scope extension laws. For (c) three quantifiers suffice.

a) $pxy \to \forall xy.\, pxy$

b) $\neg(\exists x \forall y.\, pxy) \vee \forall x.\, qx$

c) $(\neg(\exists x \forall y.\, pxy) \vee \forall x.\, qx) \wedge \forall xy.\, fxy$

Exercise 9.8.6 Rewrite the formula $\forall y \exists x.\, px \to py$ into an equivalent BSR formula. Hint: Use the scope extension laws to first push the quantifiers inside and then pull them out in reverse order.

Exercise 9.8.7 A first-order formula is **monadic** if it does not contain identities and its free variables all have types of the form α or αo. An example of a monadic first-order formula is $\forall y \exists x.\, px \to py$. Show that for every monadic first-order formula one can obtain a logically equivalent monadic first-order formula that is in $\exists^* \forall^*$-prenex form (Bernays-Schönfinkel form). The trick is to first push all quantifiers inside as much as possible (so-called *miniscoping*) and then pull them out in the right order. With miniscoping one can reach for monadic formulas a form where quantifiers are not nested and where quantified formulas don't have free variables (i.e., $\forall x.px \vee \neg qx$). One exploits that a quantifier-free first-order formula can be rewritten into CNF and DNF. One starts at innermost quantifications.

9.9 Skolemization

Two formulas s and t are **equi-satisfiable** if s is satisfiable if and only if t is satisfiable. Logically equivalent formulas are always equi-satisfiable, but there are equi-satisfiable formulas that are not logically equivalent.

Proposition 9.9.1 Let $\exists x.s$ be a formula. Then $\exists x.s$ and s are equi-satisfiable.

A formula is in **universal prenex form** if it has the form $\forall x_1 \ldots \forall x_n.\, s$ where s is quantifier-free (prenex form with universal quantification only). Universal prenex form is also known as **$\forall^*$-prenex form** or **Skolem form**. Most automated provers for first-order logic employ universal prenex form. The following theorem is due to Skolem [59].

Theorem 9.9.2 (Skolem) For every first-order formula there exists an equi-satisfiable first-order formula in universal prenex form.

Proof Let s be a first-order formula. Then there exists a logically equivalent first-order formula t in prenex form. By rewriting with Skolem's law we obtain a formula u in $\exists^*\forall^*$-prenex form (existential quantifiers before universal quantifiers) such that t is satisfiable in a standard model iff u is satisfiable in a standard model. By dropping the existential quantifiers of u we obtain a first-order formula v in universal prenex form such that u and v are equi-satisfiable (Proposition 9.9.1). By Proposition 10.4.3 it follows that s and u are equi-satisfiable. ∎

The translation process from prenex form to universal prenex form described in the proof of the proposition is called **Skolemization**, and the functional variables introduced by rewriting with Skolem's law are called **Skolem functions**. As an example, consider the first-order formula $\forall x\exists y.\, pxy$. Rewriting with Skolem's law yields $\exists f\forall x.\, px(fx)$. Now we drop the existential quantifier and obtain the first-order formula $\forall x.\, px(fx)$. This formula contains the Skolem function f and is equi-satisfiable to the initial formula $\forall x\exists y.\, pxy$.

Exercise 9.9.3 Find equi-satisfiable first-order formulas in universal prenex form for the following first-order formulas. Try to minimize the number of quantifiers and Skolem functions.

a) $(\forall x.\, px) \wedge \exists x.\, qx$

b) $(\forall x.\, px) \to \exists x.\, qx$

c) $\forall x.\, px \to qx \wedge \exists x.\, fx$

d) $\forall xy\exists z.\, pxyz$

e) $(\exists x.\, px) \vee (\exists x.\, qx) \vee \forall x.\, fx$

f) $\exists x\forall y\exists z\forall a\exists b.\, py \wedge qa \to b = x \vee z = a$

9.10 Equality Laws

Figure 9.4 shows the most important laws for equality. Note that α, β and η are special in that they are polymorphic with respect to types and terms.

An equational law $s \equiv t$ gives us two implicational laws $s \to t$ and $t \to s$. We say that the equational law has two directions. Often one direction is more important than the other. This is the case for functional equality, where the direction $(\forall x.\, fx = gx) \to f = g$ is known as functional extensionality. It is also the case for Leibniz, where $(\forall p.\, px \to py) \to x = y$ is the more significant direction.

Exercise 9.10.1 Give a set of identities so that every formula can be rewritten into a logically equivalent formula containing no other constants but identities $=_\sigma$.

$$
\begin{array}{rll}
x = x & & \text{reflexivity} \\
(x = y) \equiv (y = x) & & \text{symmetry} \\
x = y \to y = z \to x = z & & \text{transitivity} \\
x = y \to fx = fy & & \text{functionality} \\
(\forall x.\, fx = gx) \to f = g & & \text{functional extensionality} \\
(\forall x.\, f(fx) = x) \to ((\forall x.p(fx)) \equiv \forall x.px) & & \text{autoinversion} \\
f = g \equiv \forall x.\, fx = gx & & \text{functional equality} \\
x = y \equiv \forall p.\, px \to py & & \text{Leibniz} \\
x = y \to fx \equiv x = y \to fy & & \text{internal replacement} \\
x \wedge y \equiv \forall f.\, fxy \equiv f\top\top & & \text{Henkin} \\
\forall f \equiv f = \lambda x.\top & & \\
\exists f \equiv f \neq \lambda x.\bot & & \\
(\lambda x.s) = \lambda y.s^x_y & \text{if } y \notin \mathcal{N}(\lambda x.s) & \alpha \\
(\lambda x.s)t = s^x_t & & \beta \\
(\lambda x.sx) = s & \text{if } x \notin \mathcal{N}s & \eta
\end{array}
$$

Figure 9.4: Equality laws

Exercise 9.10.2 Show by rewriting that $x \wedge y \equiv (\lambda f.\, fxx) = (\lambda f.\, f\top\top)$ is valid.

Exercise 9.10.3 There is a single monomorphic law in Figure 9.4. Find it.

Exercise 9.10.4 Except for α, β, and η, all laws in Figure 9.4 can be proven with tableaux. Make sure you can do this.

9.11 Andrews' System

Andrews [3] studies a small Hilbert system for STT and shows its completeness.[2] His system takes Rew as its only proper rule. All other rules are initial and appear in Figure 9.5.

[2] Andrews [3] actually considers a system for STT with description. His completeness proof carries over to STT without description.

$$\top \equiv (=_o) = (=) \qquad \mathrm{D}\top$$
$$\bot \equiv (\lambda x.\top) = \lambda x.x \qquad \mathrm{D}\bot$$
$$\neg \equiv (=)\bot \qquad \mathrm{D}\neg$$
$$\wedge = \lambda xy.\ (\lambda f.\ fxy) = \lambda f.\ f\top\top \qquad \mathrm{D}\wedge$$
$$\vee = \lambda xy.\ \neg(\neg x \wedge \neg y) \qquad \mathrm{D}\vee$$
$$\rightarrow = \lambda xy.\ x \equiv x \wedge y \qquad \mathrm{D}\rightarrow$$
$$\forall = (=)(\lambda x.\top) \qquad \mathrm{D}\forall$$
$$\exists = \lambda f.\neg\forall x.\neg fx \qquad \mathrm{D}\exists$$

$$(\forall x.fx) \equiv f\bot \wedge f\top \qquad \mathrm{BUQ}$$
$$(f = g) \equiv \forall x.\ fx = gx \qquad \mathrm{FEQ}$$
$$x = y \rightarrow fx = fy \qquad \mathrm{Fun}$$

$$(\lambda x.x)s = s \qquad \beta_1$$
$$(\lambda x.st)u = (\lambda x.s)u((\lambda x.t)u) \qquad \beta_2$$
$$(\lambda x.s)t = s \quad \text{if } x \notin \mathcal{N}s \qquad \beta_3$$
$$(\lambda xy.s)t = \lambda y.(\lambda x.s)t \quad \text{if } y \notin \mathcal{N}t \cup \{x\} \qquad \beta_4$$

Figure 9.5: Initial rules of Andrews' system

Andrew's system takes the identities as its primary constants. All other constants can be eliminated using Rew and the first 8 rules. In fact, we can say that the first 8 rules define the non-equational constants in terms of the identities.

Andrews' system does not employ a substitution operator. Instead, the basic properties of substitution and β-reduction are provided by the rules $\beta_1, \beta_2, \beta_3, \beta_4$.

If we only consider formulas with identities, Rew, β, and the three rules obtained from BUQ, FEQ, and Fun by eliminating the non-equational constants according to the definitional rules suffice for a complete system.

All rules but Fun are stated as equations. Fun be stated equivalently as the equational rule $x{=}y \equiv x{=}y \wedge fx{=}fy$. This can be shown with D$\rightarrow$, Rew, and the β_i-laws.

Andrews' system constitutes a surprisingly compact account of the logical assumptions STT is based upon. Since it is sound and complete, we can see it as an inductive definition of the set of valid formulas. We can also see it as an

algorithm that enumerates the valid formulas.

To start working with Andrews system, it suffices to understand how we can obtain formulas from the initial rules and the proper rule Rew. No understanding of substitution or β-reduction is needed. This means that checking proofs is easy. However, finding proofs is difficult in Andrews' system. The situation is comparable with a machine language where execution is easy but programming is difficult. In fact, Andrews' erects on his primitive Hilbert system a more high-level deductive system and shows how high-level proofs can be compiled into low-level proofs.

Exercise 9.11.1 Derive Ref in Andrews' system.

Exercise 9.11.2 Derive $\beta_1, \beta_2, \beta_3, \beta_4$ with Ref and β. Use Proposition 3.4.4.

Exercise 9.11.3 Derive the formula $x{=}y \equiv x{=}y \wedge fx{=}fy$ with Fun, D$\to$, Rew, and the β_i-laws. Conversely, derive Fun with the β_i-laws, Rew, and D$\to$ from the initial rule $x{=}y \equiv x{=}y \wedge fx{=}fy$.

9.12 Remarks

Hilbert systems were the first proof systems systematically investigated. They are obtained with rules that derive valid formulas from valid formulas. Finding proofs in Hilbert systems is a difficult and unrewarding task. Clearly, tableau systems are much better suited for proof search.

The first substantial Hilbert system was given by Gottlob Frege [26] in 1879 for a logic that became first-order predicate logic. The first completeness result for first-order predicate logic was obtained by Gödel [31] for a Hilbert system. Church's [20] initial presentation of simple type theory expressed equality with Leibnz' law and came with a Hilbert system with 6 proper rules. A slightly revised version was shown complete by Henkin [35] in 1950. In 1963, Henkin [36] switched to simple type theory with primitive equality and the replacement rule. For simple type theory with primitive equality Andrews' devised and proved complete a surprisingly small Hilbert system.

10 Models

There are many formulas s such that neither s nor $\neg s$ is valid. To show that this is the case for a formula s, we need to construct two models, one in which s is true and one in which s is false. For first-order formulas standard models suffice, but in general nonstandard models are needed. We will introduce the logical relation technique for constructing nonstandard models.

10.1 Description and Choice

The function types of nonstandard models may not contain all functions that exist for this type. As a consequence, formulas that assert the existence of functions may be false in some nonstandard models although they are true in all standard models. We will now consider such formulas. For every type σ we define the following formulas:

$$\exists C \forall_{\sigma o} p.\ \exists p \to p(Cp) \qquad \mathsf{Choice}_\sigma$$
$$\exists D \forall_\sigma x.\ D(=x) = x \qquad \mathsf{Desc}_\sigma$$
$$\forall_\sigma x y \exists f.\ fx{=}y \wedge fy{=}x \qquad \mathsf{Swap}_\sigma$$

Choice_σ asserts the existence of a **choice function** $C : (\sigma o)\sigma$ that for every nonempty subset of σ returns one of its elements. Desc_σ asserts the existence of a **description function** $D : (\sigma o)\sigma$ that for every singleton subset set of σ returns its single element. Finally, Swap_σ asserts for all $x, y : \sigma$ the existence of a **swapping function** $f : \sigma\sigma$ that swaps x and y. Note that the variables x and y in Swap_σ are chosen to be different from each other.

Proposition 10.1.1 The formulas $\mathsf{Choice}_\sigma \to \mathsf{Desc}_\sigma$ and $\mathsf{Desc}_\sigma \to \mathsf{Swap}_\sigma$ are valid for every type σ.

Proof Both claims can be shown with tableaux. The first claim follows with the instantiation $D := C$, and the second claim follows with the instantiation $f := \lambda a.\, D(\lambda b.\, (a{=}x \to b{=}y) \wedge (a{=}y \to b{=}x))$. ∎

Proposition 10.1.2 If Swap_σ is invalid, then Desc_σ and Choice_σ are invalid.

Proof Straightforward consequence of Proposition 10.1.1. ∎

We have defined a frame $\mathcal{D}$ to be standard if $\mathcal{D}(\sigma\tau)$ is the set of all functions $\mathcal{D}\sigma \to \mathcal{D}\tau$. If we assume the so-called **axiom of choice** for the underlying set theory, Choice_σ is true in every standard model for every type σ. In 1904, Zermelo used the axiom of choice to prove that every set can be well-ordered. This result came as a surprise and made clear that the notion of infinite sets is far from straightforward. In response to criticism of his proof, Zermelo carefully spelled out the assumptions about sets he was using in his proof, thus founding set theory. It turned out that one can have different set theories. In particular, one can have a set theory where the axiom of choice is assumed.

Proposition 10.1.3 The formulas Swap_σ, Desc_σ, and Choice_σ are weakly valid for every type σ.

Proof The weak validity of Choice_σ follows from the axiom of choice which we assume for the underlying set theory. The weak validity of Desc_σ and Swap_σ follows with Propositions 10.1.1. ∎

Exercise 10.1.4 One can show that Choice_σ is valid whenever the type σ is constructed using only the base type o.

a) Show with tableaux that Choice_o is valid.

b) Give a term of type $((oo)o)oo$ that evaluates to a choice function for sets of functions of type oo.

Exercise 10.1.5 Show with tableaux that $\mathsf{Choice}_\sigma \to \mathsf{Desc}_\sigma$ is valid.

Exercise 10.1.6 Show with tableaux that $\mathsf{Desc}_\sigma \to \mathsf{Swap}_\sigma$ is valid.

10.2 A Nonstandard Model

We will now construct a nonstandard model in which Swap_α is false for every sort α. Consequently, Swap_α is not valid. Thus, by Proposition 10.1.1 Desc_α and Choice_α are not valid. The construction uses an inductive technique known as **logical relations technique**.

First we construct an admissible frame $\mathcal{D}$. We define sets $\mathcal{D}\sigma$ and relations $\preceq_\sigma \subseteq \mathcal{D}\sigma \times \mathcal{D}\sigma$ by induction on types:

$$\begin{aligned}
\mathcal{D}\beta &:= \{0,1\}\\
\mathcal{D}(\sigma\tau) &:= \{\, f \in \mathcal{D}\sigma \to \mathcal{D}\tau \mid \forall a,b \in \mathcal{D}\sigma\colon\ a \preceq_\sigma b \Longrightarrow fa \preceq_\tau fb \,\}\\
a \preceq_o b &:\Longleftrightarrow \text{true}\\
a \preceq_\alpha b &:\Longleftrightarrow a = 0 \vee a = b\\
f \preceq_{\sigma\tau} g &:\Longleftrightarrow \forall a,b \in \mathcal{D}\sigma\colon\ a \preceq_\sigma b \Longrightarrow fa \preceq_\tau gb
\end{aligned}$$

The definition first fixes for every base type β the set $\mathcal{D}\beta$ and the relation $\preceq_\beta$. Then, step by step, it first obtains the set $\mathcal{D}(\sigma\tau)$ and then the relation $\preceq_{\sigma\tau} \subseteq \mathcal{D}(\sigma\tau) \times \mathcal{D}(\sigma\tau)$. We can think of $\mathcal{D}(\sigma\tau)$ as the set of all monotone functions $\mathcal{D}\sigma \to \mathcal{D}\tau$. Note that all sets $\mathcal{D}\sigma$ are finite since $\mathcal{D}\beta$ is finite for all base types β.

The relations $\preceq_\sigma$ are examples of what is known as **logical relations**. The characteristic features of logical relations is their definition by induction on types and the particular way $\preceq_{\sigma\tau}$ is obtained from $\preceq_\sigma$ and $\preceq_\tau$.

Lemma 10.2.1 $a \preceq_\sigma a$ for all $a \in \mathcal{D}\sigma$.

Proof If σ is a base type, the claim is obvious. Otherwise let $\sigma = \tau\mu$, $f \in \mathcal{D}(\tau\mu)$, and $a \preceq_\tau b$. By the definition of $\preceq_{\tau\mu}$ it suffices to show that $fa \preceq_\mu fb$. This follows by $f \in \mathcal{D}(\tau\mu)$ and the definition of $\mathcal{D}(\tau\mu)$. ∎

Lemma 10.2.2 $\mathcal{D}$ is a frame.

Proof We have to show that $\mathcal{D}$ maps every type to a nonempty set. We show this by induction on types. For the base types the claim is obvious. For a functional type $\sigma\tau$ we know by the inductive hypothesis that there is some $b \in \mathcal{D}\tau$. Let $f = \lambda a{\in}\mathcal{D}\sigma.b$. Clearly, $f \in \mathcal{D}\sigma \to \mathcal{D}\tau$. By Lemma 10.2.1 we have $b \preceq_\tau b$. Hence $f \in \mathcal{D}(\sigma\tau)$ by the definition of $\mathcal{D}(\sigma\tau)$. ∎

It's now easy to see that $\mathcal{D}(\alpha\alpha)$ does not contain the swapping function $\{(0,1),(1,0)\}$ since it is not monotonic. Hence $\mathcal{D}$ is a nonstandard frame.

Lemma 10.2.3 For every sort α: $\{(0,1),(1,0)\} \in (\mathcal{D}\alpha \to \mathcal{D}\alpha) - \mathcal{D}(\alpha\alpha)$.

Lemma 10.2.4 $\mathcal{D}(\sigma o) = (\mathcal{D}\sigma \to \mathcal{D}o)$ and $\mathcal{D}(\sigma\tau o) = (\mathcal{D}\sigma \to (\mathcal{D}\tau \to \mathcal{D}o))$.

Proof Easy. Follows from the fact that $\preceq_o$ is always satisfied. ∎

Lemma 10.2.5 $\mathcal{D}$ admits logical assignments, that is, contains the values needed for the logical constants.

Proof The interpretation for $\bot$ and $\top$ are available since $\mathcal{D}o = \{0,1\}$. The types of the remaing logical constants are of the form σo and $\sigma\tau o$. By Lemma 10.2.4 we know that $\mathcal{D}$ provides all functions for these types. Hence the logical operations are available. ∎

It remains to show that $\mathcal{D}$ is admissible, that is, that every assignment into $\mathcal{D}$ can evaluate every term. This is the most delicate part of the construction. We define a relation $\preceq$ on the assignments into $\mathcal{D}$:

$$\mathcal{I} \preceq \mathcal{J} \;:\Longleftrightarrow\; \forall\sigma \in \mathrm{Ty}\ \forall x \in \mathrm{Nam}_\sigma\colon\ \mathcal{I}x \preceq_\sigma \mathcal{J}x$$

Lemma 10.2.6 Let $s : \sigma$ be a term and $\mathcal{I}$ and $\mathcal{J}$ be assignments into $\mathcal{D}$ such that $\mathcal{I} \preceq \mathcal{J}$. Then $s \in \operatorname{Dom}\hat{\mathcal{I}} \cap \operatorname{Dom}\hat{\mathcal{J}}$ and $\hat{\mathcal{I}}s \preceq_\sigma \hat{\mathcal{J}}s$.

Proof By induction on s. Let $s : \sigma$ be a term and $\mathcal{I}$ and $\mathcal{J}$ be assignments into $\mathcal{D}$ such that $\mathcal{I} \preceq \mathcal{J}$. Case analysis.

Let $s = x$. Then the claim is obvious from the definitions.

Let $s = tu$ where $t : \mu\sigma$. By the inductive hypothesis we have $\hat{\mathcal{I}}t \preceq_{\mu\sigma} \hat{\mathcal{J}}t$ and $\hat{\mathcal{I}}u \preceq_\mu \hat{\mathcal{J}}u$. Hence $(\hat{\mathcal{I}}t)(\hat{\mathcal{I}}u) \preceq_\sigma (\hat{\mathcal{J}}t)(\hat{\mathcal{J}}u)$ by the definition of $\preceq_{\mu\sigma}$. The claim follows by the definition of evaluation functions.

Let $s = \lambda x.t : \mu\tau = \sigma$. By the inductive hypothesis we have $t \in \operatorname{Dom}\widehat{\mathcal{I}^x_a} \cap \operatorname{Dom}\widehat{\mathcal{J}^x_a}$ for all $a \in \mathcal{D}\mu$. Also by the inductive hypothesis we have $\widehat{\mathcal{I}^x_a}t \preceq_\tau \widehat{\mathcal{I}^x_b}t$ whenever $a \preceq_\mu b$. Hence $(\lambda a{\in}\mathcal{I}\mu.\ \widehat{\mathcal{I}^x_a}t) \in \mathcal{D}\sigma$. Thus $s \in \mathcal{D}\hat{\mathcal{I}}$ by the definition of evaluation. Analogously, $s \in \mathcal{D}\hat{\mathcal{J}}$. To show $\hat{\mathcal{I}}s \preceq_\sigma \hat{\mathcal{J}}s$, let $a \preceq_\mu b$. It suffices to show $\hat{\mathcal{I}}sa \preceq_\tau \hat{\mathcal{J}}sb$. This holds since $\hat{\mathcal{I}}sa = \widehat{\mathcal{I}^x_a}t \preceq_\tau \widehat{\mathcal{J}^x_b}t = \hat{\mathcal{J}}sb$ by the definition of evaluation and the inductive hypothesis. ∎

Lemma 10.2.7 $\mathcal{D}$ is an admissible frame that admits logical interpretations.

Proof By Lemmas 10.2.2 and 10.2.5 we know that $\mathcal{D}$ is a frame that admits logical assignments. Let $\mathcal{I}$ be a logical assignment into $\mathcal{D}$. By Lemma 10.2.1 we have $\mathcal{I} \preceq \mathcal{I}$. By Lemma 10.2.6 we know that $\mathcal{I}$ can evaluate every term. Hence $\mathcal{I}$ is a logical interpretation. ∎

Theorem 10.2.8 The formulas Swap_α, Desc_α, and Choice_α are not valid.

Proof By Lemma 10.2.7 we have a logical interpretation $\mathcal{I}$ into $\mathcal{D}$. By Lemma 10.2.3 we know that Swap_α is not true in $\mathcal{I}$. Hence Swap_α is invalid. The invalidity of Desc_α and Choice_α follows with Proposition 10.1.1. ∎

Exercise 10.2.9 Answer the following questions.

a) Does Desc_ι have a standard model?

b) Is Swap_ι satisfiable?

c) Is $\neg\mathsf{Choice}_\iota$ satisfiable?

d) Is $\neg\mathsf{Desc}_\iota$ satisfiable?

e) Is $\neg\mathsf{Swap}_\iota$ satisfiable?

f) Is $\mathsf{Desc}_\iota \wedge \neg\mathsf{Swap}_\iota$ satisfiable?

Exercise 10.2.10 A function $f \in A \to B$ is called a *constant function* if there is some $b \in B$ such that $fa = b$ for all $a \in A$. Let $\mathcal{D}$ be the frame defined by induction on types as follows:

$$\begin{aligned} \mathcal{D}\beta &:= \{0,1\} \\ \mathcal{D}(\sigma\tau) &:= \{\, f \in \mathcal{D}\sigma \to \mathcal{D}\tau \mid f \text{ is a constant function} \,\} \end{aligned}$$

a) Is $\mathcal{D}$ admissible?

b) Does $\mathcal{D}$ admit logical assignments?

Exercise 10.2.11 Let $\mathcal{D}$ be the frame defined by induction on types as follows:

$$\begin{aligned} \mathcal{D}\beta &:= \{0,1\} \\ \mathcal{D}(\sigma\tau) &:= \{\, f \in \mathcal{D}\sigma \to \mathcal{D}\tau \mid \forall a,b \in \mathcal{D}\sigma\colon\ a \preceq_\sigma b \Longrightarrow fa \preceq_\tau fb \,\} \\ a \preceq_\beta b &:\Longleftrightarrow\ a = 0 \vee a = b \\ f \preceq_{\sigma\tau} g &:\Longleftrightarrow\ \forall a,b \in \mathcal{D}\sigma\colon\ a \preceq_\sigma b \Longrightarrow fa \preceq_\tau gb \end{aligned}$$

a) Is $\mathcal{D}$ admissible?

b) Does $\mathcal{D}$ admit logical assignments?

10.3 Simple Type Theory with Description or Choice

If we define the set of valid formula with a smaller class of models, we obtain more valid formulas. Seen this way, restricting the class of models means to weaken the notion of validity. Here are four obvious model classes:

1. All models.
2. Models that satisfy Desc_σ for every type σ.
3. Models that satisfy Choice_σ for every type σ.
4. Standard models.

By Propositions 10.1.1 and 10.1.3 we know that every model that satisfies (3) satisfies (2), and that every model that satisfies (4) satisfies (3). Thus the conditions required of the models get increasingly stronger, and the respective sets of valid formulas get increasingly larger. By constructing suitable nonstandard models one can show that this increase is strict, both as it comes to models and as it comes to formulas. Hence each of the four model classes yields a different notion of validity.

We have already seen proof systems that are complete for the class of all models. These proof systems can be extended so that they become complete for the class with description or the class with choice. Complete proof systems for the class of standard models are impossible (see § 11.3).

The extension of Andrews' system (§ 9.11) to description or choice is straightforward. One justs adds one initial rule:

$$\textbf{Desc}\ \frac{}{\mathsf{Desc}_\sigma} \qquad\qquad \textbf{Choice}\ \frac{}{\mathsf{Choice}_\sigma}$$

In fact, Andrews' [3] only considers simple type theory with description.

The extension of our tableau system TaS to the presence of description or choice functions is also straightforward: One adds rules that look exactly as the rules for Andrews' system. However, as tableau rules Desc and Choice are peculiar since they have no premises. Recall that all rules of TaS have premises. The addition of Desc and Choice destroys termination even for propositional formulas. Moreover, Choice subsumes the cut rule (§ 8.3), as is shown by the following tableau.

$\exists C \forall p.\ \exists p \to p(Cp)$ Choice rule

$\forall p.\ \exists p \to p(Cp)$

$(\exists x.s) \to s$ $p := \lambda x.s$ where $x \notin \mathcal{N}s$

$s \mid \neg\exists x.s$

$\neg s$

10.4 First-Order Validity

We will now show that a first-order formula is valid if and only if it is weakly valid. Thus for first-order formulas there is no need to consider nonstandard interpretations. First we show that the coincidence proposition can be strengthened for first-order formulas.

Proposition 10.4.1 (First-Order Coincidence) $\hat{\mathcal{I}}s = \hat{\mathcal{J}}s$ holds for every first-order formula s and all logical interpretations $\mathcal{I}$ and $\mathcal{J}$ that agree on all variables and all sorts in the footprint of s.

Proof By induction on s. Let $s = xs_1 \dots s_n$ be a first-order formula and $\mathcal{I}$ and $\mathcal{J}$ be logical interpretations that agree on all variables and all sorts in the footprint of s. Case analysis.

- x is a first-order variable. Then $s_1, \dots, s_n$ are first-order terms. Since $s_1, \dots, s_n$ are λ-free, their footprints do not contain sorts. Thus $\hat{\mathcal{I}}s_i = \hat{\mathcal{J}}s_i$ for all $i \in \{1, \dots, n\}$ by Proposition 4.1.5. Hence $\hat{\mathcal{I}}s = \hat{\mathcal{J}}s$ since $\mathcal{I}x = \mathcal{J}x$ by assumption.
- x is a propositional constant. Then $s_1, \dots, s_n$ are first-order formulas. Thus $\hat{\mathcal{I}}s_i = \hat{\mathcal{J}}s_i$ for all $i \in \{1, \dots, n\}$ by the inductive hypothesis. Hence $\hat{\mathcal{I}}s = \hat{\mathcal{J}}s$ since $\mathcal{I}$ and $\mathcal{J}$ agree on propositional constants.

- $x = (=_\alpha)$. Then $n = 2$ and s_1 and s_2 are first-order terms. Analogous to the case where x is a first-order variable we obtain $\hat{\mathcal{I}}s_1 = \hat{\mathcal{J}}s_1$ and $\hat{\mathcal{I}}s_2 = \hat{\mathcal{J}}s_2$. Thus no matter how $\mathcal{I}$ and $\mathcal{J}$ interpret α, we have $\hat{\mathcal{I}}s = \hat{\mathcal{J}}s$.
- $s = \forall_\alpha x.t$. By assumption we have $\mathcal{I}\alpha = \mathcal{J}\alpha$. By the inductive hypothesis we have $\widehat{\mathcal{I}^x_a}t = \widehat{\mathcal{J}^x_a}t$ for every $a \in \mathcal{I}\alpha = \mathcal{J}\alpha$. Hence $\hat{\mathcal{I}}(\lambda x.t) = \hat{\mathcal{J}}(\lambda x.t)$. Thus $\hat{\mathcal{I}}(\forall_\alpha(\lambda x.t)) = \hat{\mathcal{J}}(\forall_\alpha(\lambda x.t))$.
- $s = \exists_\alpha x.t$. Analogous. ∎

Lemma 10.4.2 Let $\mathcal{D}$ be a frame and $\mathcal{F}$ be a standard frame that agrees with $\mathcal{D}$ on all base types. Then $\mathcal{D}(\beta_1 \dots \beta_n) \subseteq \mathcal{F}(\beta_1 \dots \beta_n)$ for all $n \geq 1$ and all base types $\beta_1, \dots, \beta_n$.

Proof By induction on n. For $n = 1$ the claim holds by assumption. Otherwise, let $\sigma = \beta_2 \dots \beta_n$. By the inductive hypothesis we have $\mathcal{D}\sigma \subseteq \mathcal{F}\sigma$. Hence $\mathcal{D}(\beta_1\sigma) \subseteq (\mathcal{D}\beta_1 \to \mathcal{D}\sigma) = (\mathcal{F}\beta_1 \to \mathcal{D}\sigma) \subseteq (\mathcal{F}\beta_1 \to \mathcal{F}\sigma) = \mathcal{F}(\beta_1\sigma)$. ∎

Theorem 10.4.3 Every satisfiable first-order formula has a standard model.

Proof Let $\mathcal{I} \vDash s$ and s be first-order. We choose a standard interpretation $\mathcal{J}$ that agrees with $\mathcal{I}$ on all sorts and all variables that are free in s. This is possible since all variables that are free in s are first-order and hence have a type $\alpha_1 \dots \alpha_n\beta$ which satisfies $\mathcal{I}(\alpha_1 \dots \alpha_n\beta) \subseteq \mathcal{J}(\alpha_1 \dots \alpha_n\beta)$ by Lemma 10.4.2. By Proposition 10.4.1 we have $\mathcal{J} \vDash s$. ∎

Corollary 10.4.4 A first-order formula is valid if and only if it is weakly valid.

Proof Let s be a weakly valid first-order formula and suppose s is not valid. Then $\neg s$ is satisfiable. By Theorem 10.4.3 we know that $\neg s$ has a standard model. Hence there is a standard interpretation in which s is false. Contradiction. The other direction is trivial. ∎

One can show that Theorem 10.4.3 and Corollary 10.4.4 also holds for the larger class of EFO formulas [15]. An **EFO formula** is a formula that does not contain identities and quantifiers for function types.

Exercise 10.4.5 Give a satisfiable formula that has no standard model.

10.5 Remarks

Andrews [2, 1] seems to be the first who published actual constructions of nonstandard models. In [1], Andrews constructs a nonstandard model using logical

relations that shows a problem with Henkin's [35] original definition of interpretations. The construction of nonstandard models by means of logical relations is further developed by Brown [13].

The technically most involved aspect of completeness proofs are model existence theorems that assert the satisfiability of evident sets. For the model constructions in the proofs of the model existence theorems the logical relations technique can be used, both for the construction of standard and nonstandard models [16, 15, 14].

11 Decidability

11.1 Undecidability

In the late 1920's, Hilbert and Ackermann formulated the Entscheidungsproblem [40]: Is there an algorithm that decides the satisfiability of first-order formulas? Here is a famous quote from their textbook *Grunzüge der theoretischen Logik* [40] where they write that the Entscheidungsproblem is the most important problem of mathematical logic.

> Das Entscheidungsproblem ist gelöst, wenn man ein Verfahren kennt, das bei einem vorgelegten logischen Ausdruck durch endlich viele Operationen die Entscheidung über die Allgemeingültigkeit bzw. Erfüllbarkeit erlaubt. (...) Das Entscheidungsproblem muss als das Hauptproblem der mathematischen Logik bezeichnet werden.

As it turned out, the answer to the Entscheidungsproblem is negative: There is no algorithm that decides the satisfiability of first-order formulas. To show this, a formal notion of computability had to be established. One possibility are μ-recursive functions as developed by Herbrand, Gödel, and Kleene. In 1936, Alonzo Church put forward a bold thesis: The computable functions from the natural numbers to the natural numbers are exactly the μ-recursive functions. Based on this assumption he proved the undecidability of the Entscheidungsproblem [19]. In 1937, Alan Turing put forward a similiar thesis: A function from strings to strings is computable if and only if it can be computed with a Turing machine. Based on this assumption he again showed the undecidability of the Entscheidungsproblem [66]. The equivalence of Church's and Turing's thesis was quickly established.

Theorem 11.1.1 (Church 1936) Validity of first-order formulas is undecidable.

11.2 Completeness and Semi-Decidability

A complete proof system gives us an algorithm for enumerating all valid formulas. Since there are complete proof systems, validity and unsatisfiability are semi-decidable. The first completeness result was established by Kurt Gödel [31]

in 1930 for a basic deduction system for first-order logic devised by Ackermann and Hilbert [40]. Combined with Church's undecidability result this implies that the set of satisfiable (first-order) formulas is not semi-decidable. The first completeness result for simple type theory was obtained by Leon Henkin [35] in 1950. Complete tableau systems for simple type theory are a recent achievement [14]. The tableau system TaS introduced in § 7.3 is complete [14].

Theorem 11.2.1 (Henkin 1950) Validity of formulas is semi-decidable.

Exercise 11.2.2 Answer the following questions.

a) Is the set of satisfiable first-order formulas semi-decidable?

b) Is the set of valid formulas semi-decidable?

c) Is the set of unsatisfiable first-order formulas decidable?

11.3 Limited Expressivity

The semi-decidability of validity means that the expressivity of valid formulas is limited. For instance, there cannot be an algorithm that for every Turing machine produces a formula that is valid iff the Turing machine does not halt on a particular input since the respective problem for Turing machines is not semi-decidable. This principal limitation of expressivity was first recognized by Kurt Gödel who in 1931 proved a spectacular result known as Gödel's first incompleteness theorem [32]. To state Gödel's result we need a few definitions.

We fix a sort N and variables $o : N$, $S : NN$, $+ : NNN$, and $\cdot : NNN$. An interpretation is **arithmetic** if it is logical and interprets $(N, o, S, +, \cdot)$ as the natural numbers with zero, the successor function, addition, and multiplication. A formula is **arithmetic** if it is first-order and each of its free names is either a propositional constant or one of $=_N$, $\forall_N$, $\exists_N$, o, S, $+$, $\cdot$. We call a formula **arithmetically valid** if it is arithmetic and true in some arithmetic standard interpretation. By first-order coincidence we know that an arithmetic formula is arithmetically valid if it is true in every arithmetic interpretation. Using the notion of semi-decidability (which did not exist in 1931), we can formulate Gödel's result as follows.

Theorem 11.3.1 (Gödel)
The set of arithmetically valid formulas is not semi-decidable.

Proposition 11.3.2 There is a formula s such that for every arithmetic formula t the formula $s \to t$ is weakly valid if and only if t is arithmetically valid.

In the light of § 4.4 the proposition looks straightforward since we can specify the natural numbers in standard models. However, the specification is up to representation and thus a rigorous proof of the proposition is technically involved.

Theorem 11.3.3 The set of weakly valid formulas is not semi-decidable.

Proof By contradiction. Suppose the set of weakly valid formulas is semi-decidable. By Proposition 11.3.2 it follows that arithmetic validity is semi-decidable. This contradicts Theorem 11.3.1. ∎

We can now say why we care about nonstandard interpretations: While standard interpretations alone give us a noncomputational notion of validity, adding nonstandard interpretations gives us a computational notion of validity. Semi-decidability of validity is crucial since otherwise complete proof systems do not exist.

Theorem 11.3.4 There is no formula s such that for every arithmetic formula t the formula $s \to t$ is valid if and only if t is arithmetically valid.

Proof By contradiction. Suppose there is such a formula. Since validity is semi-decidable, it follows that arithmetic validity is semi-decidable. This contradicts Theorem 11.3.1. ∎

In conclusion we can say that weak validity subsumes arithmetic validity while validity does not, and that subsumption of arithmetic validity is not an option for a logical system since it must be paid for by non-semi-decidability.

Exercise 11.3.5 Argue that logical equivalence of formulas is semi-decidable.

Exercise 11.3.6 Argue that there is no formula stand such that a logical interpretation $\mathcal{I}$ satisfies stand if and only if $\mathcal{I}$ is a standard interpretation.

11.4 First-Order Reduction Classes

A **first-order reduction class** is a set R of first-order formulas such that there is an algorithm that computes for every first-order formula an equi-satisfiable formula that is in R.

Proposition 11.4.1 Let R be a first-order reduction class. Then it is undecidable whether a formula in R is satisfiable.

Proof By contradiction. Suppose there is an algorithm that decides satisfiability for R. Since R is a reduction class we have an algorithm that decides satisfiability of first-order formulas. Thus we have an algorithm that decides validity of first-order formulas. Contradiction with Church's Theorem 11.1.1. ∎

Proposition 11.4.2 First-order formulas in universal prenex form are a first-order reduction class.

Proof Skolemization, see § 9.9. ∎

In 1915, Leopold Löwenheim [49] found a first-order reduction class that shows that equations and most functional types can be translated away. In 1936, László Kálmar [47] sharpened Löwenheim's result such that only variables of type ι and a single variable of type $\iota\iota o$ (a binary predicate) is needed.

Theorem 11.4.3 (Löwenheim-Kálmar) Pure first-order formulas obtained with variables of type ι and a single variable of type $\iota\iota o$ are a first-order reduction class.

11.5 Decidability Results

In 1928, Bernays and Schönfinkel [9] showed that the satisfiability of pure first-order formulas in universal prenex form is decidable. The result was sharped by Ramsey [55] in 1930 so that equations are allowed. We defined formulas of this form as BSR formulas (§ 9.8).

Theorem 11.5.1 (Bernays-Schönfinkel-Ramsey)
Satisfiability of BSR formulas is decidable.

The satisfiability of BSR formulas can be decided with a terminating tableau system. For Bernays-Schönfinkel formulas such a system is presented in § 6.4.1. When extended with the confrontation rule, the system still terminates and decides the satisfiability of BSR formulas [15]. This gives us a straightforward proof of Theorem 11.5.1.

A **basic formula** is a lambda-free formula that contains only quantifiers and identities at base types. Two examples of basic non-first-order formulas are $px \to py \to p(x \wedge y)$ and $px = p(p(px))$ (both are valid). One can show that TaS terminates on basic formulas [16, 15]. Thus TaS decides the satisfiability of basic formulas. One can also show that a basic formula is valid if and only if it is weakly valid [16, 15].

Theorem 11.5.2 (Brown-Smolka) Satisfiability of basic formulas is decidable.

Basic formulas include quantifier-free first-order formulas. Thus satisfiability of quantifier-free first-order formulas is decidable. Using results on congruence closure [51] one can show that satisfiability of quantifier-free first-order formulas is NP-complete.

Exercise 11.5.3 For each of the following claims find out how it follows from the results stated.

a) Satisfiability of quantifier-free first-order formulas is decidable.

b) Satisfiability of formulas $\exists x_1 \ldots x_n . s$ where s is BSR or basic is decidable.

c) Validity of formulas $\forall x_1 \ldots x_n . s$ where s is basic is decidable.

d) Validity of formulas $\forall x_1 \ldots x_n \exists y_1 \ldots y_m . s$ where s is a quantifier-free relational first-order formula is decidable.

11.6 Remarks

There are many decidable fragments of first-order logic. The monograph by Börger, Grädel and Gurevich [17] is dedicated to the classical first-order decision problem and is a valuable source of information.

12 Natural Deduction

In 1935 Gentzen [30] published a paper creating two new kinds of proof systems: natural deduction calculi and sequent calculi. In this chapter we consider natural deduction calculi. The goal of natural deduction is to have a system whose proof rules are natural in the sense that they formalize proof patterns used in mathematical practice. Natural deduction calculi are of both theoretical and practical importance. Different natural deduction calculi can be found in many modern proof assistants (e.g, Isabelle/HOL [52], TPS [4] and Coq [10]).

12.1 Abstract Proof Systems

Before considering another deductive system, we define abstract proof systems and illustrate how Hilbert systems and tableaux calculi are instances of abstract proof systems. Natural deduction calculi will also be instances. Abstract proof systems give us the opportunity to explain the notion of formal proof.

An **abstract proof system** is given by a set of **propositions** and a set of **proof steps**. A **proof step** is a pair $(\{x_1,\ldots,x_n\},x)$, which may be written as

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

where $x_1,\ldots,x_n,x$ are propositions. The objects $x_1,\ldots,x_n$ are the **premises** and the object x is the **conclusion** of the proof step. A proof step is **initial** if it has no premises.

Proof steps carry their premises as a set, so there is no order between the premises. Specifying an order for the premises is not essential.

Given a proof system S and a set P of propositions, the **closure** $S[P]$ is defined inductively:

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 inductive definition of closures, we obtain **proof trees** that verify

statements of the form $x \in S[P]$. The proof tree

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

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 12.1.1 Let S be a proof system. Then:

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

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 $x \in V$ whenever (P, x) applies to V. A proof system S is **sound for** V if every proof step of S is sound for V. A proof system S is **complete for** V if $V \subseteq S[\emptyset]$.

Proposition 12.1.2 A proof system S is sound for V if and only if $S[V] = V$.

Proposition 12.1.3 If a proof system S is sound for V, then $S[\emptyset] \subseteq V$.

If a proof system S is sound and complete for V, we have $V = S[\emptyset]$. If S is sound and complete for V, we also know that $x \in V$ if and only if S has a proof tree for x.

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.

Exercise 12.1.4 Let S be the proof system with $\mathbb{N}$ as propositions and proof steps

$$\{\, (\{x, y\}, z) \mid x, y, z \in \mathbb{N} \wedge x \cdot y = z \,\}.$$

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?

12.1.1 Hilbert systems as Instances of Abstract Proof Systems

A Hilbert system is practically a proof system already. As propositions, we take formulas. As proof steps, we take all pairs $(\{s_1, \ldots, s_n\}, s)$ where $\langle s_1, \ldots, s_n, s\rangle$ is a basic rule defining the Hilbert system.

The following result is true, but we will not prove it.

Proposition 12.1.5 Let V be the set of valid formulas, S be the proof system corresponding to the Andrews system. S is sound and complete for V.

12.1.2 Tableaux Calculi as Instances of Abstract Proof Systems

Recall from Chapter 5 that a branch is a finite set of β-normal formulas. Also, a tableau rule is a tuple $\langle A, A_1, \cdots, A_n\rangle$ which is often written

$$\begin{array}{c|c|c} A & & \\ \hline A_1 & \cdots & A_n \end{array}$$

A tableau calculus is a set of tableau rules.

Let $\mathcal{T}$ be a tableau calculus. To see a tableau calculus $\mathcal{T}$ as an abstract proof system, let $S_{\mathcal{T}}$ be the abstract proof system where propositions are branches and proof steps are pairs $(\{A_1, \ldots, A_n\}, A)$, where $\langle A, A_1, \ldots, A_n\rangle \in \mathcal{T}$. In other words, the proof steps are

$$\frac{A_1 \quad \ldots \quad A_n}{A}$$

where

$$\begin{array}{c|c|c} A & & \\ \hline A_1 & \cdots & A_n \end{array}$$

is a rule in $\mathcal{T}$.

The following propositions are not difficult to verify.

Proposition 12.1.6 A branch A is $\mathcal{T}$-refutable iff $A \in S_{\mathcal{T}}[\emptyset]$.

We know the tableau system TaS is refutation sound. We also claim the tableau system TaS is refutation complete. Consequently we have the following.

Proposition 12.1.7 Let V be the set of unsatisfiable branches. S_{TaS} is sound and complete for V.

12.2 The Structure of Informal Proofs

Consider the following informal proof of

$$(\forall x.px \to qx) \to (\forall x.px) \to \forall x.qx$$

1. Assume $\forall x.px \to qx$. We will prove $(\forall x.px) \to \forall x.qx$.
2. Assume $\forall x.px$. We will prove $\forall x.qx$.
3. Let y be given. We will prove qy.
4. By assumption (1) we have $\forall x.px \to qx$.
5. Hence we have $py \to qy$.
6. By assumption (2) we have $\forall x.px$.
7. Hence we have py.
8. By (5) and (7) we have qy as desired.

There are five kinds of reasoning steps used above. First, when our goal is to prove a formula $s \to t$, then we assume s and prove the subgoal t (see steps 1 and 2). Second, when our goal is to prove a formula $\forall_\sigma s$, then we let $y : \sigma$ be a (fresh) variable and prove the subgoal $[sy]$ (see step 3). Third, we can always use an assumption (see steps 4 and 6). Fourth, when we know $\forall_\sigma s$, then we can establish $[st]$ for any $t : \sigma$ (see steps 5 and 7). Finally, when we know $s \to t$ and s, then we can establish t (see step 8). We summarize these reasoning steps as follows.

implication introduction If our goal is to prove $s \to t$, then it is enough to prove t under the assumed hypothesis s.

forall introduction If our goal is to prove $\forall_\sigma s$, then it is enough to prove $[sy]$ for a fresh variable $y : \sigma$.

assumption If we have assumed s, then we know s.

implication elimination If we know $s \to t$ and we know s, then we know t.

forall elimination If we know $\forall_\sigma s$, then we know $[st]$ for any $t : \sigma$.

Gentzen gave a natural deduction calculus by defining a set of "permissible inference figures." His inference figures handled assumptions by giving each assumption a number and referring to this number to infer the assumption. The

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{1}{\forall x.px \to qx}}{py \to qy}\forall E \quad \dfrac{\dfrac{2}{\forall x.px}}{py}\forall E}{qy}\to E}{\forall x.qx}\forall I_y}{(\forall x.px) \to \forall x.qx}\to I_2}{(\forall x.px \to qx) \to (\forall x.px) \to \forall x.qx}\to I_1$$

Figure 12.1: A natural deduction example

remaining steps were represented by the certain "inference figure schemata:" In Gentzen's notation the informal proof above could be written in natural deduction format as in Figure 12.1. This figure should give the reader some idea of how Gentzen's natural deduction calculus represents proofs. In the next section we give precise definitions. First we consider a few more simple examples.

Let $p, q : o$ be variables. Consider the following informal proofs of $p \to q \to p$, $q \to p \to p$, and $p \to p \to p$.

Example 12.2.1 We show $p \to q \to p$.

1. Assume p. We will show $q \to p$.
2. Assume q. We will show p.
3. We know p by assumption (1). □

The corresponding natural deduction proof is

$$\dfrac{\dfrac{\dfrac{1}{p}}{q \to p}\to I_2}{p \to q \to p}\to I_1$$

Example 12.2.2 We show $q \to p \to p$.

1. Assume q. We will show $p \to p$.
2. Assume p. We will show p.
3. We know p by assumption (2). □

The corresponding natural deduction proof is

$$\dfrac{\dfrac{\overset{2}{p}}{p \to p}\,{\to}I_2}{q \to p \to p}\,{\to}I_1$$

Example 12.2.3 We show $p \to p \to p$.

1. Assume p. We will show $p \to p$.
2. Assume p (again). We will show p.
3. We know p either by assumption (1) or by assumption (2). □

Depending on which assumption we use in step 3 we obtain two slightly different natural deduction proofs. Both are given below.

$$\dfrac{\dfrac{\overset{1}{p}}{p \to p}\,{\to}I_2}{p \to p \to p}\,{\to}I_1 \qquad\qquad \dfrac{\dfrac{\overset{2}{p}}{p \to p}\,{\to}I_2}{p \to p \to p}\,{\to}I_1$$

12.3 Natural Deduction as a Proof System

To present natural deduction as an abstract proof system, we first make the following assumptions and definitions.

We assume Lab is a countably infinite set of labels (i.e., Lab $\cong \mathbb{N}$). We reserve the letters a and b to range over labels. A **labeled formula** is a pair $\langle a, s\rangle$ of a label a and a formula s. We will often write $a : s$ for the labeled formula $\langle a, s\rangle$.

An **assumption context** (or **context**) Γ is a finite function where every member of Γ is a pair $a : s$ where $a \in$ Lab and s is a β-normal formula. In other words, Γ is a finite set of labeled β-normal formulas where no label appears more than once. Note that $\mathrm{Dom}\,\Gamma$ is the set of labels a such that $a : s \in \Gamma$ for some s. We say a label a is **fresh for** Γ if $a \notin \mathrm{Dom}\,\Gamma$. Whenever Γ is a context, $a \notin \mathrm{Dom}\,\Gamma$ and s is a formula, we write $\Gamma, a : s$ for the context given by $\Gamma \cup \{a : s\}$. We often write contexts by enumerating the members, e.g., $a_1 : s_1, \ldots, a_n : s_n$.

A **sequent** is a pair $\langle \Gamma, t\rangle$ where Γ is a context and t is a β-normal formula. We often write $\Gamma \Rightarrow t$ for the sequent $\langle \Gamma, t\rangle$. We write $\Rightarrow t$ for sequents where the context is empty. We say a variable x is **fresh for** $\Gamma \Rightarrow t$ if $x \notin \mathcal{V}t$ and $x \notin \mathcal{V}s$ for every $a : s \in \Gamma$.

Let $\mathcal{I}$ be a logical interpretation (a model) and Γ be a context. We write $\mathcal{I} \models \Gamma$ if $\mathcal{I} \models s$ for every $a : s \in \Gamma$. A sequent is **valid** if for every logical interpretation (model) either $\mathcal{I} \not\models \Gamma$ or $\mathcal{I} \models t$.

A **natural deduction calculus** $\mathcal{N}$ is a proof system where the propositions are sequents. We will measure soundness and completeness of these proof systems relative to the set of valid sequents. The following definitions are simple instances of the notions from abstract proof systems.

- **Soundness:** A rule

$$\frac{\Gamma_1 \Rightarrow s_1 \qquad \cdots \qquad \Gamma_n \Rightarrow s_n}{\Gamma \Rightarrow s}$$

  is **sound** if either one of the premise sequents not valid or the conclusion sequent is valid. $\mathcal{N}$ is **sound** if every rule of $\mathcal{N}$ is sound. Note that if $\mathcal{N}$ is sound, then every $\mathcal{N}$-derivable sequent $\Gamma \Rightarrow s$ is valid.

- **Completeness:** $\mathcal{N}$ is **complete** if every valid $\Gamma \Rightarrow s$ is $\mathcal{N}$-derivable.

We can also inherit the notion of a derivable $\mathcal{N}$ rule:

- **Derivability:** A rule

$$\frac{\Gamma_1 \Rightarrow s_1 \qquad \cdots \qquad \Gamma_n \Rightarrow s_n}{\Gamma \Rightarrow s}$$

  is **$\mathcal{N}$-derivable** if $\Gamma \Rightarrow s$ is in the closure

$$\mathcal{N}[\{\Gamma_1 \Rightarrow s_1, \ldots, \Gamma_n \Rightarrow s_n\}].$$

12.4 A Natural Deduction Calculus for ∀ and →

Most of the rules of our natural deduction calculi will be **introduction** and **elimination** rules for logical constants. We will also need a rule for using assumptions. To formally replay the example proof from the Section 12.2 we need introduction and elimination rules for $\to$ and $\forall_\sigma$.

We define the natural deduction calculus $\mathcal{N}_{\forall\to}$ to be the proof system given by the five rules shown in Figure 12.2.

A is the **assumption** rule.

$\to I$ is the **implication introduction** rule.

$\to E$ is the **implication elimination** rule.

$\forall I$ is the **forall introduction** rule.

$\forall E$ is the **forall elimination** rule.

In most $\mathcal{N}$-rules the left side of a sequent is the same in the premises and the conclusion. The exception in Figure 12.2 is $\to I$. The left side of the premise

$$A_a\ \frac{}{\Gamma \Rightarrow s}\ a : s \in \Gamma \qquad \forall I_y\ \frac{\Gamma \Rightarrow [sy]}{\Gamma \Rightarrow \forall_\sigma s}\ y : \sigma \text{ fresh for } \Gamma \Rightarrow s$$

$$\rightarrow I_a\ \frac{\Gamma, a : s \Rightarrow t}{\Gamma \Rightarrow s \rightarrow t}\ a \notin \mathrm{Dom}\,\Gamma \qquad \forall E\ \frac{\Gamma \Rightarrow \forall_\sigma s}{\Gamma \Rightarrow [st]} \qquad \rightarrow E\ \frac{\Gamma \Rightarrow s \rightarrow t \qquad \Gamma \Rightarrow s}{\Gamma \Rightarrow t}$$

Figure 12.2: $\mathcal{N}_{\forall\rightarrow}$ rule schemas with explicit contexts

$$\forall I_y\ \frac{[sy]}{\forall_\sigma s}\ y : \sigma \text{ fresh} \qquad \rightarrow I_a\ \frac{\begin{array}{c} a \\ s \\ \vdots \\ t \end{array}}{s \rightarrow t}\ a \text{ fresh} \qquad \forall E\ \frac{\forall_\sigma s}{[st]} \qquad \rightarrow E\ \frac{s \rightarrow t \qquad s}{t}$$

Figure 12.3: $\mathcal{N}_{\forall\rightarrow}$ rule schemas without explicit contexts

of $\rightarrow I$ is $\Gamma, a : s$ while the left side of the conclusion is Γ. We say the assumption $a : s$ is **discharged** by the rule $\rightarrow I$.

In $\mathcal{N}$ proof trees the context is usually clear. Consequently, we only give the right side s in the derivation instead of $A \Rightarrow s$. In rules such as $\rightarrow I$ which discharge an assumption $a : s$ we give the label a with the name of the rule, e.g., $\rightarrow I_a$. When we use an assumption, we give the label of the assumption over the formula. We have already seen an example of such a natural deduction proof previously (Figure 12.1). The reader should reconsider this figure and make sure the context of each sequent is clear.

We can also use these conventions to present natural deduction rule schema without explicit contexts. We show the schema defining $\mathcal{N}_{\forall\rightarrow}$ in Figure 12.3. In this format we omit the assumption rule.

12.4.1 A Reduction to $\forall$ and $\rightarrow$

Every formula of STT can be rewritten to an equivalent formula that only uses the logical constants $\rightarrow$ and the universal quantifiers $\forall_\sigma$. The relevant equivalences are displayed in Figure 12.4.

The equivalences suggest certain rules which should be $\mathcal{N}_{\forall\rightarrow}$-derivable. For example, Leibniz equality should be reflexive. This suggests that for any context

$$x = y \equiv \forall p.px \to py \qquad \text{(Leibniz)}$$
$$\top \equiv \forall x.x \to x$$
$$\bot \equiv \forall x.x$$
$$\neg x \equiv x \to \bot$$
$$x \wedge y \equiv \forall p.(x \to y \to p) \to p$$
$$x \vee y \equiv \forall p.(x \to p) \to (y \to p) \to p$$
$$(\exists f) \equiv \forall p.(\forall x.fx \to p) \to p$$

Figure 12.4: Reduction to → and ∀

Γ, β-normal term $s : \sigma$ and variable $p : \sigma o$ where $p \notin \mathcal{N}s$ the rule

$$\frac{}{\Gamma \Rightarrow \forall p.ps \to ps}$$

should be $\mathcal{N}_{\forall\to}$-derivable. We prove this by giving a proof tree (without explicit contexts). We use a variable $q : \sigma o$ which is fresh for $\Gamma \Rightarrow \forall p.ps \to ps$ and a label a which is fresh for Γ.

$$\dfrac{\dfrac{\dfrac{\begin{matrix}a\\qs\end{matrix}}{qs \to qs}\to I_a}{\forall p.ps \to ps}\forall I_q$$

Leibniz equality should also allow one to replace equals by equals. We prove $\mathcal{N}_{\forall\to}$-derivability of the rule

$$\frac{\Gamma \Rightarrow \forall p.ps \to pt \qquad \Gamma \Rightarrow [us]}{\Gamma \Rightarrow [ut]}$$

where Γ is a context, $s, t : \sigma$, $u : \sigma o$ are terms and $p : \sigma o$ is a variable such that $p \notin \mathcal{N}s \cup \mathcal{N}t$. The following proof tree suffices to establish derivability of the rule.

$$\dfrac{\dfrac{\forall p.ps \to pt}{[us \to ut]}\forall E \qquad [us]}{\Gamma \Rightarrow [ut]}\to E$$

Exercise 12.4.1 Show the following sequents are $\mathcal{N}_{\forall\to}$-derivable by giving proof trees (without explicit contexts).

a) $\Gamma \Rightarrow \forall x.x \to x$

b) $\Gamma, a : (\forall x.x) \Rightarrow s$

c) $\Gamma, a : \forall p.(s \to t \to p) \to p \Rightarrow s$ where $p \notin \mathcal{N}s \cup \mathcal{N}t$.

d) $\Gamma, a : s, b : t \Rightarrow \forall p.(s \to t \to p) \to p$ where $p \notin \mathcal{N}s \cup \mathcal{N}t$.

e) $\Gamma, a : \forall p.ps \to pt \Rightarrow \forall p.pt \to ps$ where $s, t : \sigma$, $p : \sigma o$ and $p \notin \mathcal{N}s \cup \mathcal{N}t$.

Exercise 12.4.2 Prove the validity of the following equivalences in Figure 12.4 using the tableau system TaS^+.

12.4.2 Incompleteness of $\forall$ and $\to$

The following proposition is easy to show.

Proposition 12.4.3 $\mathcal{N}_{\forall\to}$ is sound.

Since there are no rules for constants except $\to$ and $\forall$, we cannot expect $\mathcal{N}_{\forall\to}$ to be able to derive all valid sequents. One might expect that $\mathcal{N}_{\forall\to}$ is complete if we only allow the constants $\to$ and $\forall$. In fact $\mathcal{N}_{\forall\to}$ is incomplete even with respect to this fragment. There are three reasons for this incompleteness: functional extensionality, Boolean extensionality, and classical reasoning. We can represent functional and Boolean extensionality as formulas using only $\forall$ and $\to$ via Leibniz equality. A simple way of representing classical reasoning using only $\forall$ and $\to$ is using Peirce's Law. We state the following result without proof.

Proposition 12.4.4 $\mathcal{N}_{\forall\to}$ is incomplete. In particular, $\mathcal{N}_{\forall\to}$ cannot derive any of the following valid sequents.

1. (Boolean Extensionality) $\Rightarrow \forall xy.(x \to y) \to (y \to x) \to \forall p.px \to py$
2. (Functional Extensionality) $\Rightarrow \forall_{\sigma\tau} fg.(\forall xp.p(fx) \to p(gx)) \to \forall q.qf \to qg$
3. (Peirce's Law) $\Rightarrow \forall xy.((x \to y) \to x) \to x$

There is a simple way to obtain a complete calculus for the fragment with only $\to$ and $\forall_\sigma$. We would need to add new rules corresponding to each of the three missing principles. We will not pursue this option here.

12.5 Natural Deduction Rules for $\top$, $\bot$, $\neg$, $\wedge$ and $\vee$

There is a simple introduction rule for $\top$ and a simple elimination rule for $\bot$.

$$\top I\ \frac{}{\top} \qquad\qquad \bot E\ \frac{\bot}{s}$$

We describe these rules in natural language as follows.

$\top I$: (**true introduction**) If our goal is to prove $\top$, then we are done.

$\bot E$: (**false elimination**) If we know $\bot$, then we know anything. (We can only establish $\bot$ if we have inconsistent assumptions.)

Example 12.5.1 We can use the rules for $\top$ and $\bot$ to give two different natural deduction proofs of $\Rightarrow \bot \to \top$ shown below.

$$\dfrac{\dfrac{}{\top}\,\top I}{\bot \to \top}\,{\to}I_a \qquad\qquad \dfrac{\dfrac{\dfrac{a}{\bot}}{\top}\,\bot E}{\bot \to \top}\,{\to}I_a$$

We next turn to the introduction and elimination rules for negation.

$$\neg I_a\ \dfrac{\begin{array}{c} a \\ s \\ \vdots \\ \bot \end{array}}{\neg s}\ a \text{ fresh} \qquad\qquad \neg E\ \dfrac{\neg s \quad s}{\bot}$$

We describe these rules in natural language as follows.

$\neg I$: (**negation introduction**) If our goal is to prove $\neg s$, then it is enough to prove $\bot$ under the assumption s. (That is, we prove it is inconsistent to assume s.)

$\neg E$: (**negation elimination**) If we know $\neg s$ and s, then we know $\bot$.

Example 12.5.2 Consider the sequent $\Gamma \Rightarrow \neg\neg\neg s \to \neg s$ for some context Γ and formula s. To illustrate the rules for negation, we prove the following rule is derivable from the rules above. First let us give an informal argument.

Assume $\neg\neg\neg s$. Our goal is to prove $\neg s$. Assume s. Our goal is to prove $\bot$ (a contradiction). Since $\neg\neg\neg s$ holds, we will have a contradiction if we show $\neg\neg s$. Our goal is now to show $\neg\neg s$. Assume $\neg s$ holds. This contradicts our assumption of s.

The following natural deduction proof (without explicit contexts) corresponds to the informal argument above.

$$\dfrac{\dfrac{a \qquad \dfrac{\dfrac{\dfrac{c \quad b}{\neg s \quad s}\,\neg E}{\bot}\,\neg I_c}{\neg\neg\neg s \qquad \neg\neg s}\,\neg E}{\bot}\,\neg I_b}{\neg s}\,{\to}I_a$$
$$\neg\neg\neg s \to \neg s$$

Exercise 12.5.3 Show the following sequents are derivable using the rules $\top I$, $\bot E$ $\neg I$, $\neg E$ $\to I$ and $\to E$ by giving proof trees (without explicit contexts).

a) $\Gamma, a : s \Rightarrow \neg\neg s$

b) $\Gamma, a : s \to \bot \Rightarrow \neg s$

c) $\Gamma, a : \neg s \Rightarrow s \to t$

d) $\Gamma, a : s \to t \Rightarrow \neg t \to \neg s$

We next turn to the natural deduction rules for conjunction. These rules are easy to understand.

$$\wedge I\ \frac{s \quad t}{s \wedge t} \qquad \wedge E_L\ \frac{s \wedge t}{s} \qquad \wedge E_R\ \frac{s \wedge t}{t}$$

We describe these rules in natural language as follows.

$\wedge I$: (**conjunction introduction**) If our goal is to prove $s \wedge t$, then it is enough to prove s and then prove t.

$\wedge E_L$: (**conjunction elimination left**) If we know $s \wedge t$, then we know s.

$\wedge E_R$: (**conjunction elimination right**) If we know $s \wedge t$, then we know t.

The natural deduction rules for disjunction are also easy to understand. The introduction rules in essence say that in order to prove $s \vee t$ we must decide which of the disjuncts to prove. The elimination corresponds to proof by cases. If we have established a disjunction $s \vee t$, then we know we can prove any goal by separately considering the first case where s holds and the second case where t holds.

$$\vee I_L\ \frac{s}{s \vee t} \qquad \vee I_R\ \frac{t}{s \vee t} \qquad \vee E_{a,b}\ \frac{s \vee t \quad \begin{array}{c}[a]\\ s\\ \vdots\\ u\end{array} \quad \begin{array}{c}[b]\\ t\\ \vdots\\ u\end{array}}{u}\ a, b \text{ fresh}$$

We describe these rules in natural language as follows.

$\vee I_L$: (**disjunction introduction left**) If our goal is to prove $s \vee t$, then it is enough to prove s.

$\vee I_R$: (**disjunction introduction right**) If our goal is to prove $s \vee t$, then it is enough to prove t.

$\vee E$: (**disjunction elimination**) If we know $s \vee t$ and our goal is to prove u, then it is enough to prove u under the assumption s and also prove u under the assumption t. (This is a form of proof by cases.)

Example 12.5.4 We illustrate the use of the rules for conjunction and disjunction by proving the sequent $\Gamma \Rightarrow s \vee (t \wedge u) \to (s \vee t) \wedge (s \vee u)$ where Γ is a context and $s, t, u : o$ are formulas. This is, in fact, an example used by Gentzen [30]. Consider first the informal argument.

Assume $s \vee (t \wedge u)$. We must show $(s \vee t) \wedge (s \vee u)$. We consider two cases. In the first case we assume s holds. Since s holds, $s \vee t$ and $s \vee u$ hold so we are done. In the second case we assume $t \wedge u$ holds. Hence both t and u hold. Hence both $s \vee t$ and $s \vee u$ hold, and we are done.

$$\dfrac{\dfrac{\overset{a}{s \vee (t \wedge u)} \qquad \dfrac{\dfrac{\overset{b}{s}}{s \vee t} \vee I_L \qquad \dfrac{\overset{b}{s}}{s \vee u} \vee I_L}{(s \vee t) \wedge (s \vee u)} \wedge I \qquad \dfrac{\dfrac{\dfrac{\overset{c}{t \wedge u}}{t} \wedge E_L}{s \vee t} \vee I_R \qquad \dfrac{\dfrac{\overset{c}{t \wedge u}}{u} \wedge E_R}{s \vee u} \vee I_R}{(s \vee t) \wedge (s \vee u)} \wedge I}{(s \vee t) \wedge (s \vee u)} \vee E_{b,c}}{s \vee (t \wedge u) \to (s \vee t) \wedge (s \vee u)} \to I_a$$

Exercise 12.5.5 Let a be a label. Find a context Γ and formulas s, t, u such that a is fresh for Γ and the following rule is unsound.

$$\dfrac{\begin{array}{cc} & \overset{a}{s} \\ & \vdots \\ s \vee t & u \end{array}}{u}$$

12.6 Natural Deduction Rules for ∃

$$\exists I \ \dfrac{[st]}{\exists_\sigma s} \qquad\qquad \exists E_{a,y} \ \dfrac{\exists_\sigma s \qquad \begin{array}{c} \overset{a}{[sy]} \\ \vdots \\ t \end{array}}{t} \ a, y \text{ fresh}$$

We describe these rules in natural language as follows.

$\exists I$: If our goal is to prove $\exists_\sigma s$, then it is enough to prove $[st]$ for a particular term $t : \sigma$.

$\exists E$: If we know $\exists_\sigma s$ and our goal is to prove t, then it is enough to prove t under the assumption that $[sy]$ holds for a fresh $y : \sigma$.

Example 12.6.1 We illustrate the use of the rules for quantifiers by proving $\Rightarrow (\exists x \forall y.pxy) \to \forall y \exists x.pxy$ where $x : \sigma$, $y : \tau$ and $p : \sigma\tau o$ are variables. This is another example used by Gentzen [30]. Consider first the informal argument.

Assume $\exists x \forall y.pxy$. We will prove $\forall y \exists x.pxy$. Let z be fresh and assume $\forall y'.pzy'$. (Note that if we happen to choose z to be the same as y, renaming will be required.) Let w also be fresh. We will prove $\exists x'.px'w$. Using our assumption we have pzw which is enough to conclude $\exists x'.px'w$.

$$\dfrac{\dfrac{a}{\exists x \forall y.pxy} \qquad \dfrac{\dfrac{\dfrac{\dfrac{b}{\forall y'.pzy'}\,\forall E}{pzw}}{\exists x'.px'w}\,\exists I}{\forall y \exists x.pxy}\,\forall I_w}{\forall y \exists x.pxy}\,\exists E_{b,z}}{(\exists x \forall y.pxy) \to \forall y \exists x.pxy}\,{\to}I_a$$

12.7 Natural Deduction Rules for Equality

The introduction and elimination rules for equality correspond to reflexivity and replacement of equals by equals. Replacement is formulated in a restricted way using a term u instead of a context. Consequently, unlike basic deduction, we do not allow capture while applying the rule. In fact, allowing capture would no longer be sound since we have a context of assumptions.

$$=I\ \frac{}{s = s} \qquad\qquad =E\ \frac{s = t \qquad [us]}{[ut]}$$

We describe these rules in natural language as follows.

$=I$: If our goal is to prove $s = s$, then we are done.

$=E$: If we know $s = t$ and $[us]$, then we know $[ut]$.

As an example we prove symmetry of equality. The application of $=E$ below uses the term $\lambda x.x = s$.

$$\dfrac{\dfrac{a}{s = t} \qquad \dfrac{}{s = s}\,{=}I}{t = s}\,{=}E}{s = t \to t = s}\,{\to}I_a$$

12.8 An Incomplete Natural Deduction Calculus for STT

At this point, we have given rules for every logical constant. Let $\mathcal{N}_{\mathrm{STT}}^{-}$ be the natural deduction proof system with these rules. To be precise, let $\mathcal{N}_{\mathrm{STT}}^{-}$ be the natural deduction proof system consisting of the rules $\top I$, $\bot E$, $\neg I$, $\neg E$, $\to I$, $\to E$, $\wedge I$, $\wedge E_L$, $\wedge E_R$, $\vee I_L$, $\vee I_R$, $\vee E$, $\forall I$, $\forall E$, $\exists I$, $\exists E$, $=I$ and $=E$. These rules are given (along with three extra rules not included in $\mathcal{N}_{\mathrm{STT}}^{-}$) in Figures 12.5 and 12.6.

This proof system is sound, but is not complete.

Proposition 12.8.1 $\mathcal{N}_{\mathrm{STT}}^{-}$ is incomplete. In particular, $\mathcal{N}_{\mathrm{STT}}^{-}$ cannot derive any of the following valid sequents.

1. (Boolean Extensionality) $a : x \to y, b : y \to x \Rightarrow x \equiv y$.
2. (Functional Extensionality) $a : \forall x.fx = gx) \Rightarrow f = g$ where $f, g : \sigma\tau$.
3. (Peirce's Law) $\Rightarrow ((x \to y) \to x) \to x$
4. (Excluded Middle) $\Rightarrow x \vee \neg x$
5. (Double Negation) $\Rightarrow \neg\neg x \to x$

There are essentially three reasons $\mathcal{N}_{\mathrm{STT}}^{-}$ is incomplete. First, Boolean extensionality is not provable. Second, functional extensionality is not provable. Third, the calculus is **intuitionistic** instead of **classical**. If the calculus were classical we would be able to prove Peirce's Law, excluded middle and double negation. One can prove using $\mathcal{N}_{\mathrm{STT}}^{-}$ that all three of these principles are equivalent. We will not pursue this in detail.

Exercise 12.8.2 Show the following sequents are $\mathcal{N}_{\mathrm{STT}}^{-}$-derivable using by giving proof trees (without explicit contexts).

a) $\Gamma, a : s \wedge (t \vee u) \Rightarrow (s \wedge t) \vee (s \wedge u)$

b) $\Rightarrow \exists_{\sigma\sigma} f.\forall x.fx = x$

Exercise 12.8.3 Use $\mathcal{N}_{\mathrm{STT}}^{-}$ to prove excluded middle implies double negation by proving the sequent $a : \forall x.x \vee \neg x \Rightarrow \forall x.\neg\neg x \to x$.

Exercise 12.8.4 Use $\mathcal{N}_{\mathrm{STT}}^{-}$ to prove Cantor's Theorem:

$$\Rightarrow \neg\exists f \forall g \exists x.fx = g$$

where $f : \sigma\sigma o$, $g : \sigma o$ and $x : \sigma$.

12.9 A Complete Natural Deduction Calculus for STT

We now add three rules giving us classical logic and both forms of extensionality. Classical reasoning is reflected by a proof-by-contradiction rule **Con**. Boolean

extensionality is given by a rule **BE** and functional extensionality is given by a rule **FE**.

We describe these rules in natural language as follows.

Con: If our goal is to prove s, then it is enough to prove $\bot$ under the assumption $\neg s$. (This corresponds to proof by contradiction.)

BE: If our goal is to prove $s \equiv t$, then it is enough to prove $s \to t$ and $t \to s$.

FE: If our goal is to prove $s =_{\sigma\tau} t$, then it is enough to prove $\forall x.[sx] =_\tau [tx]$.

Let $\mathcal{N}_{\mathrm{STT}}$ be the natural deduction calculus given by all the rules in Figure 12.5. We also show the rules with explicit contexts in Figure 12.6.

Exercise 12.9.1 Give an $\mathcal{N}_{\mathrm{STT}}$-derivation (without explicit contexts) for the sequent $\Gamma \Rightarrow \neg\neg s \to s$ where Γ is a context and $s : o$ is a formula.

Exercise 12.9.2 Show the following sequent is $\mathcal{N}_{\mathrm{STT}}$-derivable using by giving proof trees (without explicit contexts).

$$\Gamma, a : pq \Rightarrow p(\lambda x.\neg\neg(qx))$$

where $p : (\sigma o)o$ and $q : \sigma o$ are variables.

12.10 Remarks

$$\top I\ \frac{}{\top} \qquad \bot E\ \frac{\bot}{s} \qquad \neg I_a\ \frac{\begin{array}{c}[a]\\ s\\ \vdots\\ \bot\end{array}}{\neg s}\ a \text{ fresh} \qquad \neg E\ \frac{\neg s \qquad s}{\bot} \qquad \to I_a\ \frac{\begin{array}{c}[a]\\ s\\ \vdots\\ t\end{array}}{s \to t}\ a \text{ fresh}$$

$$\to E\ \frac{s \to t \qquad s}{t} \qquad \wedge I\ \frac{s \qquad t}{s \wedge t} \qquad \wedge E_L\ \frac{s \wedge t}{s} \qquad \wedge E_R\ \frac{s \wedge t}{t} \qquad \vee I_L\ \frac{s}{s \vee t}$$

$$\vee I_R\ \frac{t}{s \vee t} \qquad \vee E_{a,b}\ \frac{s \vee t \qquad \begin{array}{c}[a]\\ s\\ \vdots\\ u\end{array} \qquad \begin{array}{c}[b]\\ t\\ \vdots\\ u\end{array}}{u}\ a, b \text{ fresh} \qquad \forall I_y\ \frac{[sy]}{\forall_\sigma s}\ y : \sigma \text{ fresh}$$

$$\forall E\ \frac{\forall_\sigma s}{[st]} \qquad \exists I\ \frac{[st]}{\exists_\sigma s} \qquad \exists E_{a,y}\ \frac{\exists_\sigma s \qquad \begin{array}{c}[a]\\ [sy]\\ \vdots\\ t\end{array}}{t}\ a, y \text{ fresh} \qquad =I\ \frac{}{s = s}$$

$$=E\ \frac{s = t \qquad [us]}{[ut]} \qquad \mathbf{Con}_a\ \frac{\begin{array}{c}[a]\\ \neg s\\ \vdots\\ \bot\end{array}}{s}\ a \text{ fresh} \qquad \mathbf{BE}\ \frac{s \to t \qquad t \to s}{s \equiv t}$$

$$\mathbf{FE}\ \frac{\forall x.[sx] = [tx]}{s =_{\sigma\tau} t}$$

Figure 12.5: The natural deduction system $\mathcal{N}_{\mathrm{STT}}$ without explicit contexts

$$A_a\ \frac{}{\Gamma \Rightarrow s}\ a : s \in \Gamma \qquad \top I\ \frac{}{\Gamma \Rightarrow \top} \qquad \bot E\ \frac{\Gamma \Rightarrow \bot}{\Gamma \Rightarrow s} \qquad \neg I_a\ \frac{\Gamma, a : s \Rightarrow \bot}{\Gamma \Rightarrow \neg s}\ a \notin \mathrm{Dom}\,\Gamma$$

$$\neg E\ \frac{\Gamma \Rightarrow \neg s \qquad \Gamma \Rightarrow s}{\Gamma \Rightarrow \bot} \qquad \to I_a\ \frac{\Gamma, a : s \Rightarrow t}{\Gamma \Rightarrow s \to t}\ a \notin \mathrm{Dom}\,\Gamma \qquad \to E\ \frac{\Gamma \Rightarrow s \to t \qquad \Gamma \Rightarrow s}{\Gamma \Rightarrow t}$$

$$\wedge I\ \frac{\Gamma \Rightarrow s \qquad \Gamma \Rightarrow t}{\Gamma \Rightarrow s \wedge t} \qquad \wedge E_L\ \frac{\Gamma \Rightarrow s \wedge t}{\Gamma \Rightarrow s} \qquad \wedge E_R\ \frac{\Gamma \Rightarrow s \wedge t}{\Gamma \Rightarrow t} \qquad \vee I_L\ \frac{\Gamma \Rightarrow s}{\Gamma \Rightarrow s \vee t}$$

$$\vee I_R\ \frac{\Gamma \Rightarrow t}{\Gamma \Rightarrow s \vee t} \qquad \vee E_{a,b}\ \frac{\Gamma \Rightarrow s \vee t \qquad \Gamma, a : s \Rightarrow u \qquad \Gamma, b : t \Rightarrow u}{\Gamma \Rightarrow u}\ a, b \notin \mathrm{Dom}\,\Gamma$$

$$\forall I_y\ \frac{\Gamma \Rightarrow [sy]}{\Gamma \Rightarrow \forall_\sigma s}\ y : \sigma \text{ fresh for } \Gamma \Rightarrow s \qquad \forall E\ \frac{\Gamma \Rightarrow \forall_\sigma s}{\Gamma \Rightarrow [st]} \qquad \exists I\ \frac{\Gamma \Rightarrow [st]}{\Gamma \Rightarrow \exists_\sigma s}$$

$$\exists E_{a,y}\ \frac{\Gamma \Rightarrow \exists_\sigma s \qquad \Gamma, a : [sy] \Rightarrow t}{\Gamma \Rightarrow t}\ a \notin \mathrm{Dom}\,\Gamma,\ y : \sigma \text{ fresh for } \Gamma \Rightarrow t$$

$$=I\ \frac{}{\Gamma \Rightarrow s = s} \qquad =E\ \frac{\Gamma \Rightarrow s = t \qquad \Gamma \Rightarrow [us]}{\Gamma \Rightarrow [ut]} \qquad \mathbf{Con}_a\ \frac{\Gamma, a : \neg s \Rightarrow \bot}{\Gamma \Rightarrow s}\ a \notin \mathrm{Dom}\,\Gamma$$

$$\mathbf{BE}\ \frac{\Gamma, \Rightarrow s \to t \qquad \Gamma \Rightarrow t \to s}{\Gamma \Rightarrow s \equiv t} \qquad \mathbf{FE}\ \frac{\Gamma \Rightarrow \forall x.[sx] = [tx]}{\Gamma \Rightarrow s =_{\sigma\tau} t}$$

Figure 12.6: The natural deduction system $\mathcal{N}_{\mathrm{STT}}$ with explicit contexts

13 Proof Terms

In this chapter we consider proof terms corresponding to proofs in natural deduction. We will first associate derivations in $\mathcal{N}_{\forall\to}$ with proof terms. Once we have a notion of proof term, we can discuss equivalence of proofs. We will also discuss proof checking. That is, we can give an algorithm which determines if a given proof term corresponds to a proof of a given sequent $\Gamma \Rightarrow s$.

In a context, a proof term may have a formula as its type. The idea of having formulas play the role of types is referred to as the Curry-Howard correspondence or Curry-Howard isomorphism [44, 24]. The mathematician de Bruijn was the driving force behind the Automath project [22, 69] which designed languages and proof checkers based on similar principles. For this reason the idea is sometimes called the Curry-Howard-de Bruijn correspondence.

13.1 Proof Terms

Recall that in Chapter 3 we first defined a set of untyped terms before restricting ourselves to those terms that are well-formed and hence have a type. For untyped terms we defined substitution, $\sim_\alpha$ and $\sim_\beta$. We will reuse all these definitions by defining proof terms as certain untyped λ-terms.

When we introduced typed names, we assumed that our set of names Nam is such that $\text{Nam} \cong \mathbb{N} \times \text{Ty}$. Consequently each name had a unique associated type and for each type there were infinitely many names with this type. For this chapter, we weaken the assumption to allow labels to be used as untyped names. Recall from Chapter 12 that Lab is a countably infinite set of labels. Assume $\text{Lab} \subseteq \text{Nam}$ and

$$(\text{Nam} - \text{Lab}) \cong \mathbb{N} \times \text{Ty}$$

We refer to names in Nam − Lab as **typed names**.

Once we have fixed a set of names, we obtain a set Ter of terms by induction (exactly as in Chapter 3).

1. Every name is a term.
2. If s and t are terms, then st is a term.
3. If x is a name and s is a term, then $\lambda x.s$ is a term.

With respect to these terms, we have well-defined notions of substitution, $\sim_\alpha$, $\rightarrow_\beta$, $\sim_\beta$ and so on.

Just as before, we have obtain a subset of well-formed terms. Each well-formed term has a unique type.

1. If x is a typed name with type σ, then x is a well-formed term with type σ.
2. If s is a well-formed term with type $\sigma\tau$ and t is a well-formed term with type σ, then st is a well-formed term with type τ.
3. If x is a typed name with type σ and t is a well-formed term with type τ, then $\lambda x.t$ is a well-formed term with type $\sigma\tau$.

From now on in this chapter, we revert to our use of s, t to range over well-formed terms and x, y to range over typed names. As before, some of the typed names are logical constants while the remaining typed names are variables.

Now that we consider labels as names, we can define a set Pf of **proof terms**. We will use $\mathcal{D}$ and $\mathcal{E}$ to range over proof terms. The set of proof terms is the least set of terms satisfying the following conditions.

1. If a is a label, then a is a proof term.
2. If $\mathcal{D}$ and $\mathcal{E}$ are proof terms, then $\mathcal{D}\mathcal{E}$ is a proof term.
3. If $\mathcal{D}$ is a proof term and s is a well-formed term of type σ, then $\mathcal{D}s$ is a proof term.
4. If a is a label and $\mathcal{D}$ is a proof term, then $\lambda a.\mathcal{D}$ is a proof term.
5. If $x : \sigma$ is a variable and $\mathcal{D}$ is a proof term, then $\lambda x.\mathcal{D}$ is a proof term.

Note that in our definition of proof terms, there are two ways to form a proof term using application and two ways to form a proof term using λ-abstraction. We can represent the set of proof terms as a grammar as follows.

$$\mathcal{D}, \mathcal{E} \in \mathrm{Pf} ::= a \mid \mathcal{D}\mathcal{E} \mid \mathcal{D}s \mid \lambda a.\mathcal{D} \mid \lambda x.\mathcal{D} \qquad \textbf{proof terms}$$

Again, since Pf is a subset of the set of all terms, we inherit the notions of substitution, $\sim_\alpha$, $\rightarrow_\beta$ and so on.

13.2 A Calculus with Proof Terms

A **proof term sequent** is a triple $\langle \Gamma, \mathcal{D}, t \rangle$ where Γ is a context, $\mathcal{D}$ is a proof term and t is a β-normal formula. We write such a proof term sequent as $\Gamma \Rightarrow \mathcal{D} : t$.

We say a variable $y : \sigma$ is **fresh for** $\Gamma \Rightarrow \mathcal{D} : t$ if y is fresh for $\Gamma \Rightarrow t$ and $y \notin \mathcal{N}\mathcal{D}$. We say a label a is **fresh for** $\Gamma \Rightarrow \mathcal{D} : t$ if $a \notin \mathrm{Dom}\,\Gamma \cup \mathcal{N}\mathcal{D}$.

We define a proof system $\mathcal{N}^p_{\forall\rightarrow}$ for proof term sequents corresponding to $\mathcal{N}_{\forall\rightarrow}$. The propositions of $\mathcal{N}^p_{\forall\rightarrow}$ are proof term sequents. The rules of $\mathcal{N}^p_{\forall\rightarrow}$ are given in Figure 13.1. The reader should compare this to the rules for $\mathcal{N}_{\forall\rightarrow}$ in

$$A_a\ \frac{}{\Gamma \Rightarrow a : s}\ a : s \in \Gamma \qquad \forall I_y\ \frac{\Gamma \Rightarrow \mathcal{D}^z_y : [sy]}{\Gamma \Rightarrow \lambda z.\mathcal{D} : \forall_\sigma s}\ y : \sigma \text{ fresh for } \Gamma \Rightarrow \lambda z.\mathcal{D} : \forall_\sigma s$$

$$\rightarrow I_a\ \frac{\Gamma, a : s \Rightarrow \mathcal{D}^b_a : t}{\Gamma \Rightarrow \lambda b.\mathcal{D} : s \rightarrow t}\ a \notin \mathrm{Dom}\,\Gamma \cup \mathcal{N}(\lambda b.\mathcal{D}) \qquad \forall E\ \frac{\Gamma \Rightarrow \mathcal{D} : \forall_\sigma s}{\Gamma \Rightarrow \mathcal{D}t : [st]}$$

$$\rightarrow E\ \frac{\Gamma \Rightarrow \mathcal{D} : s \rightarrow t \qquad \Gamma \Rightarrow \mathcal{E} : s}{\Gamma \Rightarrow \mathcal{D}\mathcal{E} : t}$$

Figure 13.1: $\mathcal{N}^p_{\forall\rightarrow}$ rule schemas

Figure 12.2. In fact, we use the same names for the corresponding rules in $\mathcal{N}_{\forall\rightarrow}$ and $\mathcal{N}^p_{\forall\rightarrow}$.

Note that in the two rules $\forall I_y$ and $\rightarrow I_a$ we allow for the possibility of changing the name of a λ-bound name in the proof term. In most cases, the following special cases of $\forall I_y$ and $\rightarrow I_a$ will suffice.

$$\forall I_y\ \frac{\Gamma \Rightarrow \mathcal{D} : [sy]}{\Gamma \Rightarrow \lambda y.\mathcal{D} : \forall_\sigma s}\ y : \sigma \text{ fresh for } \Gamma \Rightarrow s \qquad \rightarrow I_a\ \frac{\Gamma, a : s \Rightarrow \mathcal{D} : t}{\Gamma \Rightarrow \lambda a.\mathcal{D} : s \rightarrow t}\ a \notin \mathrm{Dom}\,\Gamma$$

We introduce two convenient notations. We write $\Gamma \vdash s$ if the sequent $\Gamma \Rightarrow s$ is $\mathcal{N}_{\forall\rightarrow}$-derivable. We write $\Gamma \vdash \mathcal{D} : s$ if the proof term sequent $\Gamma \Rightarrow \mathcal{D} : s$ is $\mathcal{N}^p_{\forall\rightarrow}$-derivable. When the context is empty we write $\vdash s$ or $\vdash \mathcal{D} : s$. An easy induction can be used to prove the following result.

Proposition 13.2.1 $\Gamma \vdash s$ iff there is some proof term $\mathcal{D}$ such that $\Gamma \vdash \mathcal{D} : s$.

We illustrate Proposition 13.2.1 by considering examples.

Example 13.2.2 Following Example 12.2.3 we can give two different proofs of $\Rightarrow p \rightarrow p \rightarrow p$. Again, we omit contexts.

$$\dfrac{\dfrac{\dfrac{a}{p}}{p \rightarrow p}\rightarrow I_b}{p \rightarrow p \rightarrow p}\rightarrow I_a \qquad \dfrac{\dfrac{\dfrac{b}{p}}{p \rightarrow p}\rightarrow I_b}{p \rightarrow p \rightarrow p}\rightarrow I_a$$

The corresponding $\mathcal{N}^p_{\forall\rightarrow}$ derivations reveal the related proof terms.

$$\dfrac{\dfrac{a : p}{\lambda b.a : p \rightarrow p}\rightarrow I_b}{\lambda ab.a : p \rightarrow p \rightarrow p}\rightarrow I_a \qquad \dfrac{\dfrac{b : p}{\lambda b.b : p \rightarrow p}\rightarrow I_b}{\lambda ab.b : p \rightarrow p \rightarrow p}\rightarrow I_a$$

Note that the two proof terms $\lambda ab.a$ and $\lambda ab.b$ are different. Of course, if we had simply used different labels than a and b, then we would obtain different proof terms. The important fact is that the two proof terms are not α-equivalent. That is, $\lambda ab.a \not\sim_\alpha \lambda ab.b$.

Are there any other proof terms $\mathcal{D}$ (neither α-equivalent to $\lambda ab.a$ nor $\lambda ab.b$) such that $\vdash \mathcal{D} : p \to p \to p$? Yes, there are. Consider the proof term $\lambda ab.(\lambda c.c)a$ and derivation below.

$$\dfrac{\dfrac{\dfrac{\dfrac{c : p}{\lambda c.c : p \to p}\to I_c \qquad a : p}{(\lambda c.c)a : p}\to E}{\lambda b.(\lambda c.c)a : p \to p}\to I_b}{\lambda ab.(\lambda c.c)a : p \to p \to p}\to I_a$$

This is a roundabout proof of $\Rightarrow p \to p \to p$. Also, notice that the proof term $\lambda ab.(\lambda c.c)a$ has a β-redex. If we reduce the proof term to β-normal form we obtain $\lambda ab.a$, one of the first two proof terms.

Up to $\sim_\alpha$, $\lambda ab.a$ and $\lambda ab.b$ are the only β-normal proof terms $\mathcal{D}$ such that $\vdash \mathcal{D} : p \to p \to p$? □

Exercise 13.2.3 Let $p : o$ be a variable. Find three β-normal proof terms $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$ none of which are α-equivalent to another such that

$$\vdash \mathcal{D}_i : (p \to p) \to p \to p$$

holds for $i \in \{1, 2, 3\}$.

Exercise 13.2.4 Let $p, q, r : o$ be variables. For each problem below, find a β-normal proof term $\mathcal{D}$ satisfying the given condition.

a) $\vdash \mathcal{D} : p \to q \to p$

b) $\vdash \mathcal{D} : (q \to r) \to (p \to q) \to p \to r$

c) $\vdash \mathcal{D} : (p \to q \to r) \to (p \to q) \to p \to r$

Exercise 13.2.5 For each of the following proof terms $\mathcal{D}$ find a formula s such that $\vdash \mathcal{D} : s$.

a) $\lambda a.a$

b) $\lambda xa.ax$ where $x : \sigma$.

c) $\lambda xa.axx$ where $x : \sigma$.

d) $\lambda ya.a(y \to y)(ay)$ where $y : o$.

Exercise 13.2.6 Find a proof term $\mathcal{D}$ and a formula s such that $\vdash (\mathcal{D}\mathcal{D}) : s$.

Exercise 13.2.7 Consider the proof term $(\lambda a.aa)(\lambda a.aa)$. Argue that there is no context Γ and formula s such that $\Gamma \vdash (\lambda a.aa)(\lambda a.aa) : s$.

Exercise 13.2.8 Let $r : \sigma\sigma o$ and $x, y, z : \sigma$ be variables and let a, b be distinct labels. Let Γ be the context with the two elements

- $a : \forall xy.rxy \to ryx$ and
- $b : \forall xyz.rxy \to ryz \to rxz$.

a) Find a β-normal proof term $\mathcal{D}$ such that $\Gamma \vdash \mathcal{D} : \forall xy.rxy \to rxx$.

b) Using your solution $\mathcal{D}$ to part a consider the proof term $\lambda xyc.\mathcal{D}yx(axyc)$. Find a formula s such that $\Gamma \vdash \lambda xyc.\mathcal{D}yx(axyc) : s$.

c) Compute the β-normal form of $\lambda xyc.\mathcal{D}yx(axyc)$.

13.3 Bidirectional Checking of Normal Proof Terms

We define two sets of proof term sequents by induction. We then reason that this gives a proof checking algorithm.

We write $\Gamma \vdash \mathcal{D} \Uparrow s$ when the proof term sequent $\Gamma \Rightarrow \mathcal{D} : s$ is in the first set. We write $\Gamma \vdash \mathcal{D} \Downarrow s$ when the proof term sequent $\Gamma \Rightarrow \mathcal{D} : s$ is in the second set. These two sets are defined inductively as follows.

1. If $\Gamma \vdash \mathcal{D}^x_y \Uparrow [sy]$ for a fresh variable y, then $\Gamma \vdash \lambda x.\mathcal{D} \Uparrow \forall_\sigma s$.
2. If $\Gamma, a : s \vdash \mathcal{D}^b_a \Uparrow t$ for a fresh label a, then $\Gamma \vdash \lambda b.\mathcal{D} \Uparrow s \to t$.
3. If $\Gamma \vdash \mathcal{D} \Downarrow s$, then $\Gamma \vdash \mathcal{D} \Uparrow s$.
4. If $a : s \in \Gamma$, then $\Gamma \vdash a \Downarrow s$.
5. If $\Gamma \vdash \mathcal{D} \Downarrow \forall_\sigma s$ and $t : \sigma$ is a term, then $\Gamma \vdash \mathcal{D}t \Downarrow [st]$.
6. If $\Gamma \vdash \mathcal{D} \Downarrow s \to t$ and $\Gamma \vdash \mathcal{E} \Uparrow s$, then $\Gamma \vdash \mathcal{D}\mathcal{E} \Downarrow t$.

We now describe the two sets as a proof checking algorithm for β-normal proof terms. Given $\Gamma \Rightarrow \mathcal{D} : s$, we need to be able to test whether or not $\Gamma \vdash \mathcal{D} \Uparrow s$ holds. As a mutually recursive procedure, we will also need to be able to ask whether there is some s such that $\Gamma \vdash \mathcal{D} \Downarrow s$ whenever $\mathcal{D}$ is β-normal concrete proof term. Recall that we call a term concrete if it is not a λ-abstraction.

We summarize the cases of the procedure. In the description below, we assume all proof terms are β-normal.

1. **Goal:** Test if $\Gamma \vdash \lambda z.\mathcal{D} \Uparrow s$. If s is not of the form $\forall t$, then $\Gamma \nvdash \lambda z.\mathcal{D} \Uparrow s$. If s is of the form $\forall t$, choose some fresh variable y and test if $\Gamma \vdash \mathcal{D}^z_y \Uparrow [ty]$.
2. **Goal:** Test if $\Gamma \vdash \lambda b.\mathcal{D} \Uparrow s$. If s is not of the form $t \to u$, then $\Gamma \nvdash \lambda b.\mathcal{D} \Uparrow s$. If s is of the form $t \to u$, choose some fresh label a and test if $\Gamma, a : t \vdash \mathcal{D}^b_a \Uparrow [u]$.

$$\frac{\Gamma \vdash \mathcal{D}^z_y \Uparrow [sy]}{\Gamma \vdash \lambda z.\mathcal{D} \Uparrow \forall_\sigma s}\ y : \sigma \text{ fresh} \qquad \frac{\Gamma, a : s \vdash \mathcal{D}^b_a \Uparrow t}{\Gamma \vdash \lambda b.\mathcal{D} \Uparrow s \to t}\ a \text{ fresh} \qquad \frac{\Gamma \vdash \mathcal{D} \Downarrow s}{\Gamma \vdash \mathcal{D} \Uparrow s}$$

$$\frac{}{\Gamma \vdash a \Downarrow s}\ a : s \in \Gamma \qquad \frac{\Gamma \vdash \mathcal{D} \Downarrow \forall_\sigma s}{\Gamma \vdash \mathcal{D}t \Downarrow [st]} \qquad \frac{\Gamma \vdash \mathcal{D} \Downarrow s \to t \qquad \Gamma \vdash \mathcal{E} \Uparrow s}{\Gamma \vdash \mathcal{D}\mathcal{E} \Downarrow t}$$

Figure 13.2: Proof checking algorithm

3. **Goal:** Test if $\Gamma \vdash \mathcal{D} \Uparrow s$ where $\mathcal{D}$ is concrete. Try to compute some t such that $\Gamma \vdash \mathcal{D} \Downarrow t$. If either no such t exists or s and t are different, then $\Gamma \nvdash \mathcal{D} \Uparrow s$. Otherwise $\Gamma \vdash \mathcal{D} \Uparrow s$.
4. **Goal:** Compute some s such that $\Gamma \vdash a \Downarrow s$. If $a \notin \mathrm{Dom}\,\Gamma$, then there is no such s. Otherwise, $a : s$ is in Γ (for a unique s) and we have $\Gamma \vdash a \Downarrow s$ for this s.
5. **Goal:** Compute some s such that $\Gamma \vdash \mathcal{D}t \Downarrow s$. First compute some term u such that $\Gamma \vdash \mathcal{D} \Downarrow u$. If no such u exists or u is not of the form $\forall_\sigma v$, then there is no such s. Otherwise, u has the form $\forall_\sigma v$ and we have $\Gamma \vdash \mathcal{D}t \Downarrow [vt]$ and so we take s to be $[vt]$.
6. **Goal:** Compute some s such that $\Gamma \vdash \mathcal{D}\mathcal{E} \Downarrow s$. First compute some term u such that $\Gamma \vdash \mathcal{D} \Downarrow u$. If no such u exists or u is not of the form $u_1 \to u_2$, then there is no such s. Otherwise, u has the form $u_1 \to u_2$. Check if $\Gamma \vdash \mathcal{E} \Uparrow u_1$. If $\Gamma \nvdash \mathcal{E} \Uparrow u_1$, then there is no such s. Otherwise, $\Gamma \vdash \mathcal{D}\mathcal{E} \Downarrow u_2$ and so we take s to be u_2.

We give a nice summarization of the six cases of these algorithm in Figure 13.2 by abusing our notation for rules.

Example 13.3.1 Let $x, y : o$ be variables and a, b be distinct labels. We verify

$$\vdash \lambda abx.axb \Uparrow (\forall x.y \to x) \to y \to \forall x.x$$

Using the algorithm we reduce this to checking

$$a : (\forall x.y \to x), b : y \vdash axb \Uparrow x$$

Given this context and the proof term axb, we use the algorithm to compute that x is the formula such that

$$a : (\forall x.y \to x), b : y \vdash axb \Downarrow x$$

as desired. □

Exercise 13.3.2 Let $p : \sigma o, x, y : \sigma$ be variables and let $\mathcal{D}$ be the proof term

$$a(\lambda y.\forall p.py \to px)(\lambda pb.b)$$

where b is a label different from a. Compute the formula s such that

$$a : \forall p.px \to py \vdash \mathcal{D} \Downarrow s$$

holds. Give the corresponding $\mathcal{N}_{\forall\to}$-derivation.

13.4 Proof Terms as Programs

Let s, t be β-normal formulas. We can think of s and t as types and $s \to t$ as the type of functions taking data of type s to data of type t. To avoid confusion with our types σ, let us use the word datatype for such β-normal formulas.

Suppose we have proof terms $\lambda a.\mathcal{D}$ and $\mathcal{E}$ such that $\vdash \lambda a.\mathcal{D} : s \to t$ and $\vdash \mathcal{E} : s$. This means $\lambda a.\mathcal{D}$ has the datatype of a function from type s to type t. Since $\mathcal{E}$ has type s, we have $\vdash (\lambda a.\mathcal{D})\mathcal{E} : t$. This is a β-redex with reduct $\mathcal{D}^a_{\mathcal{E}}$. In this way β-normal formulas give the datatype of certain functional programs and β-reduction of proof terms corresponds to computation of these programs.

In addition to function datatypes $s \to t$, we also have product and sum datatypes. We obtain these constructions using Girard's expression of conjunction and disjunction in terms of implication and universal quantification:

$$x \wedge y \equiv \forall p.(x \to y \to p) \to p$$

$$x \vee y \equiv \forall p.(x \to p) \to (y \to p) \to p$$

Let

$$\times \text{ be the term } \lambda xy.\forall p.(x \to y \to p) \to p$$

and

$$+ \text{ be the term } \lambda xy.\forall p.(x \to p) \to (y \to p) \to p.$$

We write $\times$ and $+$ in infix notation. Note that the Girard equivalences can be expressed as

$$x \wedge y \equiv [x \times y] \text{ and } x \vee y \equiv [x + y]$$

Let s, t be β-normal formulas.

- Let pair be the proof term $\lambda abpc.cab$. The reader can check that

$$\vdash \mathsf{pair} \Uparrow s \to t \to [s \times t]$$

- Let fst be the proof term $\lambda c.cs(\lambda ab.a)$. The reader can check that

$$\vdash \mathsf{fst} \Uparrow [s \times t] \to s$$

- Let snd be the proof term $\lambda c.ct(\lambda ab.b)$. The reader can check that

$$\vdash \mathsf{snd} \Uparrow [s \times t] \to t$$

Let $\to_\beta^*$ be the reflexive transitive closure of $\to_\beta$. We claim that for any proof terms $\mathcal{D}$ and $\mathcal{E}$,

$$\mathsf{fst}\,(\mathsf{pair}\,\mathcal{D}\mathcal{E}) \to_\beta^* \mathcal{D}$$

and

$$\mathsf{snd}\,(\mathsf{pair}\,\mathcal{D}\mathcal{E}) \to_\beta^* \mathcal{E}$$

We show the first case and leave the second case to the reader.

$$\mathsf{fst}\,(\mathsf{pair}\,\mathcal{D}\mathcal{E}) \to_\beta^* (\mathsf{pair}\,\mathcal{D}\mathcal{E})s(\lambda ab.a) \to_\beta^* (\lambda ab.a)\mathcal{D}\mathcal{E} \to_\beta^* \mathcal{D}$$

Exercise 13.4.1 Check the following facts:

a) $\vdash \mathsf{pair} \Uparrow s \to t \to [s \times t]$

b) $\vdash \mathsf{fst} \Uparrow [s \times t] \to s$

c) $\vdash \mathsf{snd} \Uparrow [s \times t] \to t$

d) $\mathsf{snd}(\mathsf{pair}\mathcal{D}\mathcal{E}) \to_\beta^* \mathcal{E}$

We next turn to sum datatypes. Again, we fix β-normal formulas s and t.

- Let inl be the proof term $\lambda apbc.ba$. The reader can check that

$$\vdash \mathsf{inl} \Uparrow s \to [s + t]$$

- Let inr be the proof term $\lambda apbc.ca$. The reader can check that

$$\vdash \mathsf{inr} \Uparrow t \to [s + t]$$

- For each β-normal formula u let case_u be the proof term $\lambda abc.aubc$. The reader can check that

$$\vdash \mathsf{case}_u \Uparrow [s + t] \to (s \to u) \to (t \to u) \to u$$

We claim that for any proof terms $\mathcal{D}$, $\mathcal{E}_1$ and $\mathcal{E}_2$,

$$\mathsf{case}_u\,(\mathsf{inl}\,\mathcal{D})\mathcal{E}_1\mathcal{E}_2 \to_\beta^* \mathcal{E}_1\mathcal{D}$$

and

$$\mathsf{case}_u\,(\mathsf{inr}\,\mathcal{D})\mathcal{E}_1\mathcal{E}_2 \to_\beta^* \mathcal{E}_2\mathcal{D}$$

We leave these computations as exercises.

Exercise 13.4.2 Check the following facts:

a) $\vdash \mathsf{inl} \Uparrow s \to [s + t]$

b) $\vdash \mathsf{inr} \Uparrow t \to [s + t]$

c) $\vdash \mathsf{case}_u \Uparrow [s + t] \to (s \to u) \to (t \to u) \to u$

d) $\mathsf{case}_u\,(\mathsf{inl}\,\mathcal{D})\mathcal{E}_1\mathcal{E}_2 \to^*_\beta \mathcal{E}_1\mathcal{D}$

e) $\mathsf{case}_u\,(\mathsf{inr}\,\mathcal{D})\mathcal{E}_1\mathcal{E}_2 \to^*_\beta \mathcal{E}_2\mathcal{D}$

13.5 Adding Signatures

Assumption contexts allow us to reason with a finite dynamic set of assumptions. In this section we model a static set of axioms which may be infinite. We do this using the notion of a signature. A signature is similar to an assumption context, except that it may be infinite.

A **signature** Σ is a function such that every member is a pair $a : s$ where $a \in \mathrm{Lab}$ and s is a β-normal formula with no free variables (i.e., $\mathcal{V}s = \emptyset$). We call a signature Σ **admissible** if $\mathrm{Lab} - \mathrm{Dom}\,\Sigma$ is infinite. We only consider admissible signatures Σ so that we always have an infinite supply of labels that are fresh with respect to the Σ.

We can now generalize the notion of a proof term sequent to include a signature. A **proof term sequent over Σ** is a 4-tuple written as $\Gamma \Rightarrow_\Sigma \mathcal{D} : s$ where Σ is an admissible signature, Γ is an assignment context with $\mathrm{Dom}\,\Sigma \cap \mathrm{Dom}\,\Gamma = \emptyset$, $\mathcal{D}$ is a proof term and s is a β-normal formula. The cases we have considered previously are special cases where the signature is empty. When the intended signature is clear we may still sometimes write $\Gamma \Rightarrow \mathcal{D} : s$ for $\Gamma \Rightarrow_\Sigma \mathcal{D} : s$.

We can now generalize our proof system $\mathcal{N}^p_{\forall\to}$ to be a proof system $\mathcal{N}^p_\Sigma$ for proof term sequents over Σ in the obvious way. The rules for $\mathcal{N}^p_\Sigma$ are given in Figure 13.3. These are essentially the same as the rules for $\mathcal{N}^p_{\forall\to}$ in Figure 13.1 except for two modifications. First, freshness of a label a also means that $a \notin \mathrm{Dom}\,\Sigma$. Second, there is a new base case for deriving $\Gamma \Rightarrow_\Sigma a : s$ when $a : s \in \Sigma$.

We write $\Gamma \vdash_\Sigma \mathcal{D} : s$ where $\Gamma \Rightarrow_\Sigma \mathcal{D} : s$ is $\mathcal{N}^p_\Sigma$ derivable.

We now generalize the proof checking algorithm to perform checking given a signature Σ. As before we define two sets by mutual recursion. This time both sets are sets of proof term sequents over over Σ. We write $\Gamma \vdash_\Sigma \mathcal{D} \Uparrow s$ when $\Gamma \Rightarrow_\Sigma \mathcal{D} : s$ is in the first set and write $\Gamma \vdash_\Sigma \mathcal{D} \Downarrow s$. when $\Gamma \Rightarrow_\Sigma \mathcal{D} : s$ is in the second set. The cases for the algorithm are given in Figure 13.4.

The algorithm works as before. If Σ, Γ, $\mathcal{D}$ (β-normal) and s are given, we can check if $\Gamma \vdash_\Sigma \mathcal{D} \Uparrow s$ holds. If Σ, Γ and $\mathcal{D}$ (β-normal concrete), then we can compute an s (if one exists) such that $\Gamma \vdash_\Sigma \mathcal{D} \Downarrow s$ holds.

The following propositions ensure correctness of the algorithm.

$$S_a\ \frac{}{\Gamma \Rightarrow_\Sigma a : s}\ a : s \in \Sigma \qquad A_a\ \frac{}{\Gamma \Rightarrow_\Sigma a : s}\ a : s \in \Gamma$$

$$\forall I_y\ \frac{\Gamma \Rightarrow_\Sigma \mathcal{D}^z_y : [sy]}{\Gamma \Rightarrow_\Sigma \lambda z.\mathcal{D} : \forall_\sigma s}\ y : \sigma \text{ fresh for } \Gamma \Rightarrow_\Sigma \lambda z.\mathcal{D} : \forall_\sigma s$$

$$\to I_a\ \frac{\Gamma, a : s \Rightarrow_\Sigma \mathcal{D}^b_a : t}{\Gamma \Rightarrow_\Sigma \lambda b.\mathcal{D} : s \to t}\ a \notin \operatorname{Dom}\Sigma \cup \operatorname{Dom}\Gamma \cup \mathcal{N}(\lambda b.\mathcal{D}) \qquad \forall E\ \frac{\Gamma \Rightarrow_\Sigma \mathcal{D} : \forall_\sigma s}{\Gamma \Rightarrow_\Sigma \mathcal{D}t : [st]}$$

$$\to E\ \frac{\Gamma \Rightarrow_\Sigma \mathcal{D} : s \to t \qquad \Gamma \Rightarrow_\Sigma \mathcal{E} : s}{\Gamma \Rightarrow_\Sigma \mathcal{D}\mathcal{E} : t}$$

Figure 13.3: $\mathcal{N}^p_\Sigma$ rule schemas

Proposition 13.5.1 Let Σ be an admissible signature, Γ be an assumption context with $\operatorname{Dom}\Sigma \cap \operatorname{Dom}\Gamma = \emptyset$ and $\mathcal{D}$ be a β-normal concrete proof term. There is at most one formula s such that $\Gamma \vdash_\Sigma \mathcal{D} \Downarrow s$ holds.

Proof This follows by induction on the definition of $\Gamma \vdash_\Sigma \mathcal{D} \Downarrow s$. The base cases follow by our assumptions on Σ and Γ. The inductive cases are easy. ∎

Proposition 13.5.2

1. If $\Gamma \vdash_\Sigma \mathcal{D} \Uparrow s$ holds, then $\mathcal{D}$ is β-normal and $\Gamma \vdash_\Sigma \mathcal{D} : s$ holds.
2. If $\Gamma \vdash_\Sigma \mathcal{D} \Downarrow s$ holds, then $\mathcal{D}$ is β-normal and concrete and $\Gamma \vdash_\Sigma \mathcal{D} : s$ holds.

Proof These two facts follow from an easy mutual induction on the definitions of $\Gamma \vdash_\Sigma \mathcal{D} \Uparrow s$ and $\Gamma \vdash_\Sigma \mathcal{D} \Downarrow s$. ∎

Proposition 13.5.3

1. If $\mathcal{D}$ is β-normal and $\Gamma \vdash_\Sigma \mathcal{D} : s$ holds, then $\Gamma \vdash_\Sigma \mathcal{D} \Uparrow s$ holds.
2. If $\mathcal{D}$ is β-normal and concrete and $\Gamma \vdash_\Sigma \mathcal{D} : s$ holds, then $\Gamma \vdash_\Sigma \mathcal{D} \Downarrow s$ holds.

Proof The two facts follow by induction on the definition of $\Gamma \vdash_\Sigma \mathcal{D} : s$. ∎

13.6 Proof Terms for $\mathcal{N}_{\mathrm{STT}}$

By giving an appropriate signature Σ_{STT} we can assign proof terms to all $\mathcal{N}_{\mathrm{STT}}$ natural deduction proofs. This will give us a notion of proof term that corresponds to a complete natural deduction calculus. To accomplish this, we choose

$$\frac{\Gamma \vdash_\Sigma \mathcal{D}^z_y \Uparrow [sy]}{\Gamma \vdash_\Sigma \lambda z.\mathcal{D} \Uparrow \forall_\sigma s}\, y : \sigma \text{ fresh} \qquad \frac{\Gamma, a : s \vdash_\Sigma \mathcal{D}^b_a \Uparrow t}{\Gamma \vdash_\Sigma \lambda b.\mathcal{D} \Uparrow s \to t}\, a \text{ fresh} \qquad \frac{\Gamma \vdash_\Sigma \mathcal{D} \Downarrow s}{\Gamma \vdash_\Sigma \mathcal{D} \Uparrow s}$$

$$\frac{}{\Gamma \vdash_\Sigma a \Downarrow s}\, a : s \in \Sigma \qquad \frac{}{\Gamma \vdash_\Sigma a \Downarrow s}\, a : s \in \Gamma \qquad \frac{\Gamma \vdash_\Sigma \mathcal{D} \Downarrow \forall_\sigma s}{\Gamma \vdash_\Sigma \mathcal{D}t \Downarrow [st]}$$

$$\frac{\Gamma \vdash_\Sigma \mathcal{D} \Downarrow s \to t \qquad \Gamma \vdash_\Sigma \mathcal{E} \Uparrow s}{\Gamma \vdash_\Sigma \mathcal{D}\mathcal{E} \Downarrow t}$$

Figure 13.4: Proof checking algorithm with signatures

a fixed an infinite number of labels which we will use to model the rules in $\mathcal{N}_{\mathrm{STT}}$ but not in $\mathcal{N}_{\forall\to}$. Of course we make sure to leave infinitely many labels untouched so that the signature will be admissible. For each of the fixed labels we pair it with a β-normal formula with no free variables. Each such formula will correspond to a rule.

- Fix $\mathsf{truei} \in$ Lab. This will correspond to $\top I$.
- Fix $\mathsf{false} \in$ Lab. This will correspond to $\bot E$.
- Fix $\mathsf{noti} \in$ Lab. This will correspond to $\neg I$.
- Fix $\mathsf{note} \in$ Lab. This will correspond to $\neg E$.
- Fix $\mathsf{andi}, \mathsf{andel}, \mathsf{ander} \in$ Lab. These will correspond to $\wedge I$, $\wedge E_L$ and $\wedge E_R$.
- Fix $\mathsf{oril}, \mathsf{orir}, \mathsf{ore} \in$ Lab. These will correspond to $\vee I_L$, $\vee I_R$ and $\vee E$.
- For each type σ, fix $\mathsf{exi}_\sigma, \mathsf{exe}_\sigma \in$ Lab. These will correspond to $\exists I$ and $\exists E$.
- Fix each type σ, fix $\mathsf{eqi}_\sigma, \mathsf{eqe}_\sigma \in$ Lab. These will correspond to $=I$ and $=E$.
- Fix $\mathsf{con} \in$ Lab. This will correspond to **Con**.
- Fix $\mathsf{be} \in$ Lab. This will correspond to **BE**.
- For all types $\sigma\tau$, fix $\mathsf{fe}_{\sigma\tau} \in$ Lab. This will correspond to **FE**.

Each label will be used to give proof terms for corresponding rules. Let Σ_{STT} be the signature with the labels above associated with formulas as listed in Table 13.1. We also show the corresponding rules.

Finally, we show how one can assign Σ_{STT} proof terms to $\mathcal{N}_{\mathrm{STT}}$ derivations in Figures 13.5 and 13.6. Note that the rules in these figures do not *define* the set of derivable proof term sequents over Σ_{STT}. This set has already been defined in Figure 13.3. Figures 13.5 and 13.6 indicate how to annotate $\mathcal{N}_{\mathrm{STT}}$ derivations to obtain appropriate proof terms.

Label	Formula	Corresponding Rule
truei	$\top$	$\top I$
false	$\forall p.\bot \to p$	$\bot E$
noti	$\forall p.(p \to \bot) \to \neg p$	$\neg I$
note	$\forall p.\neg p \to p \to \bot$	$\neg E$
andi	$\forall pq.p \to q \to p \wedge q$	$\wedge I$
andel	$\forall pq.p \wedge q \to p$	$\wedge E_L$
ander	$\forall pq.p \wedge q \to q$	$\wedge E_R$
oril	$\forall pq.p \to p \vee q$	$\vee I_L$
orir	$\forall pq.q \to p \vee q$	$\vee I_R$
ore	$\forall pqr.p \vee q \to (p \to r) \to (q \to r) \to r$	$\vee E$
exi_σ	$\forall px.px \to \exists_\sigma p$	$\exists I$
exe_σ	$\forall pq.\exists_\sigma p \to (\forall x.px \to q) \to q$	$\exists E$
eqi_σ	$\forall x.x =_\sigma x$	$=I$
eqe_σ	$\forall xyp.x =_\sigma y \to px \to py$	$=E$
con	$\forall p.(\neg p \to \bot) \to p$	**Con**
be	$\forall pq.(p \to q) \to (q \to p) \to p \equiv q$	**BE**
$\text{fe}_{\sigma\tau}$	$\forall fg.(\forall x.fx = gx) \to f =_{\sigma\tau} g$	**FE**

Table 13.1: Labeled formulas in Σ_{STT}

Not all proof terms will correspond directly to $\mathcal{N}_{\text{STT}}$ derivations. For an easy example, consider the proof term $\mathsf{false}\,\top$. We clearly have $\vdash_{\Sigma_{\text{STT}}} \mathsf{false}\,\top : \bot \to \top$. On the other hand, any $\mathcal{N}_{\text{STT}}$ derivation of $\Rightarrow \bot \to \top$ must use $\to I$.

Example 13.6.1 Let $p : o$ be a variable. An $\mathcal{N}_{\text{STT}}$ derivation of $\Rightarrow p \vee \neg p$ (making a vital use of the rule **Con**) is given below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(p \vee \neg p)}\,a \qquad \dfrac{\dfrac{}{p}\,b}{p \vee \neg p}\vee I_L}{\bot}\neg E}{\neg p}\neg I_b \qquad \dfrac{}{\neg(p \vee \neg p)}\,a}{p \vee \neg p}\vee I_R}{\bot}\neg E}{p \vee \neg p}\mathbf{Con}_a$$

The corresponding proof term $\mathcal{D}$ is

$$\mathsf{con}\,(p \vee \neg p)\lambda a.\mathcal{E}$$

where $\mathcal{E}$ is

$$\mathsf{note}\,(p \vee \neg p)a(\mathsf{orir}\,p(\neg p)(\mathsf{noti}\,p\lambda b.\mathsf{note}\,(p \vee \neg p)a(\mathsf{oril}\,p(\neg p)b)))$$

$$\top I\ \frac{}{\mathsf{truei} : \top} \qquad \bot E\ \frac{\mathcal{D} : \bot}{\mathsf{false}\, s\mathcal{D} : s} \qquad \neg I_a\ \frac{\begin{array}{c}a\\ s\\ \vdots\\ \mathcal{D} : \bot\end{array}}{\mathsf{noti}\, s(\lambda a.\mathcal{D}) : \neg s}\ a \text{ fresh}$$

$$\neg E\ \frac{\mathcal{D} : \neg s \qquad \mathcal{E} : s}{\mathsf{note}\, s\mathcal{D}\mathcal{E} : \bot} \qquad \to I_a\ \frac{\begin{array}{c}a\\ s\\ \vdots\\ \mathcal{D} : t\end{array}}{\lambda a.\mathcal{D} : s \to t}\ a \text{ fresh} \qquad \to E\ \frac{\mathcal{D} : s \to t \qquad \mathcal{E} : s}{\mathcal{D}\mathcal{E} : t}$$

$$\wedge I\ \frac{\mathcal{D} : s \qquad \mathcal{E} : t}{\mathsf{andi}\, st\mathcal{D}\mathcal{E} : s \wedge t} \qquad \wedge E_L\ \frac{\mathcal{D} : s \wedge t}{\mathsf{andel}\, st\mathcal{D} : s} \qquad \wedge E_R\ \frac{\mathcal{D} : s \wedge t}{\mathsf{ander}\, st\mathcal{D} : t}$$

$$\vee I_L\ \frac{\mathcal{D} : s}{\mathsf{oril}\, st\mathcal{D} : s \vee t} \qquad \vee I_R\ \frac{\mathcal{D} : t}{\mathsf{orir}\, st\mathcal{D} : s \vee t}$$

$$\vee E_{a,b}\ \frac{\mathcal{D} : s \vee t \qquad \begin{array}{c}a\\ s\\ \vdots\\ \mathcal{E}_1 : u\end{array} \qquad \begin{array}{c}b\\ t\\ \vdots\\ \mathcal{E}_2 : u\end{array}}{\mathsf{ore}\, stu\mathcal{D}(\lambda a.\mathcal{E}_1)(\lambda b.\mathcal{E}_2) : u}\ a, b \text{ fresh} \qquad \forall I_y\ \frac{\mathcal{D} : [sy]}{\lambda y.\mathcal{D} : \forall_\sigma s}\ y : \sigma \text{ fresh}$$

$$\forall E\ \frac{\mathcal{D} : \forall_\sigma s}{\mathcal{D}t : [st]} \qquad \exists I\ \frac{\mathcal{D} : [st]}{\mathsf{exi}_\sigma\, st\mathcal{D} : \exists_\sigma s} \qquad \exists E_{a,y}\ \frac{\mathcal{D} : \exists_\sigma s \qquad \begin{array}{c}a\\ [sy]\\ \vdots\\ \mathcal{E} : t\end{array}}{\mathsf{exe}_\sigma\, st\mathcal{D}(\lambda y a.\mathcal{E}) : t}\ a, y \text{ fresh}$$

$$=I\ \frac{}{\mathsf{eqi}_\sigma s : s =_\sigma s} \qquad =E\ \frac{\mathcal{D} : s =_\sigma t \qquad \mathcal{E} : [us]}{\mathsf{eqe}_\sigma\, stu\mathcal{D}\mathcal{E} : [ut]}$$

Figure 13.5: Obtaining proof terms from $\mathcal{N}_{\mathrm{STT}}$ derivations, part 1

$$\textbf{Con}_a\ \frac{\begin{array}{c}a\\ \neg s\\ \vdots\\ \mathcal{D} : \bot\end{array}}{\mathsf{con}\, s\, (\lambda a.\mathcal{D}) : s}\ a \text{ fresh} \qquad \textbf{BE}\ \frac{\mathcal{D} : s \to t \qquad \mathcal{E} : t \to s}{\mathsf{be}\, st\mathcal{D}\mathcal{E} : s \equiv t}$$

$$\textbf{FE}\ \frac{\mathcal{D} : \forall x.[sx] = [tx]}{\mathsf{fe}_{\sigma\tau}\, st\mathcal{D} : s =_{\sigma\tau} t}$$

Figure 13.6: Obtaining proof terms from $\mathcal{N}_{\mathrm{STT}}$ derivations, part 2

The reader should verify that

$$\vdash_{\Sigma_{\mathrm{STT}}} \mathcal{D} \Uparrow p \vee \neg p$$

in fact holds for this proof term. □

Exercise 13.6.2 Let $f : oo$ be a variable. Construct an $\mathcal{N}_{\mathrm{STT}}$ derivation of $\Rightarrow \exists f.f\bot \wedge \neg f\top$ and then find a corresponding proof term $\mathcal{D}$ such that

$$\vdash_{\Sigma_{\mathrm{STT}}} \mathcal{D} \Uparrow \exists f.f\bot \wedge \neg f\top$$

Exercise 13.6.3 Let $p : o$ be a variable. Construct an $\mathcal{N}_{\mathrm{STT}}$ derivation of $\Rightarrow \neg\exists p.p \wedge \neg p$ and then find a corresponding proof term $\mathcal{D}$ such that

$$\vdash_{\Sigma_{\mathrm{STT}}} \mathcal{D} \Uparrow \neg\exists p.p \wedge \neg p$$

Exercise 13.6.4 Let $p : o$ be a variable. Construct an $\mathcal{N}_{\mathrm{STT}}$ derivation of

$$\Rightarrow p \equiv \neg\neg p$$

and then find a corresponding proof term $\mathcal{D}$ such that

$$\vdash_{\Sigma_{\mathrm{STT}}} \mathcal{D} \Uparrow p \equiv \neg\neg p$$

13.7 Remarks

14 Decision Trees

In this chapter we study logical equivalence of propositional formulas. We observe that every propositional formula denotes a Boolean function and that two propositional formulas are logically equivalent if and only if they denote the same Boolean function. Boolean functions matter since they are used for the logical specification of computer circuits. We identify a class of canonical propositional formulas such that every propositional formula is logically equivalent to exactly one canonical formula. The canonical form is based on decision trees. Decision trees are represented as nodes of graphs known as BDDs (for binary decision diagrams). The data structures and algorithms presented in this chapter have many applications, including computer-aided design of computer hardware and verification of finite state systems. The essential ideas are due to Randal Bryant [56].

14.1 Boolean Functions

An important building block of the hardware of computers are functional circuits. A functional circuit maps some inputs $x_1, \ldots, x_m$ to some outputs $y_1, \ldots, y_n$. 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.

Let X be a nonempty and finite set of propositional variables. The following definitions are taken with respect to X.

An **assignment** (σ) is a function $X \to \mathbb{B}$. A **Boolean function** (φ) is a function $(X \to \mathbb{B}) \to \mathbb{B}$. 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 readable and 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:

$$\begin{aligned}
\mathcal{F}\bot\sigma &= 0\\
\mathcal{F}\top\sigma &= 1\\
\mathcal{F}x\sigma &= \sigma x\\
\mathcal{F}(\neg s)\sigma &= 1 - \mathcal{F}s\sigma\\
\mathcal{F}(s \wedge t)\sigma &= \min\{\mathcal{F}s\sigma, \mathcal{F}t\sigma\}\\
\mathcal{F}(s \vee t)\sigma &= \max\{\mathcal{F}s\sigma, \mathcal{F}t\sigma\}\\
\mathcal{F}(s \to t)\sigma &= \max\{1 - \mathcal{F}s\sigma, \mathcal{F}t\sigma\}\\
\mathcal{F}(s \equiv t)\sigma &= (\mathcal{F}s\sigma = \mathcal{F}t\sigma)
\end{aligned}$$

Proposition 14.1.1 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 14.1.2 Let $s, t \in \mathrm{PF}$. Then $\mathcal{F}s = \mathcal{F}t$ iff $s = t$ 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 14.1.1. ∎

Given Proposition 14.1.2, 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.

Proposition 14.1.3 Let $s, t \in \mathrm{PF}$. Then the following statements are equivalent:

1. s and t are logically equivalent.
2. $s = t$ is a tautology.
3. $\mathcal{F}s = \mathcal{F}t$.

Proof Follows with Propositions 4.5.2 and 14.1.2. ∎

$\mathcal{F}$ 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 logic equivalence: Two propositional formulas are logically equivalent if and only if they denote the 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 14.1.4 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 $\bot$. ∎

Exercise 14.1.5 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?

14.2 Decision Trees and Prime Trees

Every Boolean function can be represented by many different 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 states properties of conditionals that are familiar from programming.

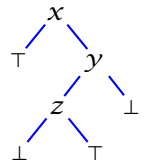
Proposition 14.2.1 The following formulas are valid:

1. $(\bot, x, y) \equiv x$
2. $(\top, x, y) \equiv y$
3. $(x, y, y) \equiv y$
4. $f(x,y,z) \equiv (x, fy, fz)$
5. $f(x,y,z)u \equiv (x, fyu, fzu)$
6. $f(x,y,z)(x,y',z') \equiv (x, fyy', fzz')$
7. $fx \equiv (x, f\bot, f\top)$

Decision trees are defined inductively:

1. $\bot$ and $\top$ are decision trees.
2. (x,s,t) is a decision tree if x is a propositional 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 $\mathcal{F}s\sigma$ for a decision tree s, we start at the root of s and follow the path determined by σ where $\sigma x = 0$ means "go left" and $\sigma x = 1$ means "go

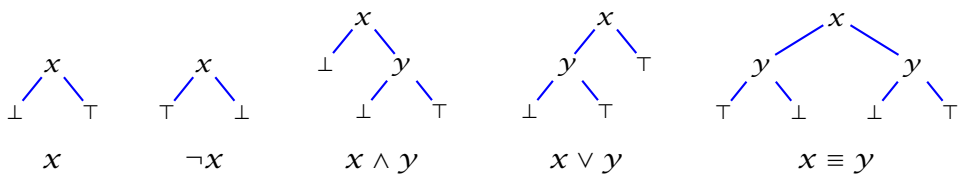


Figure 14.1: Prime trees for some simple formulas

right". If we reach a leaf, the result is determined by the label of the leaf (0 for $\bot$ and 1 for $\top$).

Proposition 14.2.2 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 14.2.3 (Coincidence)
Let $s \in \mathrm{PF}$ and $\sigma x = \sigma' x$ for all $x \in \mathcal{V}s$. Then $\mathcal{F}s\sigma = \mathcal{F}s\sigma'$.

Proof By induction on s. ∎

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 propositional 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 inductively:

1. $\bot$ 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 propositional variable that is smaller than every variable that occurs in s or t.

We will show that every propositional formula is logically equivalent to exactly one prime tree. Figure 14.1 shows the prime trees for some simple formulas.

Exercise 14.2.4 Find tableau proofs for the formulas in Proposition 14.2.1.

Exercise 14.2.5 Derive the formulas in Proposition 14.2.1 with basic deduction rules. Make use of BCR.

14.3 Existence of Prime Trees

First we outline an algorithm that computes for a propositional formula a logically equivalent prime tree. The algorithm is based on the following proposition.[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

Proposition 14.3.1 (Shannon Expansion) For every formula s and every propositional variable x the formula $s \equiv (x, s^x_\bot, s^x_\top)$ is valid.

Proof By Proposition 14.2.1 we know that $fx \equiv (x, f\bot, f\top)$ is valid. The rest follows with basic deduction. Substitution gives us the validity of $(\lambda x.s)x \equiv (x, (\lambda x.s)\bot, (\lambda x.s)\top)$. The claim follows with β and Replacement. ∎

The algorithm works by recursion on the number of propositional variables occurring in s. If s contains no propositional variables, s can be evaluated to $\bot$ or $\top$. Otherwise, the algorithm determines the least propositional 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. We show the correctness of the algorithm by proving the relevant properties.

A term s is **variable-free** if $\mathcal{V}s = \emptyset$.

Proposition 14.3.2 (Evaluation) Let s be a variable-free propositional formula. Then either the formula $s \equiv \bot$ or the formula $s \equiv \top$ is valid.

Proof Follows with logical coincidence (Proposition 4.3.1). ∎

Lemma 14.3.3 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 logically 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 logically equivalent to s.
2. If $s_0 \neq s_1$, then (x, s_0, s_1) is a prime tree that is logically equivalent to s.

Proof Follows with Proposition 14.3.1 and Proposition 14.2.1. ∎

Proposition 14.3.4
For every propositional formula s there exists a logically equivalent prime tree t such that $\mathcal{V}t \subseteq \mathcal{V}s$.

Proof Follows with Lemma 14.3.3 by induction on the number of variables occurring in s. ∎

Exercise 14.3.5 Draw all prime trees containing no other variables but x and y. Assume $x < y$. For each tree give a logically equivalent propositional formula that is as simple as possible.

Exercise 14.3.6 Let s be the propositional formula $x = (y = z)$. Assume $x < y < z$. Draw the prime trees for the following formulas: s, $\neg s$, $s \wedge s$, $s \to s$.

Boolean algebra.

14.4 Example: Diet Rules

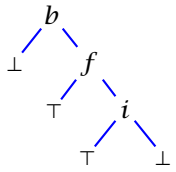
On a TV show a centenarian is asked for the secret of his long life. Oh, he says, my long life is due to a special diet that I started 60 years ago and follow by every day. The presenter gets all excited and asks for the diet. Oh, that's easy, says the old gentleman, there are only 3 rules you have to 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 propositional 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 possible description of the diet. A more abstract representation of the diet is the Boolean function decribed by the formula. Yet another representation of the diet is the prime tree that is logically equivalent to the initial formula:



The prime tree is more explicit than the initial formula. It tells us 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 14.4.1 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 logically equivalent to the conjunction of the rules. Use the order $c < p < m < r$.

14.5 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 14.5.1 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 $x \notin \mathcal{V}t$. 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 $x \notin \mathcal{V}s$. 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 14.5.2 (Prime Tree) For every propositional formula there exists exactly one logically equivalent prime tree.

Proof The existence follows with Proposition 14.3.4. To show the uniqueness, assume that s is a propositional formula and t_1, t_2 are different prime trees that are logically 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 14.5.1. Hence t_1, t_2 are not logically equivalent by Proposition 14.1.3. Contradiction since t_1, t_2 are both logically equivalent to s. ∎

14.6 Properties of Prime Trees

For every propositional formula s we denote the unique prime tree that is logically equivalent to s with πs. We call πs the **prime tree for** s.

Proposition 14.6.1 Let s and t be propositional formulas.

1. s is logically equivalent to πs.
2. $\mathcal{V}(\pi s) \subseteq \mathcal{V}s$.
3. s, t are logically 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 14.3.4 and Theorem 14.5.2. Claim (3) follows with (1) and Theorem 14.5.2. For Claim (4) first note that s is a tautology iff $s = \top$ is a tautology. By Proposition 14.1.3 this is the case iff s and $\top$ are logically equivalent. By Claim (3) this is the case iff $\pi s = \pi\top$. Now we are done since $\pi\top = \top$. ∎

A propositional variable is **significant for a propositional formula** if it occurs in the prime tree of the formula. A variable $x \in X$ is **significant for a Boolean function** φ if there exists an assignment σ such that $\varphi\sigma_0^x \neq \varphi\sigma_1^x$. We speak of the **significant variables** of propositional formulas and Boolean functions. We will show that the significant variables of a propositional formula are exactly the significant variables of the Boolean function described by the formula.

Proposition 14.6.2 If a propositional variable x is significant for a propositional formula s, then x occurs in s.

Proof Follows with Proposition 14.6.1. ∎

Lemma 14.6.3 Let $s \in \mathrm{PF}$, $x \in X$, and σ be an assignment.
Then $\mathcal{F}s\sigma_0^x = \mathcal{F}(s_\bot^x)\sigma$.

Proof By induction on s. ∎

Lemma 14.6.4 Let $s \in \mathrm{PF}$ be a prime tree and $x \in \mathcal{V}s$. Then there exists an assignment σ such that $\mathcal{F}s\sigma_0^x \neq \mathcal{F}s\sigma_1^x$.

Proof By contradiction. Assume $\mathcal{F}s\sigma_0^x = \mathcal{F}s\sigma_1^x$ for every assignment σ. Lemma 14.6.3 gives us $\mathcal{F}s\sigma = \mathcal{F}s\sigma_0^x = \mathcal{F}(s_\bot^x)\sigma$ for every assignment σ. Thus $\mathcal{F}s = \mathcal{F}(s_\bot^x)$ and hence $\pi s = \pi(s_\bot^x)$ by Propositions 14.1.3 and 14.6.1. Since s is a prime tree, we have $s = \pi s = \pi(s_\bot^x)$. Since $x \in \mathcal{V}s$ we have $x \in \mathcal{V}(\pi(s_\bot^x))$ and hence $x \in \mathcal{V}(s_\bot^x)$ by Proposition 14.6.1. Contradiction. ∎

Proposition 14.6.5 Let $s \in \mathrm{PF}$ and $x \in X$. Then x is significant for s if and only if x is significant for $\mathcal{F}s$.

Proof Since $\mathcal{F}s = \mathcal{F}(\pi s)$ we assume without loss of generality that s is a prime tree. The left-to-right direction follows with Lemma 14.6.4. To see the other direction, let $\mathcal{F}s\sigma_0^x \neq \mathcal{F}s\sigma_1^x$. By Coincidence we have $x \in \mathcal{V}s$. Since s is a prime tree, x is a significant variable of 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 propositional 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.

Exercise 14.6.6

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 $(x \to y) \wedge (x \vee y) \wedge (y \vee z)$.

14.7 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.

We formulate the algorithms for negation and conjunction as procedures that compute the following function:

$$\mathit{not} \in \mathrm{PT} \to \mathrm{PT} \qquad \mathit{and} \in \mathrm{PT} \to \mathrm{PT} \to \mathrm{PT}$$
$$\mathit{not}\, s = \pi(\neg s) \qquad \mathit{and}\, s\, t = \pi(s \wedge t)$$

Let DT be the set of all decision trees. Here is a procedure that computes *not*:

$$\begin{aligned} &\mathit{not} : \mathrm{PT} \to \mathrm{DT} \\ &\quad \mathit{not}\ \bot = \top \\ &\quad \mathit{not}\ \top = \bot \\ &\mathit{not}(x,s,t) = (x,\ \mathit{not}\ s,\ \mathit{not}\ t) \end{aligned}$$

It is easy to see that the defining equations of the procedure *not* are well-typed, exhaustive, and terminating. To show that the procedure *not* computes the function *not* it remains to show that the function *not* satisfies the defining equations of the procedure *not*. For the first two equations this is obvious. We argue that the function *not* satisfies the third equation $\mathit{not}(x,s,t) = (x,\ \mathit{not}\ s,\ \mathit{not}\ t)$. Given the definition of the function *not*, this boils down to showing that the equation

$$\pi(\neg(x,s,t)) = (x,\ \pi(\neg s),\ \pi(\neg t))$$

holds for every prime tree (x,s,t). Let (x,s,t) be a prime tree. By Propositions 14.2.1 and 14.6.1 we know that the propositional formula

$$\neg(x,s,t) \equiv (x,\ \pi(\neg s),\ \pi(\neg t))$$

is valid. Hence it suffices to show that $(x,\ \pi(\neg s),\ \pi(\neg t))$ is a prime tree. Since $\mathcal{V}(\pi(\neg s)) \subseteq \mathcal{V}s$, $\mathcal{V}(\pi(\neg t)) \subseteq \mathcal{V}t$ and (x,s,t) is a prime tree, we know that $(x,\ \pi(\neg s),\ \pi(\neg t))$ is an ordered decision tree. It remains to show that $\pi(\neg s)$ and $\pi(\neg t)$ are different. By Proposition 14.6.1 this is the case if $\neg s$ and $\neg t$ are not logically equivalent. Suppose $\neg s$ and $\neg t$ are logically equivalent. Then $\neg\neg s$ and $\neg\neg t$ and hence s and t are logically equivalent (compatibility amd double negation). Contradiction by Proposition 14.6.1 since $\pi s \neq \pi t$ since (x,s,t) is a prime tree.

Next we devise a procedure for conjunction. We base the procedure on the following tautologies (verify!):

$$\begin{aligned} \bot \wedge y &\equiv \bot \\ \top \wedge y &\equiv y \\ (x,y,z) \wedge (x,y',z') &\equiv (x, y \wedge y', z \wedge z') \\ (x,y,z) \wedge u &\equiv (x, y \wedge u, z \wedge u) \end{aligned}$$

Moreover, we exploit the commutativity of $\wedge$. We also use an auxiliary function

$$\begin{aligned} &\mathit{red} \in \mathrm{DT} \to \mathrm{DT} \\ &\mathit{red}\ \bot = \bot \\ &\mathit{red}\ \top = \top \\ &\mathit{red}(x,s,t) = \text{if } s = t \text{ then } s \text{ else } (x,s,t) \end{aligned}$$

Next we verify that the following equations hold:

$$\begin{aligned}
\pi(\bot \wedge t) &= \bot \\
\pi(\top \wedge t) &= t \\
\pi((x, s_0, s_1) \wedge (x, t_0, t_1)) &= \mathit{red}(x, \pi(s_0 \wedge t_0), \pi(s_1 \wedge t_1)) \\
\pi((x, s_0, s_1) \wedge t) &= \mathit{red}(x, \pi(s_0 \wedge t), \pi(s_1 \wedge t)) \\
&\qquad \text{if } t = (y, t_0, t_1) \text{ and } x < y
\end{aligned}$$

The correctness of the equations is established in 2 steps. First one verifies that for each equation the formula on the left is logically equivalent to the formula on the right. Since π and *red* yield logically 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:

$$\begin{aligned}
&\mathit{and} : \mathrm{PT} \to \mathrm{PT} \to \mathrm{DT} \\
&\mathit{and}\ \bot\ t = \bot \\
&\mathit{and}\ \top\ t = t \\
&\mathit{and}\ s\ \bot = \bot \\
&\mathit{and}\ s\ \top = s \\
&\mathit{and}\ (x, s_0, s_1)\ (x, t_0, t_1) = \mathit{red}(x,\ \mathit{and}\ s_0\ t_0,\ \mathit{and}\ s_1\ t_1) \\
&\mathit{and}\ (x, s_0, s_1)\ t = \mathit{red}(x,\ \mathit{and}\ s_0\ t,\ \mathit{and}\ s_1\ t) \\
&\qquad \text{if } t = (y, t_0, t_1) \text{ and } x < y \\
&\mathit{and}\ s\ (y, t_0, t_1) = \mathit{red}(y,\ \mathit{and}\ s\ t_0,\ \mathit{and}\ s\ t_1) \\
&\qquad \text{if } s = (x, s_0, s_1) \text{ and } x > y
\end{aligned}$$

The procedure *and* computes the function *and* since the following properties are satisfied:

1. The defining equations are well-typed, exhaustive, and terminating.
2. The defining equations hold for the function *and*. This is the case since the equations reduce to the equations verified before (exploiting commutativity of $\wedge$) if we replace the functions *and* and *red* with their definitions.

You now know enough so that you can devise algorithms for the other Boolean operations. Things are as before since by Proposition 14.2.1 the following for-

mulas are valid for every name $\circ : ooo$ ($\circ$ is written as infix operator):

$$(x, y, z) \circ (x, y', z') \equiv (x, y \circ y', z \circ z')$$
$$u \circ (x, y', z') \equiv (x, u \circ y', u \circ z')$$
$$(x, y, z) \circ u \equiv (x, y \circ u, z \circ u)$$

Exercise 14.7.1 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.

$$(x \equiv \top) \equiv$$
$$(\bot \equiv \bot) \equiv$$
$$((x, y, z) \equiv (x, y', z')) \equiv$$
$$((x, y, z) \equiv u) \equiv$$

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 14.7.2 Let decision trees be represented as in Exercise 14.7.1, 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 $\bot$).

Exercise 14.7.3 Find two prime trees (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) $\forall y \in \mathcal{V}t\colon\ x < y$.

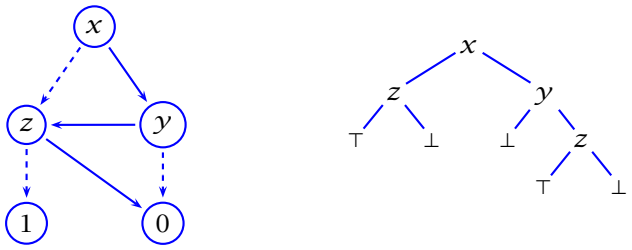


Figure 14.2: A BDD and the decision tree represented by the topmost node

14.8 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 14.2 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.

Let Var_o be the set of all propositional variables. Formally, a **BDD** is a function γ such that there exists a natural number $N \geq 1$ such that

1. $\gamma \in \{2,\ldots,N\} \to \mathrm{Var}_o \times \{0,\ldots,N\} \times \{0,\ldots,N\}$.
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 $0,\ldots,N$. The nodes 0 und 1 represent the decision trees $\bot$ 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 14.2 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 14.2 is minimal.

Proposition 14.8.1 (Minimality) A BDD is minimal if and only if it is injective.

Proof Let γ be a BDD such that $\mathrm{Dom}\,\gamma = \{2,\ldots,n\}$. 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 $\{2,\ldots,n-1\}$ 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 14.8.2 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 14.8.3 (Parity Function) Let the propositional 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 14.8.4 (Impact of Variable Order) The size of the BDD representing the parity function in Exercise 14.8.3 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 $n = 2$.

14.9 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 numbers of 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.

14.10 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 [56]. You will find his famous paper in the Internet. Huth and Ryan's textbook [45] gives a detailed presentation of BDDs. You will also find useful information in the Wikipedia entry for binary decision diagrams.

15 Modal Logic

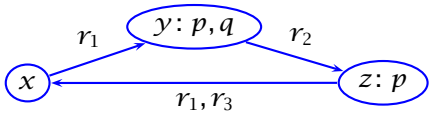
Modal logics are specialized logic languages that talk about transition systems. Modal logics have applications as ontology languages (description logics) in semantic web technology and as specification languages (temporal logics) in system verification. Modal logics are attractive for applications since they combine high expressivity with decidability. In this chapter we present a modal logic known as ALC. As we did for propositional logic and first-order logic, we present modal logic as a sublogic of simple type theory.

15.1 Transition Systems

Transition systems are labelled graphs that may serve as models of computational systems. A well-known class of transition systems are the finite-state automata used with regular string languages. There are many possible ways transition systems can be represented formally. We represent transition systems as sets of formulas.

A **transition system** is a nonempty set of formulas where each formula has the form rxy or px where $r : \iota\iota o$, $p : \iota o$, $x : \iota$ and $y : \iota$ range over variables. Given a transition system T, we call the variables $x \in \mathcal{V}_\iota T$ the **states of** T, and the formulas $rxy \in T$ the **transitions of** T.

We consider the transition system $\{r_1xy, r_2yz, r_1zx, r_3zx, py, qy, pz\}$ as an example. This system has 3 states and 4 transitions. It is illuminating to draw the transition system as a graph:



Seen from the graph perspective, the individual variables x, y, z act as nodes, the unary predicate variables p and q act as labels of nodes, and the binary predicate variables r_1, r_2, and r_3 act as labels of edges. The transition system has three states x, y, z, where x is unlabelled, y is labelled with p and q, and z is labelled with p. Moreover, the system has 4 transitions: An r_1-transition from x to y, an r_2-transition from y to z, and both and r_1- and an r_3-transition from z to x.

$$
\begin{aligned}
\dot{\bot} &= \lambda x.\, \bot \\
\dot{\top} &= \lambda x.\, \top \\
\dot{\neg} &= \lambda p x.\, \neg p x \\
\dot{\wedge} &= \lambda p q x.\, p x \wedge q x \\
\dot{\vee} &= \lambda p q x.\, p x \vee q x \\
\dot{\rightarrow} &= \lambda p q x.\, p x \rightarrow q x \\
\doteq &= \lambda p q x.\, p x \equiv q x \\
\Box &= \lambda r p x.\, \forall y.\, r x y \rightarrow p y \\
\Diamond &= \lambda r p x.\, \exists y.\, r x y \wedge p y
\end{aligned}
$$

Figure 15.1: Defining equations for the modal constants

Finite automata for strings can be seen as transition systems. In this case we would label edges with characters and distinguish initial and accepting states with labels. Given the above system, if q labels intial states and p label accepting states, the system accepts the strings "", "r_2", "$r_2 r_1 r_1$", "$r_2 r_3 r_1$", "$r_2 r_1 r_1 r_2$", "$r_2 r_3 r_1 r_2$", and so on.

Exercise 15.1.1 Draw the transition system $\{rxy, ryy, ryz, px, qy\}$.

15.2 Modal Constants and Modal Interpretations

Modal languages obtain their expressivity with λ-free terms of type ιo. Such terms, which we call *modal expressions*, describe unary predicates on the states of transition systems. Modal expressions are obtained with *modal constants* that hide λ-abstractions and quantifications. Syntactically, modal constants are names different from the logical constants (§ 4.3). We fix the following **modal constants**:

$$
\begin{aligned}
\dot{\bot}, \dot{\top} &: \iota o \\
\dot{\neg} &: (\iota o)\iota o \\
\dot{\wedge}, \dot{\vee}, \dot{\rightarrow}, \doteq &: (\iota o)(\iota o)\iota o \\
\Box, \Diamond &: (\iota\iota o)(\iota o)\iota o
\end{aligned}
$$

The semantics of the modal constants is given by the equations in Figure 15.1. These equations define the modal constants in terms of the logical constants.

The **modalities** $\Box$ and $\Diamond$ (read box and diamond) are specialized quantifiers that can be explained as follows:

- $\Box rpx$ is true if every r-successor of x satisfies p.
- $\Diamond rpx$ is true if there is an r-successor of x that satisfies p.

The other modal constants lift the propositional constants from truth values (type o) to properties (type ιo).

The modal constants $\dot{\wedge}$, $\dot{\vee}$, $\dot{\rightarrow}$, $\doteq$ are written with infix notation. Notationally, the modal constants $\doteq$, $\dot{\rightarrow}$, $\dot{\vee}$, $\dot{\wedge}$, $\dot{\neg}$ act as operators that take their arguments according to the following precedence hierarchy:

$\doteq$
$\dot{\rightarrow}$
$\dot{\vee}$
$\dot{\wedge}$
$\dot{\neg}$

Modal operators take their arguments after ordinary application and before the operators for the logical constants. The modal operators group to the right. For instance, the notation $\dot{\neg}\Box rt = \Diamond r(\Box r\dot{\neg}t)$ is to be read as $(\dot{\neg}(\Box rt)) = (\Diamond r(\Box r(\dot{\neg}t)))$.

From now on, a **constant** will be either a logical constant as introduced in § 4.3 or a modal constant as defined above. A **variable** will be any name that is not a constant.

A **modal interpretation** is a logical interpretation that satisfies the defining equations of the modal constants. When we talk about modal logic, we will only consider modal interpretations. A formula is **modally valid** if it is satisfied by every modal interpretation, and **modally satisfiable** if it is satisfied by some modal interpretation. Figure 15.2 shows some modally valid equations.

Since the modal constants are defined in terms of the logical constants, we can use our existing methods to prove that formulas are modally valid or satisfiable. Let M be the conjunction of the equations defining the modal constants (Figure 15.1).

Proposition 15.2.1 Let s be a formula. Then:

1. s is modally valid if and only if $M \rightarrow s$ is valid.
2. s is modally satisfiable if and only if $M \wedge s$ is satisfiable.

When we prove modal validity with tableaux, making use of the defining equations for the modal constants turns out to be tedious. Proofs become more pleasant if we use the sound tableau rule

$$\frac{s{=}t\,,\ [us]}{[ut]}$$

$$
\begin{aligned}
\dot\neg\dot\neg p &= p\\
\dot\neg(p \mathbin{\dot\wedge} q) &= \dot\neg p \mathbin{\dot\vee} \dot\neg q\\
\dot\neg(p \mathbin{\dot\vee} q) &= \dot\neg p \mathbin{\dot\wedge} \dot\neg q\\
\dot\neg\Box r p &= \Diamond r \dot\neg p\\
\dot\neg\Diamond r p &= \Box r \dot\neg p\\
\neg p x &\equiv (\dot\neg p)x\\
\neg\forall p &\equiv \exists(\dot\neg p)\\
\neg\exists p &\equiv \forall(\dot\neg p)\\
p \mathbin{\dot\to} q &= \dot\neg p \mathbin{\dot\vee} q\\
p \doteq q &= (p \mathbin{\dot\to} q) \mathbin{\dot\wedge} (q \mathbin{\dot\to} p)\\
p{=}q &\equiv \forall(p \doteq q)\\
r x y &\equiv \Diamond r(=y)x\\
\Box r(p \mathbin{\dot\wedge} q) &= \Box r p \mathbin{\dot\wedge} \Box r q\\
\Diamond r(p \mathbin{\dot\vee} q) &= \Diamond r p \mathbin{\dot\vee} \Diamond r q\\
\Diamond r(p \mathbin{\dot\to} q) &= \Box r p \mathbin{\dot\to} \Diamond r q\\
\Diamond r\dot\bot &= \dot\bot\\
\Box r\dot\top &= \dot\top
\end{aligned}
$$

Figure 15.2: Some modally valid equations

as additional rule.

Example 15.2.2 Here is a tableau that proves that the formula $\dot\neg\dot\neg p = p$ is modally valid.

$$
\begin{array}{c}
(\dot\neg) = \lambda p x.\neg p x\\
\dot\neg\dot\neg p \not\equiv p\\
(\dot\neg\dot\neg p)x \not\equiv p x\\
\neg(\dot\neg p)x \not\equiv p x\\
\neg\neg p x \not\equiv p x\\
\hline
\begin{array}{c|c}
\neg\neg p x & \neg\neg\neg p x\\
\neg p x & p x\\
 & \neg p x
\end{array}
\end{array}
$$

□

Exercise 15.2.3 Prove with tableaux that some of the equations in Figure 15.2 are modally valid. You may eliminate all modal constants before you start.

Exercise 15.2.4 Decide for each of the following formulas whether it is modally valid.

a) $\forall\,(\Diamond r(p \mathbin{\dot\wedge} q) \mathrel{\dot\rightarrow} \Diamond r p)$

b) $\forall\,(\Box r(p \mathbin{\dot\wedge} q) \mathrel{\dot\rightarrow} \Box r p)$

c) $\Diamond r \bot = \top$

d) $\Diamond r \top = \Box r p \mathrel{\dot\rightarrow} \Diamond r p$

Exercise 15.2.5 (First-Order Translation) For each of the following modal formulas s find a first-order formula t not containing modal constants such that $s \equiv t$ is true in every modal interpretation. Hint: Eliminate the modal constants using their defining equations and apply β-reduction.

a) $(p \mathbin{\dot\vee} q)x$

b) $\forall(p \mathrel{\dot\rightarrow} q)$

c) $\exists(\Box r(p \mathrel{\dot\rightarrow} \Diamond r p))$

d) $\forall(\Diamond r(\Box r(\Diamond r \dot\neg p)))$

15.3 Modal Expressions and Modal Formulas

A **modal variable** is a variable whose type is either ι or ιo or $\iota\iota o$. If not said otherwise, the following letters will range over modal variables with the specified type:

$x, y, z : \iota$	individual variables
$p, q : \iota o$	property variables
$r : \iota\iota o$	relation variables

A **modal expression** is a term of type ιo that can be obtained with the grammar

$$s ::= p \mid \bot \mid \top \mid \dot\neg s \mid s \mathbin{\dot\wedge} s \mid s \mathbin{\dot\vee} s \mid s \mathrel{\dot\rightarrow} s \mid s \doteq s \mid \Box r s \mid \Diamond r s$$

where p and r range over variables of type ιo and $\iota\iota o$. A **modal formula** is a formula that can be obtained with the grammar

$$t ::= sx \mid rxx \mid \forall s \mid \exists s \mid \neg t \mid t \wedge t \mid t \vee t \mid t \rightarrow t \mid t \equiv t$$

where s ranges over modal expressions and x ranges over variables of type ι. A **modal term** is a modal expression or a modal formula.

To evaluate a modal term s with a modal interpretation $\mathcal{I}$, it suffices to know how $\mathcal{I}$ interprets the sort ι and the modal variables that occur in s. We will now see that there is a relationship between transition systems and modal interpretations. Eventually, it will turn out that a modal formula is modally satisfiable if and only if it is satisfied by some finite transition system.

An interpretation $\mathcal{I}$ **agrees with a transition system** T if it satisfies the following conditions:

1. $\mathcal{I}\iota = \mathcal{V}_\iota T$
2. $\mathcal{I}x = x$ for every $x \in \mathcal{V}_\iota T$
3. $\mathcal{I}px = 1 \iff px \in T$ for every variable $p : \iota o$
4. $\mathcal{I}rxy = 1 \iff rxy \in T$ for every variable $r : \iota\iota o$

Note that a transition system fully determines the interpretation of the sort ι and the interpretation of all modal variables of type ιo and $\iota\iota o$. Moreover, the interpretation of all individual variables that occur in T is determined.

Proposition 15.3.1 For every transition system there is a modal interpretation that agrees with it. Moreover, if a logical interpretation agrees with a transition system, it satisfies every formula of the system.

Proposition 15.3.2 (Modal Coincidence) Let $\mathcal{I}$ and $\mathcal{J}$ be modal interpretations that agree with a transition system T. Then $\hat{\mathcal{I}}$ and $\hat{\mathcal{J}}$ agree on all modal terms s such that $\mathcal{V}_\iota s \subseteq \mathcal{V}_\iota T$.

Proof By induction on s. ∎

A transition system T **satisfies** a modal formula s if $\mathcal{V}_\iota s \subseteq \mathcal{V}_\iota T$ and there is a modal interpretation $\mathcal{I}$ that agrees with T and satisfies s. By the preceding propositions we know that there always is an agreeing interpretation and that it does not matter which of the agreeing interpretations we take.

Proposition 15.3.3 (Model Checking) It is decidable whether a finite transition system satisfies a modal formula.

Exercise 15.3.4 Give a recursive procedure *check* that for a finite transition system and a modal formula decides whether the system satisfies the formula. Hint: You need equations for the different forms of modal formulas. The equations for modal formulas of the form $\forall s$ and $s \wedge t$ are as follows.

$$\begin{aligned} \mathit{check}\ T\ (\forall s) &= \text{if } \forall x \in \mathcal{V}_\iota T\text{: } \mathit{check}\ T\ (sx) = 1 \text{ then 1 else 0} \\ \mathit{check}\ T\ (s \wedge t) &= \text{if } \mathit{check}\ T\ s = 1 \text{ then } \mathit{check}\ T\ t \text{ else 0} \end{aligned}$$

Exercise 15.3.5 For each of the following modal formulas draw a finite transition system that satisfies the formula.

a) $\forall(\Diamond r \dot{\neg} p \mathbin{\dot{\wedge}} \Box r(\Diamond r(\Diamond r p)))$

b) $px \wedge \forall(p \doteq \dot{\neg} q) \mathbin{\dot{\wedge}} \forall(\Diamond r q)$

c) $\forall((p \mathbin{\dot{\rightarrow}} \Diamond r q) \mathbin{\dot{\wedge}} (q \mathbin{\dot{\rightarrow}} \Diamond r p) \mathbin{\dot{\wedge}} (p \mathbin{\dot{\vee}} q))$

d) $px \wedge qx \wedge \forall(\Diamond r \dagger) \wedge \forall(\Box r(p \doteq \dot{\neg} q))$

Exercise 15.3.6 For each of the following modal formulas draw a transition system that does not satisfy the formula.

a) $\forall(\Diamond r p \mathbin{\dot{\rightarrow}} \Diamond r q \mathbin{\dot{\rightarrow}} \Diamond r(p \mathbin{\dot{\wedge}} q))$

b) $\exists(\Box r(p \mathbin{\dot{\vee}} q) \mathbin{\dot{\rightarrow}} \Box r p \mathbin{\dot{\vee}} \Box r q)$

Exercise 15.3.7 Let s and t be modal expressions and $x : \iota$ be a variable. Give a modal expression u such that ux is not modally satisfiable if and only if $\forall(s \mathbin{\dot{\rightarrow}} t)$ is modally valid.

15.4 Main Results

There are two important results about modal formulas:

1. A modal formula is modally satisfiable if and only if it is satisfied by a finite transition system.
2. There is an algorithm that decides for every modal formula s whether there is a finite transition system that satisfies s. Whenever such a transition system exists, the algorithm constructs one.

The first result says that the abstract semantics provided by modal interpretations coincides with the concrete semantics provided by finite transition systems. It also says that modal formulas have the finite model property. The second result tells us that many properties of modal formulas are decidable.

We will prove the above results only for modal expressions and modal formulas that can be obtained with the grammar

$$s ::= p \mid \dot{\neg} p \mid s \mathbin{\dot{\wedge}} s \mid s \mathbin{\dot{\vee}} s \mid \Box r s \mid \Diamond r s$$

$$t ::= sx \mid rxx \mid \forall s \mid \exists s$$

We refer to such expressions and formulas as **simple modal expressions** and **simple modal formulas**. Our proof can be extended to all modal expressions and all modal formulas. The treatment of modal formulas containing negative occurrences of formulas rxy requires some thought.

Exercise 15.4.1 Give a set of equations so that every modal expression can be rewritten into a simple modal expression. Every equation must be modally valid.

$$\frac{px\,,\ (\dot\neg p)x}{} \qquad \frac{(s\,\dot\wedge\, t)x}{sx\,,\ tx} \qquad \frac{(s\,\dot\vee\, t)x}{sx\mid tx}$$

$$\frac{\Box rsx\,,\ rxy}{sy} \qquad \frac{\Diamond rsx}{rxy\,,\ sy}\quad \begin{array}{l} y \text{ fresh} \\ \Diamond rsx \text{ not realized in } A \end{array}$$

$$\frac{\forall s}{sx}\ x \in \mathcal{V}_\iota A \qquad \frac{\exists s}{sx}\quad \begin{array}{l} x \text{ fresh} \\ \exists s \text{ not realized in } A \end{array}$$

Figure 15.3: Rules of the tableau system $\mathcal{T}$

15.5 Terminating Tableau System

The decision algorithm for simple modal formulas comes as a terminating tableau system, which constructs for every modally satisfiable formula a finite transition system satisfying it. We start with some definitions.

A **branch** is a set A of simple modal formulas such that A contains either a formula of the form $\exists s$ or a formula that contains a variable of type ι. A modal interpretation **satisfies a branch** A if it satisfies every formula $s \in A$. A formula $\exists s$ **is realized in a branch** A if $sx \in A$ for some variable x. A formula $\Diamond rsx$ **is realized in a branch** A if there exists a transition $rx'y \in A$ such that $sy \in A$ and $\Box rtx' \in A$ whenever $\Box rtx \in A$. More formally, $\Diamond rsx$ is realized in A if and only if

$$\exists x' \exists y\colon\ rx'y \in A \wedge sy \in A \wedge \forall t\colon\ \Box rtx \in A \Longrightarrow \Box rtx' \in A$$

The **tableau system** $\mathcal{T}$ operates on branches as defined and employs the rules shown in Figure 15.3.

Example 15.5.1 We want to show that the formula

$$s\ :=\ \forall(\Diamond r(p\,\dot\vee\, q) \dot\rightarrow \Diamond rp\,\dot\vee\,\Diamond rq)$$

is modally valid. We do this by showing that $\neg s$ is not modally satisfiable. Using the equations in Figure 15.2 we can rewrite $\neg s$ to the modally equivalent formula $\exists(\Diamond r(p\,\dot\vee\, q)\,\dot\wedge\,\Box r\dot\neg p\,\dot\wedge\,\Box r\dot\neg q)$. A refutation of this formula with $\mathcal{T}$ looks

as follows:

$$\begin{array}{c}
\exists(\Diamond r(p \dot\vee q) \dot\wedge \Box r \dot\neg p \dot\wedge \Box r \dot\neg q) \\
(\Diamond r(p \dot\vee q) \dot\wedge \Box r \dot\neg p \dot\wedge \Box r \dot\neg q)x \\
\Diamond r(p \dot\vee q)x,\ \Box r(\dot\neg p)x,\ \Box r(\dot\neg q)x \\
rxy,\ (p \dot\vee q)x, \\
(\dot\neg p)y,\ (\dot\neg q)y \\
\hline
py \mid qy
\end{array}$$

□

Exercise 15.5.2 Refute the following modal formulas with the tableau system $\mathcal{T}$.

a) $\exists(\Box r(p \dot\wedge q) \dot\wedge \Diamond r \dot\neg p)$

b) $\exists(\Diamond(p \dot\vee q) \dot\wedge \Box r(\dot\neg p \dot\wedge \dot\neg q))$

15.5.1 Refutation Soundness

For refutation soundness we have to show for every rule $A/A_1 \ldots A_n$ of $\mathcal{T}$ and every modal interpretation that satisfies A there exists an $i \in [1,n]$ such that there exists a modal interpretation that satisfies A_i. This is easy to verify.

Proposition 15.5.3 $\mathcal{T}$ is refutation sound.

15.5.2 Termination

Let A be a branch. We use $\mathcal{M}A$ to denote the set of all modal expressions that occur as a subterm in a formula of A. The key observation for the termination proof is that a tableau expansion does not create new modal expressions. That is, every rule $A/A_1 \ldots A_n$ of $\mathcal{T}$ satisfies $\mathcal{M}A = \mathcal{M}A_1 = \cdots = \mathcal{M}A_n$.

We now assume that the intial branch A_0 is finite and that we only talk about branches that can be obtained from A_0 by finitely many applications of the rules of $\mathcal{T}$.

A modal expression s is **realized in a branch** A if $sx \in A$ for some variable x. We observe that every application of the $\exists$-rule realizes a modal expression $s \in \mathcal{M}A_0$ that was not realized before. Since extension of a branch with new formulas preserves realization of modal expressions and $\mathcal{M}A_0$ is finite, we can ignore the $\exists$-rule when we prove termination.

A **pattern** is a set $\{\Diamond rs, \Box rt_1, \ldots \Box rt_n\}$ of modal expressions. A pattern $\{\Diamond rs, \Box rt_1, \ldots \Box rt_n\}$ is **realized in a branch** A if there exist x and y such that $\{rxy, sy, \Box rt_1 x, \ldots \Box rt_n x\} \subseteq A$. We observe that every application of the $\Diamond$-rule realizes a pattern $P \subseteq \mathcal{M}A_0$ that was not realized before. Since extension of a branch with new formulas preserves realization of patterns and $\mathcal{M}A_0$ is finite, we can ignore the $\Diamond$-rule when we prove termination.

The remaining rules do not introduce new names. We consider the closure

$$CA \;:=\; A \cup \{\, sx \mid s \in \mathcal{M}A,\ x \in \mathcal{V}_\iota A \,\}$$

that is finite if the branch A is finite. Every remaing rule $A/A_1 \dots A_n$ of $\mathcal{T}$ satisfies $CA = CA_1 = \cdots = CA_n$. Since all reachable branches are subsets of CA_0 and CA_0 is finite, the remaining rules terminate (we assume that the $\exists$-rule and the $\Diamond$-rule are not applied).

Proposition 15.5.4 $\mathcal{T}$ terminates on finite branches.

15.5.3 Verification Soundness

For every branch A we define the **associated transition system** T_A as follows:

$$T_A \;:=\; \{\, px \mid px \in A \,\} \cup \{\, rxy \mid \{r,x,y\} \subseteq \mathcal{V}A \wedge \forall t\colon\ \Box rtx \in A \Longrightarrow ty \in A \,\}$$

Note that T_A is finite if A is finite. The formulas $rxy \in T_A$ are called **safe transitions**.

Example 15.5.5 Let $A = \{\Diamond rpx, \Diamond r(\dot\neg p)x, \Box rqx, py, qy, (\dot\neg p)z, qz\}$. Then $\Diamond rpx$ and $\Diamond r(\dot\neg p)x$ are realized in A. Moreover, $T_A = \{py, qy, qz, rxy, rxz\}$.□

Proposition 15.5.6 Let A be a branch that is maximal and open for $\mathcal{T}$. Then A is satisfied by the transition system T_A.

Proof First observe that T_A contains for every formula $\Diamond rsx$ a transition rxy such that $sy \in A$ and $ty \in A$ whenever $\Box rtx \in A$. Let $\mathcal{I}$ be a modal interpretation that agrees with T_A. We show by induction on s that $\mathcal{I}$ satisfies every formula $s \in A$. Let $s \in A$. Case analysis.

Let $s = px$ or $s = rxy$. Then $s \in T_A$ and is thus satisfied by $\mathcal{I}$.

Let $s = \forall t$. Then $tx \in A$ and $\mathcal{I}x = x$ for all $x \in \mathcal{I}\iota$. By induction hypothesis we know that $\mathcal{I}$ satisfies tx for every $x \in \mathcal{I}\iota$. Thus $\mathcal{I}$ satisfies $\forall t$.

To be completed. ∎

Exercise 15.5.7 Use the tableau system $\mathcal{T}$ to construct finite transition systems satisfying the following modal formulas.

a) $\exists(\Diamond rp \mathbin{\dot\wedge} \Box r(\Diamond r \dot\neg p))$

b) $\dot\neg px \wedge \forall(\Diamond r(\Diamond rp))$

Exercise 15.5.8 (Challenge) Give the additional tableau rules needed so that all modal expressions can be handled. Make sure that the resulting tableau system is refutation sound, terminating, and verification sound.

15.6 Remarks

Modal logic is an active research area and there are many modal logics. The textbook [11] is an excellent introduction to modal logic in general and gives a good overview of the field. The handbook of description logics [5] focusses on applications of modal logic in knowledge representation. The textbook [33] covers dynamic logic, a branch of modal logic developed for programm verification. For temporal logic you may consult the textbook [6].

The modal logic considered in this chapter is the essentially the description logic ALC. Deciding satisfiability of ALC formulas is EXPTIME-complete [5]. Our terminating tableau system is derived from a terminating tableau system devised by Kaminski and Smolka [46].

Most 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.

We conclude with some historical remarks.

- 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, Vaughan 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 [5]. Description logic is the basis for the web ontology language OWL (see Wikipedia). There is a connection between dynamic logic and description logic.

Bibliography

[1] P. B. Andrews. General models and extensionality. *J. Symb. Log.*, 37:395–397, 1972.

[2] P. B. Andrews. General models, descriptions and choice in type theory. *J. Symb. Log.*, 37:385–394, 1972.

[3] 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.

[4] Peter B. Andrews and Chad E. Brown. TPS: A hybrid automatic-interactive system for developing proofs. *Journal of Applied Logic*, 4(4):367–395, 2006.

[5] 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.

[6] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[7] 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.

[8] Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.

[9] Paul Bernays and Moses Schönfinkel. Zum Entscheidungsproblem der Mathematischen Logik. *Math. Annalen*, 99:342–372, 1928.

[10] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[11] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge University Press, 2001.

[12] George Boole. *An Investigation of the Laws of Thought*. Walton, London, 1847.

[13] Chad E. Brown. *Automated Reasoning in Higher-Order Logic: Set Comprehension and Extensionality in Church's Type Theory*. College Publications, 2007.

[14] Chad E. Brown and Gert Smolka. Complete cut-free tableaux for equational simple type theory. Technical report, Programming Systems Lab, Saarland University, April 2009.

[15] Chad E. Brown and Gert Smolka. Extended first-order logic. In Tobias Nipkow and Christian Urban, editors, *TPHOLs 2009*, volume 5674 of *LNCS*. Springer, August 2009.

[16] Chad E. Brown and Gert Smolka. Terminating tableaux for the basic fragment of simple type theory. In M. Giese and A. Waaler, editors, *TABLEAUX 2009*, volume 5607 of *LNCS (LNAI)*, pages 138–151. Springer, 2009.

[17] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Springer, 1997.

[18] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 32:346–366, 1932.

[19] Alonzo Church. A note on the entscheidungsproblem. *J. Symb. Log.*, 1(1):40–41, 1936.

[20] Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(1):56–68, 1940.

[21] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM Press.

[22] N .G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 579–606. Academic Press, 1980.

[23] 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.

[24] Ph. de Groote, editor. *The Curry-Howard Isomorphism*. Academia, Louvain-la-Neuve (Belgium), 1995.

[25] Melvin Fitting. *First-order logic and automated theorem proving*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[26] Gottlob Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Verlag von Luois Nebert, Halle, 1879. Translated in [67], pp. 1–82.

[27] Gottlob Frege. *Grundgesetze der Arithmetik begriffsschriftlich abgeleitet*. Verlag Hermann Pohle, Jena, 1893. Translated in [67].

[28] 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.

[29] R. O. Gandy. An early proof of normalization by A. M. Turing. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 453–455. Academic Press, 1980.

[30] Gerhard Gentzen. Untersuchungen über das natürliche Schließen I, II. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.

[31] Kurt Gödel. Die Vollständigkeit der Axiome des logischen Functionenkalküls. *Monatshefte für Mathematik und Physik*, 37:349–360, 1930. Translated in [67], pp. 102-123.

[32] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik*, 38(1):173–198, 1931. Translated in [67].

[33] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic logic*. MIT Press, 2000.

[34] John Harrison. HOL Light tutorial (for version 2.20). http://www.cl.cam.ac.uk/~jrh13/hol-light/tutorial_220.pdf, 2006.

[35] L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15(2):81–91, June 1950.

[36] L. Henkin. A theory of propositional types. *Fundamenta Mathematicae*, 52:323–344, 1963.

[37] Leon Henkin. The completeness of the first-order functional calculus. *Journal of Symbolic Logic*, 14:159–166, 1949.

[38] Jacques Herbrand. Recherches sur la théorie de la démonstration. *Travaux de la Société des Sciences et des Lettres de Varsovie, Classe III Sciences Mathematiques et Physiques*, 33, 1930. Translated in [39].

[39] Jacques Herbrand. *Logical Writings*. Harvard University Press, 1971. Edited by Warren D. Goldfarb.

[40] David Hilbert and Wilhelm Ackermann. *Grundzüge der theoretischen Logik*. Springer, 1928.

[41] J. R. Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1997.

[42] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators, an Introduction*. Cambridge University Press, 2008.

[43] K. Jaakko J. Hintikka. Form and content in quantification theory. Two papers on symbolic logic. *Acta Philosophica Fennica*, 8:7–55, 1955.

[44] W. A. Howard. The formula-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 480–490. Academic Press, 1980.

[45] Michael Huth and Mark Ryan. *Logic in Computer Science*. Cambridge, second edition, 2004.

[46] Mark Kaminski and Gert Smolka. Terminating tableau systems for hybrid logic with difference and converse. *Journal of Logic, Language and Information*, 2009.

[47] László Kálmar. Zurückführung des Entscheidungsproblems auf den Fall von Formeln mit einer einzigen, binären Funktionsvariablen. *Compositio Mathematica*, 74:137–144, 1936.

[48] E. Landau. *Grundlagen der Analysis*. Leizig, 1930.

[49] Leopold Löwenheim. Über Möglichkeiten im Relativkalkül. *Math. Annalen*, 76:447–470, 1915. Translated in [67].

[50] J. C. Mitchell. *Foundations for Programming Languages*. Foundations of Computing. The MIT Press, 1996.

[51] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[52] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[53] G. Peano. Arithmetices principia, nova methodo exposita. Turin, 1889. Translated in [67], pp. 83–97.

[54] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[55] Frank Ramsey. On a problem of formal logic. *Proc. of the London Math. Society, 2nd series*, 30:264–286, 1930.

[56] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[57] Bertrand Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30:222–262, 1908.

[58] Moses Schönfinkel. Über die Bausteine der Mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924.

[59] Thoralf Skolem. Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit oder Beweisbarkeit mathematischer Sätze nebst einem Theorem über dichte Mengen. *Nork Vid-Akad. Oslo Mat.-Natur Kl. Skr.*, 4, 1920. Translated in [67], pp. 252–263.

[60] Thoralf Skolem. Über die mathematische Logik. *Norse matematisk tidsskrift*, 10:125–142, 1928. Translated in [67], pp. 508–524.

[61] Raymond Smullyan. *First-order logic*. Dover, 1995.

[62] Raymond M. Smullyan. A unifying principle in quantification theory. *Proceedings of the National Academy of Sciences (U.S.A.)*, 49:828–832, 1963.

[63] Richard Statman. The typed lambda-calculus is not elementary recursive. In *Proc. of FOCS*, pages 90–94. IEEE, 1977.

[64] Allen Stoughton. Substitution revisited. *Theoretical Computer Science*, 59:317–325, 1988. http://people.cis.ksu.edu/~stough/research/subst.ps.

[65] W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

[66] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. of the London Math. Society, 2nd series*, 42:230–265, 1937.

[67] 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.

[68] John Eldon Whitesitt. *Boolean Algebra and Its Applications*. Dover Publications, 1995.

[69] Freek Wiedijk. A new implementation of Automath. *J. Autom. Reasoning*, 29(3-4):365–387, 2002.