Higher-Order Logic

Lecture Notes

July 19, 2007

Gert Smolka
Department of Computer Science
Saarland University

Copyright © 2007 by Gert Smolka, All Rights Reserved

Contents

1 The Language of Higher-Order Logic

In this section, we review basic logical notions such as functions, Boolean connectives, quantification and equality. We do this so that we can sketch the language of higher-order logic, a simple yet amazingly expressive language for mathematical statements.

We assume that you have some experience with mathematical language and argumentation. In particular, you should be familiar with numbers and sets. If you know a functional programming language (e.g., ML or Haskell) you already have seen some of the key ideas of higher-order logic.

1.1 Functions

Functions will play a major role in our language. Everyone knows that a function is something that takes an argument and yields a result. Nowadays, functions are defined as sets of pairs. We will use the following definition.

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

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

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

The canonical means for describing functions is the **lambda notation** developed by the American logician Alonzo Church around 1930. Here is an example:

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

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

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

It shows the analogy between lambda notation and the usual notation for sets.

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

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

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:

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

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. They first appeared 1924 in a paper by Moses Schönfinkel on the primitives of mathematical language. In traditional mathematics, one typically uses **cartesian representations** of binary operations. The type of the cartesian representation of integer addition would be $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. We prefer to work with the cascaded representation because it represents operations with multiple arguments just with functions. In contrast, the cartesian representation requires pairs in addition to functions.

For convenience, 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 $\mathit{plus}\ 3\ 7 = 10$ and $\mathit{plus} \in \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$.

We mention two laws that hold for the lambda notation:

$$\begin{aligned} &\lambda x{\in}X.\ f\,x = f && \eta\text{-law} \\ &(\lambda x{\in}X.\ e)\,e' = [x := e']\,e && \beta\text{-law} \end{aligned}$$

In the β-law, e and e' are expressions and $[x := e']e$ stands for the expression that is obtained from e by replacing the occurrences of the variable x with the expression e'. Here is an **instance of the β-law**:

$$(\lambda x{\in}\mathbb{N}.\ x^2)(3 + y) = (3 + y)^2$$

1.2 Boolean Connectives

Mathematical statements often employ Boolean connectives such as conjunction and implication. We use the numbers 0 and 1 as **Boolean values,** where 0 may be thought of as "false" and 1 as "true". We define

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

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

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

The Boolean connectives are defined as follows:

$$\neg x = 1 - x \qquad \textbf{Negation}$$

$$x \wedge y = \min\{x, y\} \qquad \textbf{Conjunction}$$
$$x \vee y = \max\{x, y\} \qquad \textbf{Disjunction}$$
$$x \Rightarrow y = \neg x \vee y \qquad \textbf{Implication}$$
$$x \Leftrightarrow y = (x = y) \qquad \textbf{Equivalence}$$

We can represent negation as a function $\mathbb{B} \to \mathbb{B}$ and the binary Boolean connectives as functions $\mathbb{B} \to \mathbb{B} \to \mathbb{B}$. We use the symbols $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$ to name these functions, and we refer to them as **Boolean operations**. There are many interesting laws for the Boolean operations:

$$\neg(x \wedge y) = \neg x \vee \neg y$$
$$x \wedge (x \vee y) = x$$
$$x \vee y = (x \Rightarrow y) \Rightarrow y$$
$$\neg(\neg x) = x$$

We can prove these laws by exhaustive case analysis, i.e., by verifying them for all possible values of x and y. This yields 4 cases:

1. $x = 0$ and $y = 0$
2. $x = 0$ and $y = 1$
3. $x = 1$ and $y = 0$
4. $x = 1$ and $y = 1$

There are smarter ways to decide the validity of Boolean equations. We will say more about this later.

1.3 Quantification

Mathematical statements often involve quantification, for instance

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

Church realized that the quantifiers $\forall$ and $\exists$ can be represented as functions and that quantification can be represented as the application of a quantifier to a function described with the lambda notation. We can say that Church did for the quantifiers what Boole did for the Boolean connectives, i.e., reduce them to simple functions.

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

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

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

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

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

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

The usual notation for quantification can in fact be obtained as notation:

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

There are many interesting laws for quantifiers:

$$\begin{aligned} \neg(\forall x{\in}X\colon e) &= (\exists x{\in}X\colon \neg e) \\ (\forall x{\in}X\colon e \wedge e') &= (\forall x{\in}X\colon e) \wedge (\forall x{\in}X\colon e') \\ (\forall x{\in}X\colon y) &= y \end{aligned}$$

1.4 Identities

You cannot do mathematics without using equality. For each set X, equality for the members of X can be represented by the following function:

$$\begin{aligned} &\doteq_X \in X \to X \to \mathbb{B} \\ &x \doteq_X y = (x = y) \end{aligned}$$

We refer to $\doteq_X$ as the **identity predicate for** X, or shorter as the **identity for** X. With the identity $\doteq_{\mathbb{Z}}$ we are able to represent the equation $x = y$ as an expression $x \doteq_{\mathbb{Z}} y$ that applies the function $\doteq_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{Z} \to \mathbb{B}$ to x and y.

There are lots of interesting laws for identities. Everyone knows the following:

$$\begin{aligned} (x \doteq_X x) &= 1 && \textbf{Reflexivity} \\ (x \doteq_X y) &= (y \doteq_X x) && \textbf{Symmetry} \\ (x \doteq_X y \wedge y \doteq_X z \Rightarrow x \doteq_X z) &= 1 && \textbf{Transitivity} \end{aligned}$$

A very interesting law is the Leibniz law:

$$(x \doteq_X y) = (\forall f{\in}X \to \mathbb{B}\colon\ fx \Rightarrow fy) \qquad \textbf{Leibniz}$$

It says that the identities can be expressed with the universal quantifiers and implication. It can be paraphrased as saying that two objects x and y are equal if y satisfies every property x satisfies. Let's look at a proof of Leibniz' law.

Proof Let $x, y \in X$. Case Analysis.

Let $x = y$. Then

$$\begin{aligned}(\forall f{\in}X \to \mathbb{B}\colon\ fx \Rightarrow fy) &= (\forall f{\in}X \to \mathbb{B}\colon\ fx \Rightarrow fx)\\ &= (\forall f{\in}X \to \mathbb{B}\colon\ 1)\\ &= 1\\ &= (x \doteq_X y)\end{aligned}$$

Let $x \neq y$. Then there is a function $g \in X \to \mathbb{B}$ such that $gx = 1$ and $gy = 0$. Hence $(gx \Rightarrow gy) = 0$. Hence $(\forall f{\in}X \to \mathbb{B}\colon\ fx \Rightarrow fy) = 0 = (x \doteq_X y)$. ∎

Here are some more laws for the identities:

$$\begin{aligned}\neg x &= (x \doteq_{\mathbb{B}} 0)\\ x \wedge y &= (\forall f{\in}\mathbb{B} \to \mathbb{B} \to \mathbb{B}\colon\ fxy \doteq_{\mathbb{B}} f11)\\ (f \doteq_{X\to Y} g) &= (\forall x{\in}X\colon\ fx \doteq_Y gx)\end{aligned}$$

1.5 Sets as Functions

Let X be a set. The subsets of X can be expressed as functions $X \to \mathbb{B}$. We will represent a subset A with the function $\lambda x{\in}X.\ x \in A$ that yields 1 for $x \in X$ iff x is an element of A. This function is called the **characteristic function of A with respect to X**. Now we can obtain the usual set notations for subsets $A, B \subseteq X$ as follows:

$$\begin{aligned}x \in A &:= Ax\\ A \cap B &:= \lambda x{\in}X.\ Ax \wedge Bx\\ A \cup B &:= \lambda x{\in}X.\ Ax \vee Bx\end{aligned}$$

1.6 Intensional and Extensional Interpretation

Given the notation $2 + 3$, we can interpret it in two ways. The **intensional interpretation** sees it as an expression applying addition to 2 and 3. The **extensional interpretation** sees it as the number 5. When we say that the equation $2 + 3 = 5$ is valid, we use the extensional interpretation of $2 + 3$. When we say that the addition of 2 and 3 yields 5, we use the intensional interpretation of $2 + 3$. Due to our mathematical training, we automatically choose the interpretation required by a given context.

The intensional interpretation of a notation is called the **intension of the notation**, and the extensional interpretation of a notation is called the **extension of**

the notation. Things related to the intensions of notations are called **syntactic**, and things related to the extensions of notations are called **semantic**.

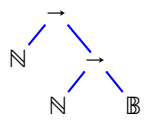
Given a language, we distinguish between the **concrete syntax** (i.e., the notation), the **abstract syntax** (i.e., the intensions), and the **semantics** (i.e., the extensions). At the heart of every language is the abstract syntax.

1.7 Types and Terms

We now sketch the abstract syntax of the language of higher-order logic. It distinguishes between two kinds of objects, called types and terms. Types are the intensions of notations like $\mathbb{B}$ and $\mathbb{N} \to \mathbb{N} \to \mathbb{B}$. Terms are the intensions of notations like $3 + 4$, $2 < 5$, and $\forall x{\in}\mathbb{Z}\colon x + 0 = x$.

We assume that some primitive types, called **sorts**, and some primitive terms, called **names**, are given. Typical examples of sorts are $\mathbb{Z}$ and $\mathbb{B}$. Typical examples of names are 1, $+$, $\Rightarrow$, $\forall_T$ and $\doteq_T$.

A type is either a sort or a functional type. A **functional type** is a pair (S, T) of two types. The usual notation for a functional type (S, T) is $S \to T$. Often it is helpful to think of types as trees. For instance, the type $\mathbb{N} \to \mathbb{N} \to \mathbb{B}$ can be represented by the following tree:



We speak of the **tree representation of a type**.

Every name must come with a type. For instance, the type of 1 is $\mathbb{B}$ and the type of $\forall_T$ is $(T \to \mathbb{B}) \to \mathbb{B}$.

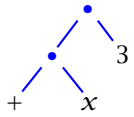
A term is either a name, an application or an abstraction. Every term has a unique type.

An **application** is a pair (s, t) of two terms where s must have a functional type $S \to T$ and t must have the type S. The usual notation for an application (s, t) is st. We also use **infix notation** for applications. For instance, the notation $x + y$ describes the double application $((+x)y)$.

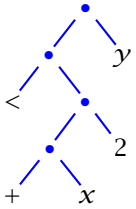
Every **abstraction** can be described with the notation $\lambda u{\in}T.t$, where u is some name, T is a type, and t is a term.

Terms that can be described without the use of lambda notation are called **combinatorial**. For instance, $x + 2{\cdot}y$ and $\forall_{\mathbb{B}}(\neg)$ are combinatorial terms, but $\forall_{\mathbb{N}}(\lambda x{\in}\mathbb{N}.\, x < 5)$ is not a combinatorial term.

Like types, terms have **tree representations**. The tree representation of the term $x + 3$ is



It shows explicitly how the term $x + 3$ is obtained with two applications from the names $+$, x and 3. As further example we show the tree representation of the term $x + 2 < y$:

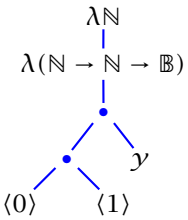


Before we can show the tree representation of abstractions, we need to clarify an important issue. It is clear that the notations $\lambda n{\in}\mathbb{N}.n$ and $\lambda x{\in}\mathbb{N}.x$ describe the same function, that is

$$(\lambda n{\in}\mathbb{N}.n) = (\lambda x{\in}\mathbb{N}.x) = \{\,(x,x) \mid x \in \mathbb{N}\,\}$$

In other words, it does not matter for the extensional interpretation what name we use for the argument variable of a lambda notation. For reasons that will become clear later, this should also be the case for the intensional interpretation of the lambda notation. That is, $\lambda n{\in}\mathbb{N}.n$ and $\lambda x{\in}\mathbb{N}.x$ should describe the same term. To account for this fact, the tree representation of an abstraction does not represent argument references by names. Rather, argument references are represented as natural numbers that identify the lambda node they refer to. For instance, the tree representation of the term $\lambda n{\in}\mathbb{N}.n$ is

and the tree representation of the term $\lambda x{\in}\mathbb{N}.\ \lambda f{\in}\mathbb{N} \to \mathbb{N} \to \mathbb{B}.\ fxy$ is



An argument reference $\langle n\rangle$ represents the argument of the abstraction whose lambda node appears on the path to the root encountered after skipping n lambda nodes. This numeric form of argument reference was invented by the

Dutch logician Nicolaas de Bruijn (around 1965). Hence a numeric argument reference $\langle n \rangle$ is often called a **de Bruijn index**.

The language given by types and terms is a a very simple functional language discovered by Alonzo Church between 1930 and 1940. It is known as the **language of higher-order logic**, and its terms are referred to as **simply typed lambda terms**. Simply typed lambda terms can express an amazingly rich class of mathematical statements. All the example statements we have discussed so far can be expressed as terms, given the right sorts and names. We will see later that the language of higher-order logic is more expressive than the languages that come with propositional logic, predicate logic and modal logic.

1.8 Notational Conventions

We will give a precise definition of the abstract syntax of the language of higher-order logic in the next section. As it comes to the notation for this language, we will rely on your experience with mathematical notation in general, and a few special conventions.

The main notational conventions for types and terms are summarized by the following grammar:

C	sorts
$S, T ::= C \mid S \to T$	types
u, x	names
$s, t ::= u \mid st \mid \lambda x{\in}T.t$	terms

There are conventions that make it possible to omit parentheses. For example:

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

The symbols $+$ and $\cdot$ are said to have **infix status**, and the symbol $\cdot$ is said to take its arguments before the symbol $+$. We assume the following **precedence hierarchy** for some of the function symbols:

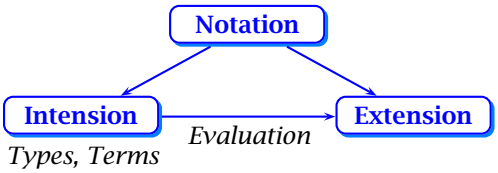
$$\begin{array}{c} = \\ \Leftrightarrow \\ \Rightarrow \\ \vee \\ \wedge \\ \neg \\ \doteq \quad < \quad \leq \quad > \quad \geq \\ + \quad - \\ \cdot \end{array}$$

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

$$x \vee x \wedge y = x \quad \rightsquigarrow \quad (x \vee (x \wedge y)) = x$$
$$\neg\neg x \doteq y = y \doteq x \quad \rightsquigarrow \quad \neg(\neg(x \doteq y)) = (y \doteq x)$$

2 Terms and Types

So far, we know that we can associate a mathematical notation with an intension and an extension. The intension represents the syntactic structure of the notation, and the extension is the value decribed by the notation. The language of higher order logic has two kinds of intensions, called types and terms. The intension of a notation contains more information than the extension of the notation. Hence we can obtain the extension from the intension. One says that the extension is obtained by evaluation of the intension. We can represent the situation with the following diagram:



In this section, we give a mathematical account of types and terms. To simplify things, we ignore their extensions and treat types and terms as objects of a mathematical data structure.

2.1 Untyped Terms

We first consider so called untyped terms that do not involve types. Many interesting aspects of terms can be studied with this simplified model, and types can be added easily once they are needed.

An **untyped term** is either a name, an application, or an abstraction. Since we ignore types and extensions, we can take any two terms s, t and construct the application st. Moreover, we can take a name x and a term t and construct the abstraction $\lambda x.t$. We summarize our notational conventions for untyped terms as follows:

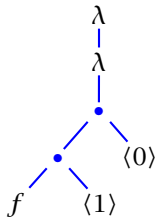
$$x, y, z, f, g, h \qquad \text{names}$$

$$s, t ::= x \mid st \mid \lambda x.t \qquad \text{terms}$$

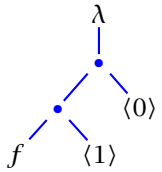
$$\lambda xy.t := \lambda x.\lambda y.t$$

$$stt' := (st)t'$$

The **tree representation** of the term $\lambda xy.fxy$ looks as follows:



Note that the tree representations of terms are **not subtree-closed**. For instance, the tree



is not a tree representation of a term since it contains the **dangling argument reference** $\langle 1\rangle$. Since combinatorial terms do not involve argument references, their tree representations are subtree-closed.

Since the names used for argument references in the notation are not part of the term, the following equations hold:

$$\lambda xy.fyx \;=\; \lambda yx.fxy \;=\; \lambda zy.fyz$$

In other words, the three lambda notations describe the same term although as notations they are different. On the other hand we have $\lambda xy.fyx \neq \lambda xy.fxy$, that is, the notations $\lambda xy.fyx$ and $\lambda xy.fxy$ describe different terms (draw the tree representations).

The **size $|t|$ of a term t** is the number of nodes in its tree representation, for instance, $|\lambda xy.fxy| = 7$. The **names $\mathcal{N}t$ of a term t** are the names occurring in its tree representation; for instance, $\mathcal{N}(\lambda xy.fxy) = \{f\}$.

The most interesting operation on terms is **substitution**. Substitution is needed for symbolic reasoning based on terms. For instance, substitution is needed for the formulation of the β-law (§ 1.1). Applying the substitution $[x{:=}s]$ to a term t means to replace in t all occurrences of the name x with s. We write $[x{:=}s]t$ for the term obtained by applying the substitution $[x{:=}s]$ to the term t. For example, the following equations hold:

$$\begin{aligned}
[x{:=}y]x &= y\\
[x{:=}y](fxy) &= fyy\\
[f{:=}gx](\lambda xy.fyx) &= \lambda x'y.gxyx' \;=\; \lambda yz.gxzy
\end{aligned}$$

The last equation is particularly interesting. It shows that names used for argument references may have to be changed if a substitution is applied. The reason

is that the application of a substitution is defined on the tree representation of a term, not on its notation. Hence a name used as argument reference in the notation cannot capture a name that is imported with a substitution. For instance, we have

$$[y{:=}x](\lambda x.fxy) \;=\; (\lambda x'.fx'x) \;=\; (\lambda y.fyx) \;\neq\; (\lambda x.fxx)$$

if we assume that x, x' and y are distinct names.

We don't associate an extension with untyped terms. In fact, there is no function f such that the application ff (f applied to f) makes sense (because of the well-foundedness of sets).

2.2 Axiomatization

There are two complementary methods for giving a precise definition of terms:

1. The constructive approach: Give a construction of terms in set theory.[1]
2. The axiomatic approch: Give an axiomatization of terms in set theory.[2]

In 1889, Peano gave an axiomatization for the natural numbers. The real numbers are a mathematical data structure that is often introduced with the axiomatic approach. A construction of the real numbers based on the rational numbers was devised by Richard Dedekind around 1860.

To validate an axiomatization, at least one model (i.e., a construction satisfying the axioms) has to be devised.

For complex data structures, it is desirable to have an axiomatization since it has some qualities a construction doesn't have:

1. An axiomatization forces us to formulate the basic properties of the data structure explicitly. Some of these properties may not be obvious for a particular construction.
2. Since an axiomatization formulates the basic properties of the data structure explicitly, it is a better and more explicit base for proofs.
3. Proofs based on an axiomatization are automatically valid for all models of the axiomatization.
4. Since an axiomatization specifies the properties that must be satisfied by all constructions, it doesn't commit us to the details of a particular construction.
5. An axiomatization gives us the freedom to work with different constructions, as long as they satisfy the axioms.

Figures 1 and 2 show our axiomatization of untyped terms. The starting point is a set *Ter* whose elements we will call terms. Next there is a operation *N*

[1] Constructions are related to implementations in programming.
[2] Axiomatizations are related to abstract data types (ADTs) in programming.

that yields for every natural number a term. Terms obtainable with N are called names, and the set of all names is denoted by *Nam*. For terms and names we introduce, as a notational device, so-called **meta-variables**. The fact that x is declared as a meta-variable for names means that x stands for a name if not said otherwise.

The axiomatization uses a general form of substitution that can replace any number of names in parallel. Hence we model substitutions as functions $\theta \in Nam \to Ter$. If we have $\theta x = x$, the substitution θ replaces x with x, which means that it leaves the name x unchanged. The axiomatization defines *Sub* to be the set $Nam \to Ter$ of all substitutions and declares θ as meta-variable for substitutions.

Next we look at the operations of the axiomatization, which we will refer to as **syntactic operations**. With N one obtains names, with A applications, and with L abstractions. There are also operations that yield the size of a term and the names occurring in a term. Finally, there is an operation that applies a substitution to a term. The choice of the syntactical operations represents the result of a careful analysis. The operations have been chosen such that all properties of terms can be expressed in a simple way. N, A and L are needed so that we can obtain every term in finitely many steps from numbers. The size operation is needed so that we can prove properties of terms by induction on their size. $\mathbf{S}$ and $\mathcal{N}$ are needed so that we can axiomatize the abstraction operation L.

Figure 1 also shows notations for terms, most of which we have already seen. The notations st, $\lambda x.t$ and θt make it possible to not write the syntactic operations A, L and $\mathbf{S}$ explicitly. This is very convenient, but keep in mind that in case of doubt it is a good idea to write the syntactic operations explicitly.

The notation $[x{:=}t]$ stands for the substitution that maps the name x to the term t, and all other names to themselves.

Figure 2 shows the axioms of the axiomatization. Except for IL and SL, the axioms are obvious from our intuitive understanding of terms.

Axiom IN says that N is a bijection between $\mathbb{N}$ and *Nam*. This means that there are as many names as there are natural numbers, no more, no less.

Axiom IL states under which conditions two descriptions $\lambda x.t$ and $\lambda x'.t'$ yield the same term. Read from right to left, IL makes precise which variables can be used as argument variable in the description of an abstraction, and how the choice of the argument variable affects the body of the description. For instance, IL says that if t is an abstraction and x is a name that does not occur in t, then there exists exactly one term s such that $t = \lambda x.s$.

Axiom SN makes explicit use of the substitution operation $\mathbf{S}$ to avoid a notational disambiguity. It says that the application of $\mathbf{S}$ to a substitution θ and a

Meta-Variables and Sets

$s,t \in Ter$	**terms**
$x,y,z,f,g,h \in Nam := \{Nn \mid n \in \mathbb{N}\}$	**names**
$\theta \in Sub := Nam \to Ter$	**substitutions**

Operations

$N \in \mathbb{N} \to Ter$	**name**
$A \in Ter \to Ter \to Ter$	**application**
$L \in Nam \to Ter \to Ter$	**abstraction**
$\lvert\cdot\rvert \in Ter \to \mathbb{N}$	**size**
$\mathcal{N} \in Ter \to \mathcal{P}(Nam)$	**names**
$\mathbf{S} \in Sub \to Ter \to Ter$	**substitution**

Notations

$$
\begin{aligned}
st &:= Ast \\
\lambda x.t &:= Lxt \\
\lambda xy.t &:= \lambda x.\lambda y.t \\
stt' &:= (st)t' \\
\theta t &:= \mathbf{S}\theta t \\
[x{:=}t] &:= \lambda y{\in}Nam.\ \text{if } y = x \text{ then } t \text{ else } y
\end{aligned}
$$

- Terms that can be obtained with N are called **names**.
- Terms that can be obtained with A are called **applications**.
- Terms that can be obtained with L are called **abstractions**.
- A name x **occurs in a term** t if $x \in \mathcal{N}t$.

Figure 1: Axiomatization of untyped terms: sets, operations, notations

Par Every term is exactly one of the following:
a name, an application, or an abstraction.

IN $Nn = Nn' \iff n = n'$

IA $st = s't' \iff s = s' \wedge t = t'$

IL $\lambda x.t = \lambda x'.t' \iff x' \notin \mathcal{N}(\lambda x.t) \wedge t' = [x{:=}x']t$

CN $|x| = 1$

CA $|ts| = 1 + |t| + |s|$

CL $|\lambda x.t| = 1 + |t|$

NN $\mathcal{N}x = \{x\}$

NA $\mathcal{N}(st) = \mathcal{N}s \cup \mathcal{N}t$

NL $\mathcal{N}(\lambda x.t) = \mathcal{N}t - \{x\}$

SN $\mathbf{S}\theta x = \theta x$

SA $\theta(ts) = (\theta t)(\theta s)$

SL $\theta(\lambda x.t) = \lambda x.(\theta[x{:=}x])t \quad$ if $\forall y \in \mathcal{N}(\lambda x.t)\colon\ x \notin \mathcal{N}(\theta y)$

Figure 2: Axiomatization of untyped terms: axioms

name x yields the same result as the application of the function θ to x.

Axiom SL states how substitution applies to abstractions. It uses a notation defined as follows:

$$\theta[x{:=}t] \ := \ \lambda y{\in}Nam.\ \text{if } y = x \text{ then } t \text{ else } \theta y$$

Given this definition, the substitution $\theta[x{:=}x]$ behaves on all names like θ, except possibly for x, which it maps to x.

Use of the Substitution Axioms

We demonstrate the use of the substitution axioms with two examples. We assume that the names x, y, z and f are pairwise distinct. Here is the first example:

$$\begin{aligned}
[x{:=}y](\lambda z.fxz) &= \lambda z.[x{:=}y](fxz) && \text{SL}\\
&= \lambda z.([x{:=}y]f)([x{:=}y]x)([x{:=}y]z) && \text{SA}\\
&= \lambda z.fyz && \text{SN}
\end{aligned}$$

The first line also uses the axioms NN, NA and NL. Here is another example:

$$\begin{aligned}
[y{:=}x](\lambda x.\, fxy) &= [y{:=}x](\lambda z.[x{:=}z](fxy)) && \text{IL}\\
&= [y{:=}x](\lambda z.\, fzy) && \text{SA, SN}\\
&= \lambda z.[y{:=}x](fzy) && \text{SL}\\
&= \lambda z.\, fzx && \text{SA, SN}
\end{aligned}$$

Again, the axioms for $\mathcal{N}$ are used tacitly.

We will have to apply substitutions frequently in the following. Usually, we will rely on our intuition and proceed without giving a detailed proof. The point we want to make with the detailed proofs given above is that in case there is a doubt about the application of a substitutions, there is firm foundation for distinguishing between right and wrong.

Explicit Formulation of Axioms

The formulation of the axioms leaves two things implicit:

1. The syntactic operations A, L and $\mathbf{S}$ are not written explicitly.
2. The universal quantification of the meta-variables is left implicit.

The omission of the syntactic operations was announced explicitly as a matter of notation. The omission of the universal quantifiers for meta-variables is a general convention we have not mentioned so far. As an example, we give the explicit formulation of the axiom IL:

$$\begin{aligned}
&\forall x \in \mathit{Nam}\ \forall t \in \mathit{Ter}\ \forall x' \in \mathit{Nam}\ \forall t' \in \mathit{Ter}:\\
&\quad Lxt = Lx't' \iff x' \notin \mathcal{N}(Lxt) \ \wedge\ t' = \mathbf{S}[x{:=}x']t
\end{aligned}$$

2.3 Basic Properties

Based on the axiomatization of terms, one can prove many properties of terms. For most proofs one needs induction on the size of terms.

Proposition 2.1 $\mathcal{N}t$ is a finite set.

Proof By induction on $|t|$. Case analysis according to Par.

Case $t = x$. Then $\mathcal{N}t = \{x\}$ by NN.

Case $t = ss'$. Then $\mathcal{N}t = \mathcal{N}s \cup \mathcal{N}s'$ by NA. By CA we can use induction for s and s' and hence know that $\mathcal{N}s$ and $\mathcal{N}s'$ are finite. Hence $\mathcal{N}t$ is finite.

Case $t = \lambda x.s$. Then $\mathcal{N}t = \mathcal{N}s - \{x\}$ by NL. By CL we can use induction for s and hence know that $\mathcal{N}s$ is finite. Hence $\mathcal{N}t$ is finite. ∎

Proposition 2.2 (Coincidence) $(\forall x \in \mathcal{N}t\colon\ \theta x = \theta' x) \implies \theta t = \theta' t$

Proposition 2.3 (Identity Substitution) $\mathbf{S}(\lambda x{\in}Nam.\, x)t = t$

Proposition 2.4 (Renaming) $x \notin \mathcal{N}t \implies [x{:=}s]([y{:=}x]t) = [y{:=}s]t$

Proposition 2.5 $x \in \mathcal{N}(\theta t) \iff \exists y \in \mathcal{N}t\colon\ x \in \mathcal{N}(\theta y)$

Proposition 2.6 $t' = [x{:=}x']t \implies (t = [x'{:=}x]t' \iff x' \notin \mathcal{N}(\lambda x.t))$

Proposition 2.7 $\lambda x.t = \lambda x'.t' \iff t = [x'{:=}x]t' \ \wedge\ t' = [x{:=}x']t$

2.4 Reduction

Here are two rewrite rules for terms:

$$(\lambda x.s)t \ \to\ [x{:=}t]s \qquad\qquad \beta\text{-rule}$$
$$\lambda x.sx \ \to\ s \quad \text{if } x \notin \mathcal{N}s \qquad\qquad \eta\text{-rule}$$

The rules correspond to the left-to-right application of the β- and η-law (§ 1.1). Here is an example where a term is rewritten three times with the β-rule:

$$\begin{aligned}(\lambda f.fa)(\lambda xy.fxy)b &\to (\lambda xy.fxy)ab && \beta\\ &\to (\lambda y.fay)b && \beta\\ &\to fab && \beta\end{aligned}$$

In the next example, a term is rewritten with the η- and the β-rule:

$$\begin{aligned}(\lambda hx.fhx)(\lambda x.gx)a &\to (\lambda hx.fhx)ga && \eta\\ &\to (\lambda h.fh)ga && \eta\\ &\to fga && \beta\end{aligned}$$

We write $s \to t$ and say that s **reduces to** t **in one step** if s can be rewritten to t by a single application of the β- or η-rule. Moreover, we write $s \to^* t$ and say that s **reduces to** t if s can be rewritten to t by finitely many applications of the β- and η-rule. Note that we have $s \to^* s$ for every term s since finitely many applications includes zero applications.

A term t is called $\beta\eta$**-normal** if it cannot be reduced, that is, neither the β-rule nor the η-rule apply to it. For instance, the term $\lambda xy.fyx$ is $\beta\eta$-normal, but the term $\lambda xy.fxy$ is not. Every combinatorial term is $\beta\eta$-normal.

A term t is called a $\beta\eta$**-normal form of a term** s if $s \to^* t$ and t is $\beta\eta$-normal. Here is a famous theorem that was shown first around 1935.

Theorem 2.8 (Chuch-Rosser) A term has at most one $\beta\eta$-normal form.

The proof of the theorem is not straightforward and will not be given here. However, it is easy to show that the theorem is sharp, that is, that there are terms that don't have a $\beta\eta$-normal form. Here we go:

$$(\lambda x.xx)(\lambda x.xx) \to (\lambda x.xx)(\lambda x.xx)$$

The example shows that rewriting with the β-rule may reproduce a term. This means that reduction doesn't terminate on all terms. Interestingly, there are terms that have a $\beta\eta$-normal form although reduction doesn't necessarily terminate on them:

$$(\lambda x.y)\Omega \to y$$
$$(\lambda x.y)\Omega \to (\lambda x.y)\Omega$$

where $\Omega = (\lambda x.xx)(\lambda x.xx)$.

Reduction of untyped terms is computationally interesting. Church and Turing showed that for every computable function there is a term such that reduction with the β-rule computes the function. The system given by untyped terms and the reduction rules β and η is often referred to as the **untyped lambda calculus**.

2.5 Subterms

We define the **subterms** of a term t recursively:

1. If $t = ss'$, then s and s' are subterms of t.
2. If $t = \lambda x.s$, then s is a subterm of t.
3. t is a subterm of t.
4. If s is a subterm of a subterm of t, then s is a subterm of t.

A term s is a **proper subterm** of term t if $s \neq t$ and s is a subterm of t. Here are examples:

- The proper subterms of fxy are fx, f, x and y.
- s is a proper subterm of $\lambda x.x$ if and only if s is a name.
- $\lambda x.y$ has only one proper subterm, provided $x \neq y$.

Proposition 2.9 A term is combinatorial if and only if none of its subterms is an abstraction.

Proposition 2.10 A term has infinitely many subterms if and only if it has a subtem $\lambda x.s$ with $x \in \mathcal{N}s$.

2.6 Recursive Definitions and the Canonical Name Convention

Terms are a recursive data structure. This is a consequence of the fact that every term can be obtained in finitely many steps from names with the operations A and L. Since terms are a recursive data structure, functions on terms must usually be defined by recursion of the structure of terms. Here is the recursive definition of a function that yields the depth of a term:

$$depth \in Ter \to \mathbb{N}$$

$$depth\, x = 0$$
$$depth(st) = 1 + \max\{depth\, s,\ depth\, t\}$$
$$depth(\lambda x.s) = 1 + depth\, s$$

Look carefully at this definition. The case analysis is exhaustive and the recursion is terminating. However, the last equation is not disjoint since for every abstraction t there are infinitely many pairs x, s such that $t = \lambda x.s$. This means that the definition, as it stands, is not admissible. We fix the problem by choosing for every abstraction t a name $x \notin \mathcal{N}t$, and by constraining the last equation to use this **canonical name**:

$$depth(\lambda x.s) = 1 + depth\, s \qquad \text{if } x \text{ is the canonical name for } \lambda x.s$$

Now that the definition of *depth* is fixed, we look again at the unconstrained equation

$$depth(\lambda x.s) = 1 + depth\, s$$

Intuitively, it is clear that the equation holds for all names x and all terms s. But how can we prove this? By Axiom IL, we know that it suffices to show

$$depth([x{:=}y]s) = depth\, s$$

for all names x, y and all terms s. We prove the following, more general property.

Claim $\forall t \in Ter\ \forall \theta \in Sub\colon\ Ran\, \theta \subseteq Nam \Longrightarrow depth(\theta t) = depth\, t$

Proof By induction on $|t|$. Let $Ran\ \theta \subseteq Nam$. To show: $depth(\theta t) = depth\ t$. Case analysis.

Case $t = x$. Exercise.

Case $t = ss'$. Exercise.

Case $t = \lambda x.s$ where x is the canonical name for t. By the definition of *depth* it suffices to show $depth(\theta t) = 1 + depth\ s$. We choose a name y such that

$t = \lambda y.[x{:=}y]s$ and $\theta t = \lambda y.(\theta[y{:=}y])([x{:=}y]s)$. Let z be the canonical name for θt. Then $\theta t = \lambda z.[y{:=}z]((\theta[y{:=}y])([x{:=}y]s))$. Let

$$\theta' := \lambda n{\in}\mathcal{N}.[y{:=}z]((\theta[y{:=}y])([x{:=}y]n))$$

Then $\theta t = \lambda z.\theta' s$ and $Ran\ \theta' \subseteq Nam$. By induction $depth(\theta' s) = depth\ s$. Thus $depth(\theta t) = depth(\lambda z.\theta' s) = 1 + depth(\theta' s) = 1 + depth\ s$. ∎

In the following we will use the **canonical name convention (CNC)**:

1. In a defining equation $f(\lambda x.s) = \cdots$, the name x is always the canonical name for the abstraction $\lambda x.s$. This constraint applies without mentioning.
2. Defining equations $f(\lambda x.s) = \cdots$ will always be chosen such that they are valid for all names that can be used as argument references, not just the canonical ones. This fact will be used tacitly and proofs will not be given (they tend to be tedious, see above).

To give an example that violates the second requirement of the CNC, we choose a name a and define a function

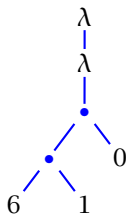
$$foo \in Ter \to \mathbb{N}$$
$$foo\ x = (x = a)$$
$$foo(st) = foo\ s + foo\ t$$
$$foo(\lambda x.s) = foo\ s \qquad \text{if } x \text{ canonical for } \lambda x.s$$

The definition of *foo* is fine. However, the defining equation for abstractions does not hold unconstrained. To see this, let b be a name different from a. Since we have $\lambda a.a = \lambda b.b$, we can obtain a contradiction as follows:

$$1 = (a = a) = foo\ a = foo(\lambda a.a) = foo(\lambda b.b) = foo\ b = (b = a) = 0$$

2.7 Construction of a Term Structure

We now construct a structure satisfying the axiomatization of terms. Terms are modeled as trees whose leaves are natural numbers. Dangling argument references are interpreted as names. This way we obtain a tree representation of terms that is subtree-closed. For instance, the term $\lambda xy.fxy$, where f, x, and y are distinct names and $f = 4$, is represented by the following tree:



This leads to the following recursive definition:

$$Ter := \mathbb{N} \cup (\{1\} \times Ter) \cup (\{2\} \times Ter \times Ter)$$

Names are modelled as natural numbers, abstractions as pairs $(1,t)$, and applications as triples $(2,s,t)$. The operation N and A are defined as follows:

$$\begin{aligned} Nn &:= n \\ Ast &:= (2,s,t) \end{aligned}$$

The definition of the operation L is not as straightforward and will be given once we have defined the substitution operator. We will write λt for $(1,t)$ and st for $(2,s,t)$. The definition of the size operation is by recursion on the term structure:

$$\begin{aligned} |x| &= 1 \\ |\lambda s| &= 1 + |s| \\ |st| &= 1 + |s| + |t| \end{aligned}$$

The definition of $\mathcal{N}$ is more involved since $\mathcal{N}$ must distinguish between argument references and names. To do this for a leaf marked with the number x, we need to know the number d of lambdas on the path from the leaf to the root of the tree. If $x < d$, the leaf represents an argument reference. If $x \geq d$, the leaf represents the name $x - d$. We can define $\mathcal{N}$ with an auxiliary function $\mathcal{N}' : \mathbb{N} \to Ter \to \mathcal{P}(Ter)$:

$$\begin{aligned} \mathcal{N}\, s &= \mathcal{N}'\, 0\, s \\ \mathcal{N}'\, d\, x &= \text{if } x < d \text{ then } \emptyset \text{ else } \{x - d\} \\ \mathcal{N}'\, d\, (\lambda s) &= \mathcal{N}'\, (d+1)\, s \\ \mathcal{N}'\, d\, (st) &= \mathcal{N}'\, d\, s \cup \mathcal{N}'\, d\, t \end{aligned}$$

The substitution operator $\mathbf{S}$ is defined in a similiar way with an auxiliary function $\mathbf{S}' : \mathbb{N} \to Sub \to Ter \to Ter$:

$$\begin{aligned} \mathbf{S}\, \theta\, s &= \mathbf{S}'\, 0\, \theta\, s \\ \mathbf{S}'\, d\, \theta\, x &= \text{if } x < d \text{ then } x \text{ else } shift\, d\, (\theta(x - d)) \\ \mathbf{S}'\, d\, \theta\, (\lambda s) &= \lambda(\mathbf{S}'\, (d+1)\, \theta\, s) \\ \mathbf{S}'\, d\, \theta\, (st) &= (\mathbf{S}'\, d\, \theta\, s)(\mathbf{S}'\, d\, \theta\, t) \end{aligned}$$

The first equation for $\mathbf{S}'$ uses an auxiliary function $shift : \mathbb{N} \to Ter \to Ter$ that increments the name references of a term by a given number. We can define $shift$ mutually recursive with $\mathbf{S}$:

$$shift\, d\, x = x + d$$

$$\mathit{shift}\, d\, s = \mathbf{S}\,(\lambda n{\in}\mathbb{N}.\, n + d)\, s \quad \text{if } s \text{ is not a name}$$

The mutual recursion terminates since *shift* handles names directly and uses **S** only with substitutions that replace names with names. For such substitutions, **S′** uses *shift* only for names.

With the substitution operation **S** it is easy to define the abstraction operation L:

$$L\, x\, s = \lambda(\mathbf{S}\,(\lambda n{\in}\mathbb{N}.\, \text{if } n = x \text{ then } 0 \text{ else } n + 1)\, s)$$

This completes our construction of a term structure. We will skip the proofs showing that the axioms are satisfied. The construction was invented by Nicolaas de Bruijn in the late 1960's to be used for the implementation of his language Automath.

Exercise 2.11 Implement the term structure in a functional programming language.

2.8 Typed Terms

The language of higher-order logic uses typed terms so that it can syntacticly ensure that functions are only applied to admissible arguments. This way, meaningless applications like $5x$ or $f\,f$ are excluded.

We start with a set *Sor* of primitive types, which we call **sorts**. The set *Ty* of **types** is the least set that contains all sorts and is closed under pairing of types:

$$A, B \in \mathit{Sor} \qquad \textbf{sorts}$$
$$S, T \in \mathit{Ty} \;:=\; \mathit{Sor} \cup (\mathit{Ty} \times \mathit{Ty}) \qquad \textbf{types}$$

We assume that *Sor* and $\mathit{Ty} \times \mathit{Ty}$ are disjoint sets and arrange the following notations:

$$S \to T \;:=\; (S, T)$$
$$S \to S' \to T \;:=\; S \to (S' \to T)$$

Types of the form $S \to T$ are called **functional types**. Every type can be uniquely written in the form $S_1 \to \cdots \to S_n \to A$ where $n \geq 0$ and A is a sort. We may think of the elements of a type $S_1 \to \cdots \to S_n \to A$ as operations that take n arguments of the types $S_1, \ldots, S_n$ and yield a result of the sort A. (Note that we have not yet defined what an element of a type is.)

To arrive at typed terms, we modify our axiomatization of terms such that names are obtained from numbers and types:

$$N \in \mathbb{N} \to \mathit{Ty} \to \mathit{Ter}$$

$$Nam := \{ NnT \mid n \in \mathbb{N},\ T \in Ty \}$$

The axiom IN now takes the form

IN $\quad NnT = Nn'T' \iff n = n' \wedge T = T'$

It ensures that the type of a name is unique and that there are infinitely many names for every type. Moreover, we include a syntactical operation

$$\tau \in Nam \to Ty$$

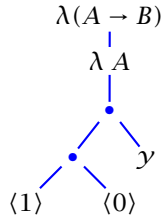
that satisfies the axiom

Tau $\quad \tau(NnT) = T$

and thus yields the type of a name. Furthermore, we extend the Axiom IL as follows:

IL $\quad \lambda x.t = \lambda x'.t' \iff \tau x = \tau x' \ \wedge\ x' \notin \mathcal{N}(\lambda x.t) \ \wedge\ t' = [x{:=}x']t$

The axiom now ensures that an abstraction knows the type of its argument, and that only names of this type can be used as argument references. Hence, we annotate every lambda node in the tree representation of a typed term with the argument type of the corresponding abstraction. For instance, if we have $\tau f = A \to B$ and $\tau x = A$, then the tree representation of the typed term $\lambda f x.\, f x y$ looks as follows:



Finally, we need an axiom that ensures that the sets *Ter* and *Ty* are disjoint:

Dis $\quad Ter \cap Ty = \emptyset$

This completes our axiomatization of typed terms.

2.9 Well-typed Terms

Next we define a predicate $(:) \in Ter \to Ty \to \mathbb{B}$ that we will use to define the notion of a well-typed term. We define $(:)$ such that $(t : T) = 1$ if and only if the term t is well-typed and has the type T:

$$(:) \in Ter \to Ty \to \mathbb{B}$$

$$(x : S) = (\tau x = S)$$
$$(st : S) = (\exists T \in Ty\colon\ (S : T \to S)\ \wedge\ (t : T))$$
$$(\lambda x.s : S) = (\exists T \in Ty\colon\ S = \tau x \to T\ \wedge\ (s : T))$$

Note that this definition complies with the CNC. We can now define well-typed terms and substitutions:

- A term t is **well-typed** if there exists a type T such that $t : T$.
- A substitution θ is **well-typed** if $\theta x : \tau x$ for all names x.

Every well-typed term has exactly one type:

Proposition 2.12 (Unique Types) $s : S\ \wedge\ s : T\ \Longrightarrow\ S = T$

If s is a well-typed term, we call the unique type S sucht that $s : S$ **the type of** s and denote is with τs.

There is a straightforward algorithm that checks whether a term is well-typed and yields its type if this is the case. The algorithm can be illustrated on the tree representation of a term. First, every leaf is attributed with a type. If the leaf is a name, the type is obtained with τ. If the leaf is an argument reference, the type is obtained from the λ-node the argument reference refers to. Now the types are propagated upwards to the root. At a node $\lambda(S)$ nothing can go wrong: The node receives the type $S \to T$ where T is the type of the node below. At an application node, there is a constraint: The type from the left son must be a functional type $S \to T$ and the type from the right son must be S. If this is the case, the application node receives the type T.

The next proposition says that applications of well-typed substitutions and reduction steps with the β- and η-rule preserve well-typedness and, even stronger, preserve the type of a term.

Proposition 2.13 (Type Preservation)

1. $s : S\ \wedge\ \theta$ well-typed $\Longrightarrow\ \theta s : S$
2. $s : S\ \wedge\ s \to t\ \Longrightarrow\ t : S$

The next theorem is difficult to prove. It says that applying the β- and η-rule to a well-typed term will always yield a $\beta\eta$-normal form after finitely many steps.

Theorem 2.14 (Tait 1967) Reduction of well-typed terms terminates.

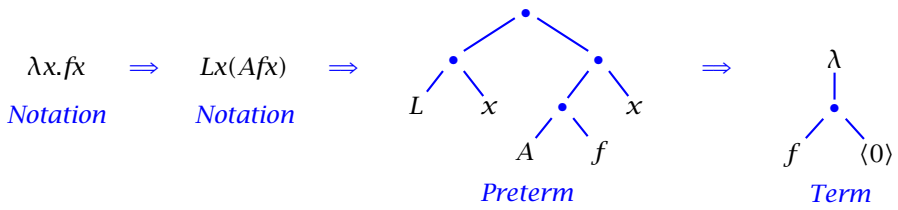
Together with the Church-Rosser Theorem, Tait's theorem yields the following corollary.

Corollary 2.15 Every well-typed term has exactly one $\beta\eta$-normal form.

Tait's theorem gives us a straightforward algorithm for computing the $\beta\eta$-normal form of a term: Apply the rules as long as they are applicable. The number of reduction steps needed to reach the normal form may be huge. Rick Statman [1979] showed that there is no elementary recursive bound with respect to the size of the term.

2.10 Preterms

By now, we are used to interpret a notation like $\lambda x.fx$ as a description of a term. To discuss some fine points, we will introduce a further interpretation, which sees a notation like $\lambda x.fx$ as a description of a so called preterm. We will ignore types. The situation is best explained with a picture:



The preterm is a direct representation of the notation obtained with the syntactic operations A and L. The preterm contains more information than the term. In fact, given a term structure, we can see the preterm is the intension and the term as is the extension of the notation $\lambda x.fx$. This means that evaluation of the preterm will yield the term. The main difference between preterms and terms is the fact that preterms model argument references implicitly with names while terms model argument references explicitly with numbers.

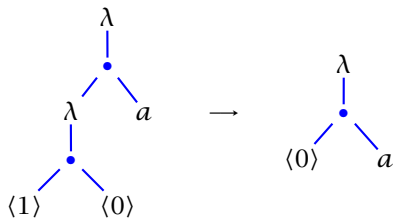
Church based higher-order logic on preterms. It took some time until the notion of a term was invented by de Bruijn. The advantage of preterms is that they are close to the usual mathematical notation. The problem with preterms is that they lead to a rather complex definition of the substitution operation. For instance, it is not even clear, which preterm the application of the substitution $[x{:=}y]$ to the preterm $\lambda y.x$ should yield.

Church introduced **α-equivalence of preterms**. Two preterms are α-equivalent if they are equal up to consistent renaming of the names used for argument references. Based on preterms, it is a tedious enterprise to formally define what is meant by consistent renaming. With terms, however, we obtain a straightforward characterization of α-equivalence: Two preterms are α-equivalent if and only if they evaluate to the same term.

One can use a more direct notation for terms than the usual one. For instance, the term $\lambda x.fx$ can be described more directly with the notation $\lambda(f\langle 0\rangle)$, and the term $\lambda f.(\lambda x.fx)a$ with the notation $\lambda(\lambda(\langle 1\rangle\langle 0\rangle)a)$. Technically, it would

be preferable to use the direct notation but most people prefer the conventional notation. The picture changes when we implement terms. Then the conventional notation is only used at the user interface and all internal processing is done with the direct notation. The construction of the term structure in § 2.7 gives us a good idea how this looks like.

The presentation of reduction in § 2.4 is based on the conventional notation for terms. For instance, we have the reduction $\lambda f.(\lambda x.fx)a \to \lambda f.fa$. If we switch to the direct notation, we get $\lambda(\lambda(\langle 1\rangle\langle 0\rangle)a) \to \lambda(\langle 0\rangle a)$, or, graphically,



Canonical names give us a means to single out canonical preterms. We call a preterm **canonical** if it employs only canonical names as argument references. There exists exactly one canonical preterm for every term. Hence we can construct a term structure based on preterms.[3]

Exercise 2.16

a) Give 2 different preterms for the term $\lambda x.x$.

b) How many preterms exist for the term $\lambda x.x$?

c) How many preterms exist for the term $f(gx)$?

Exercise 2.17 Let a term structure with $Nam = \mathbb{N}$ be given and assume that the canonical name for an abstraction is the least name that does not occur in the abstraction. Then $L\,0\,0$ and $L\,0\,(L\,1\,(A\,0\,1))$ are canonical preterms that evaluate to the terms $\lambda x.x$ and $\lambda fx.fx$. Give the canonical preterms for the following terms.

a) $\lambda xy.y$

b) $\lambda xy.x$

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

[3] Details can be found in Allen Stoughton, Substitution Revisited, Theoretical Computer Science, 59:317-325, 1988.

Specification	Group	
Sort	G	
Constants	$\cdot : G \to G \to G$	
	$e : G$	
	$i : G \to G$	
Axioms	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	*Associativity*
	$e \cdot x = x$	*Identity*
	$ix \cdot x = e$	*Inversion*

Figure 3: Specification of groups

3 Specifications and Models

Higher-Order logic (HOL) is a specification language based on well-typed terms. It can specify abstract structures like groups, but also concrete data structures like the natural numbers.

3.1 Motivating Example: Groups

The specification shown in Figure 3 specifies a class of mathematical structures known as groups. According to this specification, a *group* consists of a set G and three objects

$$\cdot \in G \to G \to G$$
$$e \in G$$
$$i \in G \to G$$

such that for all $x, y, z \in G$ the following equations hold:

$$(x \cdot y) \cdot z = x \cdot (y \cdot z) \qquad e \cdot x = x \qquad ix \cdot x = e$$

A concrete group is obtained if we take the set $\mathbb{Z}$ of natural numbers as G, addition as $\cdot$, 0 as e, and $\lambda x{\in}\mathbb{Z}.\ -x$ as i.

Speaking generally, the groups are the *models* of the group specification. The group specification is useful since (1) many concrete structures contain groups and (2) we can use the specification to prove properties of all groups. Here is a typical example.

Proposition 3.1 For every group G and every $x \in G$ the equation $x \cdot ix = e$ holds.

Proof Let G be a group and $x \in G$. Then:

$$\begin{aligned}
x \cdot ix &= (e \cdot x) \cdot ix && \text{Identity} \\
&= ((i(ix) \cdot ix) \cdot x) \cdot ix && \text{Inversion} \\
&= i(ix) \cdot ((ix \cdot x) \cdot ix) && \text{Associativity} \\
&= i(ix) \cdot (e \cdot ix) && \text{Inversion} \\
&= i(ix) \cdot ix && \text{Identity} \\
&= e && \text{Inversion}
\end{aligned}$$

■

Exercise 3.2 Prove that $x \cdot e = x$ holds for every group G and every $x \in G$.

In HOL, mathematical statements are represented as well-typed terms of type **B**, called *formulas*, where the sort **B** represents the set $\mathbb{B} = \{0, 1\}$. A *specification* is just a set of formulas. To express the axioms for groups as formulas, we use a name $\doteq : G \to G \to \mathbf{B}$ representing the identity predicate for the set G. Moreover, we distinguish between two groups of names: The *constants* $\cdot$, e, i, $\doteq$ on the one hand, and the *variables* x, y, z on the other hand.

In the following, we will give a precise definition of specifications and their models. This will include a definition of what is means that a formula is valid in a structure.

3.2 Interpretations and Evaluations

We will only consider well-typed terms and well-typed substitutions in this chapter. We will use *Ter* as name for the set of all well-typed terms.

The first step consists in giving a definition of what we mean by the extension of types and terms.

1. First we fix extensions for all sorts. The extension of a sort must be a non-empty set.
2. We obtain the extension of a functional type $S \to T$ as the set of all functions from the extension of S to the extension of T.
3. Next we fix type-observing extensions for all names (i.e., the extension of a name must be an element of the extension of its type).
4. The extension of an application st is the value φv, where φ is the function obtained as extension of s and v is the extension of t. Since the term st is well-typed, we can be sure that φ is in fact a function defined on v.
5. The extension of an abstraction $\lambda x.s$ is the function φ that maps every value v in the extension of τx to the value that is obtained as the extension of s, where the name x is given the extension v.

A proper definition of the extensions for types and terms requires recursion on types and terms. Moreover, the change of the extension of the name x in clause (5) asks for a more formal definition.

First we address clauses (1), (2) and (3). An **interpretation** is a function $\mathcal{I}$ such that:

1. $Dom\,\mathcal{I} = Ty \cup Nam$
2. $\mathcal{I}(S \to T) = \{\, \varphi \mid \varphi \text{ function } \mathcal{I}S \to \mathcal{I}T \,\}$
3. $\mathcal{I}x \in \mathcal{I}(\tau x)$

Condition (2) is required for all types S and T, and condition (3) is required for all names x.

We will use the word **atom** to refer to both sorts and names.

Proposition 3.3 (Coincidence) If $\mathcal{I}$ and $\mathcal{I}'$ agree on all atoms, then $\mathcal{I} = \mathcal{I}'$.

Proof We need to show: $\forall T\colon\ \mathcal{I}T = \mathcal{I}'T$. This can be done by induction on $|T|$. ∎

Proposition 3.4 $\mathcal{I}T \neq \emptyset$.

Proof Let $\mathcal{I}$ be an interpretation and T be a type. By Axiom Inf we know that there is a name x with $\tau x = T$. Hence $\mathcal{I}x \in \mathcal{I}(\tau x) = \mathcal{I}T$. ∎

Given an interpretation $\mathcal{I}$, a name x and a value $v \in \mathcal{I}(\tau x)$, we use $\mathcal{I}_{x,v}$ to denote the interpretation $\mathcal{I}[x{:=}v]$. Note that $\mathcal{I}_{x,v}$ satisfies the following equations:

$$\mathcal{I}_{x,v}T = \mathcal{I}T$$
$$\mathcal{I}_{x,v}y = \mathit{if}\ y = x\ \mathit{then}\ v\ \mathit{else}\ \mathcal{I}y$$

Proposition 3.5 (Evaluation) For every interpretation $\mathcal{I}$ there exists one and only one function $\hat{\mathcal{I}}$ such that:

1. $Dom\,(\hat{\mathcal{I}}) = Ter$
2. $\hat{\mathcal{I}}t \in \mathcal{I}(\tau t)$
3. $\hat{\mathcal{I}}x = \mathcal{I}x$
4. $\hat{\mathcal{I}}(st) = (\hat{\mathcal{I}}s)(\hat{\mathcal{I}}t)$
5. $\hat{\mathcal{I}}(\lambda x.t) = \lambda v{\in}\mathcal{I}(\tau x).\ \hat{\mathcal{I}}_{x,v}t$

We call $\hat{\mathcal{I}}$ the **evaluation function** for $\mathcal{I}$.

Proof To show the existence of $\hat{\mathcal{I}}$, we define $\hat{\mathcal{I}}$ recursively according to (3), (4) and (5), where (5) is constrained to canonical names. The properties (1), (2) and the uniqueness of $\hat{\mathcal{I}}$ are immediate consequences of this definition. The unconstrained version of (5) can be shown with Proposition 3.6. The proof of this proposition must be based on the recursive definition of $\hat{\mathcal{I}}$. ∎

Given an interpretation $\mathcal{I}$ and a substitution θ, we use $\mathcal{I}_\theta$ to denote the interpretation defined as follows:

$$\mathcal{I}_\theta T = \mathcal{I} T$$
$$\mathcal{I}_\theta x = \hat{\mathcal{I}}(\theta x)$$

Proposition 3.6 (Substitution) $\hat{\mathcal{I}}(\theta t) = \hat{\mathcal{I}}_\theta t$.

Proof By induction on $|t|$. Tedious. ∎

The **atoms occurring in a type or term** are defined as follows:

$$\begin{aligned} Atom\, A &= \{A\} \\ Atom\,(S \to T) &= Atom\, S \cup Atom\, T \\ Atom\,(x) &= \{x\} \cup Atom\,(\tau x) \\ Atom\,(st) &= Atom\, s \cup Atom\, t \\ Atom\,(\lambda x.t) &= Atom\,(\tau x) \cup (Atom\, t - \{x\}) \end{aligned}$$

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

Proposition 3.7 (Coincidence) If $\mathcal{I}$ and $\mathcal{I}'$ agree on $Atom\, t$, then $\hat{\mathcal{I}} t = \hat{\mathcal{I}}' t$.

3.3 Semantic Equivalence and Correctness of the β- and η-Law

Two terms s and t are **semantically equivalent** if they have the same type and $\hat{\mathcal{I}} s = \hat{\mathcal{I}} t$ for every interpretation $\mathcal{I}$.

Recall the statement of the β-law in § 1.1. There we had no definition of terms, substitutions, and the extension of terms. Hence the β-law appeared as basic law one just had to believe in. Given our formalization of terms and evaluation, the β-law has turned into an ordinary mathematical statement that we can prove.

Proposition 3.8 (β-Law) $(\lambda x.s)t$ and $[x := t]s$ are semantically equivalent if $(\lambda x.s)t$ is a term.

Proof Let $\mathcal{I}$ be an interpretation and $(\lambda x.s)t$ be a term. Then:

$$\begin{aligned} \hat{\mathcal{I}}((\lambda x.s)t) &= (\lambda v \in \mathcal{I}(\tau x).\ \hat{\mathcal{I}}_{x,v} s)(\hat{\mathcal{I}} t) && \text{Proposition 3.5} \\ &= \hat{\mathcal{I}}_{x,\hat{\mathcal{I}} t}(s) \\ &= \hat{\mathcal{I}}_{[x:=t]}(s) \\ &= \hat{\mathcal{I}}([x := t]s) && \text{Proposition 3.6} \end{aligned}$$

∎

The alert reader will notice that the proof uses the β-law at the meta level when it proceeds from the first to the second line. So what the proof shows is just the validity of the β-law in our term-based reconstruction of mathematical language. This is good enough since just this was claimed. What we prove are properties of our reconstruction, and for the proofs we use ordinary mathematical reasoning, which includes the β-law. Studying reconstructions of mathematical language and mathematical reasoning by means of mathematical language and mathematical reasoning is what mathematical logic does, and this approach has turned out to be highly successful, both as it comes to insight and to practical applications.

Exercise 3.9 (η-Law) Prove that $\lambda x.sx$ and s are semantically equivalent if $\lambda x.sx$ is a term such that x does not occur in s.

Proposition 3.10 If s is obtained from t by $\beta\eta$-reduction, then s and t are semantically equivalent.

Proof Follows with Proposition 3.8 and Exercise 3.9. ∎

3.4 Formulas and Specifications

To use the language of types and terms as specification language, we have to built-in some logical primitives. We follow Henkin and choose the identities. To do so, we fix in the underlying term structure a sort **B** (read bool) and, for every type T, a name $\doteq_T$ whose type is $T \to T \to \mathbf{B}$. We call the name $\doteq_T$ the **identity for** T. Moreover, we tacitly assume that all interpretations $\mathcal{I}$ give **B** and the identities their canonical extensions:

$$\begin{aligned} \mathcal{I}\,\mathbf{B} &= \mathbb{B} \\ \mathcal{I}(\doteq_T) &= \lambda u{\in}\mathcal{I}T.\,\lambda v{\in}\mathcal{I}T.\,(u{=}v) \end{aligned}$$

Terms of type **B** are called **formulas**. We use the notation

$$s = t \;:=\; (\doteq_T)st \qquad \text{if } s : T \text{ and } t : T$$

and call formulas of the form $s = t$ **equations**. The sort **B** and the identities are jointly referred to as **fixed atoms**; all other atoms are called **unfixed**.

We also tacitly assume that all substitutions respect the identities, that is, $\theta(\doteq_T) = (\doteq_T)$ for every substitution and every identity.

Finally, we partition the set of names in two subsets *Var* and *Par* whose elements we call **variables** and **parameters**. The partition is chosen such that there are infinitely many variables and infinitely many parameters for every type.

A **signature** is a set Σ of unfixed atoms such that every unfixed sort occurring in the type of a parameter of Σ is in Σ. A type is **licensed** by a signature if every unfixed sort occurring in the type is in the signature. A term is **licensed** by a signature if every unfixed sort and every unfixed parameter occurring in the term is in the signature.

A **structure** is a function $\mathcal{A}$ such that $\mathit{Dom}\,\mathcal{A}$ is a signature and there exists an interpretation $\mathcal{I}$ such that $\mathcal{A} \subseteq \mathcal{I}$. Given a structure $\mathcal{A}$, we use $\Sigma_{\mathcal{A}} := \mathit{Dom}\,\mathcal{A}$ to denote the **signature of** $\mathcal{A}$. An interpretation $\mathcal{I}$ is **licensed by a structure** $\mathcal{A}$ if $\mathcal{A} \subseteq \mathcal{I}$. A type or a term is **licensed by a structure** if it is licensed by the signature of the structure.

Proposition 3.11 (Coincidence) Let $\mathcal{I}$ and $\mathcal{I}'$ be licensed by $\mathcal{A}$. Then:

1. If a type T is licensed by $\mathcal{A}$, then $\mathcal{I}T = \mathcal{I}'T$.
2. If term t is licensed by $\mathcal{A}$ and $\mathcal{I}$ and $\mathcal{I}'$ agree on all variables occurring in t, then $\hat{\mathcal{I}}t = \hat{\mathcal{I}}'t$.

A structure $\mathcal{A}$ **satisfies** a formula s if $\hat{\mathcal{I}}s = 1$ for all interpretations licensed by $\mathcal{A}$. We write $\mathcal{A} \models s$ if $\mathcal{A}$ satisfies s. We also say that s **is valid in** $\mathcal{A}$ if $\mathcal{A}$ satisfies s. It is the definition of $\mathcal{A} \models s$ where the difference between variables and parameters becomes apparent: The structure $\mathcal{A}$ must satisfy the formula s for all possible values of the variables, while it can fix the extensions of parameters and sorts.

A **specification** is a set of formulas. The formulas of a specification are called the **axioms of the specification**, and the parameters occurring in the axioms of a specification are called the **constants of the specification**. We use CA to denote the set of all constants of a specification A. An atom **occurs** in a specification if it occurs in an axiom of the specification. The **signature of a specification** is the set of all unfixed sorts and all unfixed parameters that occur in the specification. We use Σ_A to denote the signature of a specification A. A type or a term is **licensed by a specification** if it is licensed by the signature of the specification.

A **structure satisfies a specification** if it satisfies every formula of the specification. We write $\mathcal{A} \models A$ to say that a structure $\mathcal{A}$ satisfies a specification A.

A **model** of a specification is a structure that satisfies the specification and interprets exactly the unfixed atoms occurring in the specification.

You should now verify that the models of the specification Group in Figure 3 are exactly the structures that are called groups. So we have succeeded in constructing a specification language that can specify groups.

We say that a specification A **semantically entails** a formula s and write $A \models s$ if every structure that satisfies A also satisfies s.

Proposition 3.12 $A \models s \iff$ every model of A satisfies s.

Convince yourself that Proposition 3.1 states that the specification *Group* entails the formula $x \cdot ix = e$.

Proposition 3.13 (Semantic Equivalence) Let s and t be terms of the same type. Then s and t are semantically equivalent if and only if $\emptyset \models s = t$.

The **kernel of a substitution** contains all names that are changed by the substitution:

$$Ker\theta \;:=\; \{\, x \in Nam \mid \theta x \neq x \,\} \qquad \textbf{Kernel of } \theta$$

Note that the kernel of the identity substitution is empty.

Proposition 3.14 (Stability) $A \models s \;\wedge\; Ker\theta \cap CA = \emptyset \;\Longrightarrow\; A \models \theta s$

Here is a summary of the three relations that we denote with the symbol $\models$:

$$\begin{array}{rcll} \mathcal{A} \models s & \Longleftrightarrow & \forall \mathcal{I}\colon\ \mathcal{A} \subseteq \mathcal{I} \Longrightarrow \hat{\mathcal{I}} s = 1 & \text{structure satisfies formula} \\ \mathcal{A} \models A & \Longleftrightarrow & \forall s \in A\colon\ \mathcal{A} \models s & \text{structure satisfies specification} \\ A \models s & \Longleftrightarrow & \forall \mathcal{A}\colon\ \mathcal{A} \models A \Longrightarrow \mathcal{A} \models s & \text{specification entails formula} \end{array}$$

The notion of semantic entailment was first defined by Alfred Tarski around 1930.

3.5 Boolean Algebras

George Boole was a self-taught mathematician who wanted to axiomatize the laws of thought (this is the title of his famous book from 1854). What he came up with was essentially the specification BA shown in Figure 4. Boole thought of the sort D as the domain of truth values, of 0 as false, 1 as true, $+$ as "or", $\cdot$ as "and", and $\bar{}$ as "not". He thought it possible that there are more than 2 truth values. As it turned out, Boole's axioms are also satisfied by the set operations intersection, union and complement. Historically, Boole's work was the first investigation of abstract algebras, and it preceded Cantor's invention of set theory.

The models of BA are known as **Boolean Algebras**. The **two-valued Boolean algebra** $\mathcal{T}$ is the structure that interprets the sort D as $\mathbb{B} = \{0, 1\}$, the constants 0 and 1 as their names suggest, the functional constant $\bar{}$ as negation, and $+$ and $\cdot$ as disjunction and conjunction. It is easy to see that $\mathcal{T}$ is a model of BA.

We now come to the models of BA that interpret the functional constants as set operations. To obtain such a model, we start from any set X. Now we interpret the sort D as the set of all subsets of X (the power set of X). The basic constants 0 and 1 are interpreted as $\emptyset$ and X. The functional constants $\bar{}$, $+$, and $\cdot$

Specification	BA	
Sorts	D	
Constants	$0, 1 : D$	
	$\bar{} : D \to D$	
	$+, \cdot : D \to D \to D$	
Axioms		
Commutativity	$x \cdot y = y \cdot x$	$x + y = y + x$
Associativity	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	$(x + y) + z = x + (y + z)$
Distributivity	$x \cdot (y + z) = x \cdot y + x \cdot z$	$x + y \cdot z = (x + y) \cdot (x + z)$
Identity	$x \cdot 1 = x$	$x + 0 = x$
Complement	$x \cdot \bar{x} = 0$	$x + \bar{x} = 1$

Figure 4: Specification of Boolean algebras

are interpreted as the set operations complement with respect to X, union and intersection. The verification that the thus obtained structure $\mathcal{P}_X$ is a model of BA is not difficult. The Boolean algebras $\mathcal{P}_X$ are known as **power set algebras**.

Exercise 3.15 How would you prove BA $\not\models 0 = 1$?

Here is a well-known example of a law that holds in Boolean algebras.

Proposition 3.16 *BA* $\models x \cdot x = x$

Proof The proof uses the Commutativity and Associativity tacitly and mentions the use of the other axioms explicitly.

$x \cdot x = x \cdot x + 0$	Identity
$= x \cdot x + x \cdot \bar{x}$	Complement
$= x \cdot (x + \bar{x})$	Distributivity
$= x \cdot 1$	Complement
$= x$	Identity

■

A famous result of Boolean Algebra is **Stone's Representation Theorem** (1936), which says that every finite Boolean algebra is isomorphic to a power set algebra, and that every infinite Boolean algebra is isomorphic to a subalgebra of a power set algebra.

Exercise 3.17 Is there a Boolean algebra with 7 elements?

The specification BA is not minimal. In J. Eldon Whitesitt's *Boolean Algebra and its applications* (Addison Wesley, 1961) you will find a proof that the associativity axioms are semantically entailed by the other axioms.

There exist many equivalent specifications of Boolean Algebra. Here is one due to Huntington and Robbins (1933) that consists of only four axioms:

$$\begin{aligned} x + y &= y + x \\ (x + y) + z &= x + (y + z) \\ xy &= \overline{\bar{x} + \bar{y}} \\ (x + y)(x + \bar{y}) &= x \end{aligned}$$

Let's call this specification HR. It's easy to see that every model of BA is a model of HR. However, it took until 1996 that William McCune could prove the other direction with the help of an automated theorem prover. From this we learn that deciding whether two specifications have the same models can be extremely difficult.

You will find lots of interesting information about Boolean algebras in the Web (start with Wikipedia).

3.6 Specification of Logical Operations

We have seen in Chapter 1, that 0 and 1 and the logical operations can be expressed as terms that contain no other names but the identities (Henkin's reduction). The specification LD shown in Figure 5 specifies 0 and 1 and the logical operations based on this insight. If X is a non-empty set, then LD has exactly one model that interprets D as X. Note that LD uses the identities for $\mathbf{B}$, $\mathbf{B} \to \mathbf{B} \to \mathbf{B}$, $(\mathbf{B} \to \mathbf{B} \to \mathbf{B}) \to \mathbf{B}$, and $(D \to \mathbf{B}) \to \mathbf{B}$.

It is possible to be even more minimal and specify 0, 1 and the logical operations just with the identity for $\mathbf{B}$. This is done with specification LA in Figure 6. As we will see, given the interpretation of the sort D, the specification LD has exactly one model. In this model, 0, 1, $\to$, and $\forall$ are interpreted as announced, and $\cong$ is interpreted as the identity predicate for the interpretation of D.

To prove the claim, we look at the specification in the usual mathematical mode. Let a set D and objects

$$\begin{aligned} &a, b \in \mathbb{B} \\ &\quad \to \in \mathbb{B} \to \mathbb{B} \to \mathbb{B} \\ &\quad \forall \in (D \to \mathbb{B}) \to \mathbb{B} \\ &\quad \cong \in D \to D \to \mathbb{B} \end{aligned}$$

Specification	LD	
Sort	D	
Constants	$0, 1 : \mathbf{B}$	
	$\neg : \mathbf{B} \to \mathbf{B}$	
	$\wedge, \vee, \to, \leftrightarrow : \mathbf{B} \to \mathbf{B} \to \mathbf{B}$	
	$\forall, \exists : (D \to \mathbf{B}) \to \mathbf{B}$	
Axioms	$0 = (\lambda xy.x = \lambda xy.y)$	where $x : \mathbf{B}$
	$1 = (0 = 0)$	
	$\neg x = (x = 0)$	
	$x \wedge y = (\lambda f.fxy = \lambda f.f11)$	where $f : \mathbf{B} \to \mathbf{B} \to \mathbf{B}$
	$x \vee y = \neg(\neg x \wedge \neg y)$	
	$x \to y = \neg x \vee y$	
	$x \leftrightarrow y = (x \to y) \wedge (y \to x)$	
	$\forall f = (f = \lambda x.1)$	
	$\exists f = \neg(f = \lambda x.0)$	

Figure 5: Specification of logical operations by explicit definition

Specification	LA	
Sort	D	
Constants	$0, 1 : \mathbf{B}$	
	$\to : \mathbf{B} \to \mathbf{B} \to \mathbf{B}$	
	$\forall : (D \to \mathbf{B}) \to \mathbf{B}$	
	$\cong : D \to D \to \mathbf{B}$	
Axioms	$x = (x = 1)$	A1
	$1 \to x = x$	I1
	$f0 \to f1 \to fx$	*BCA* (*Boolean case analysis*)
	$\forall(\lambda x.1)$	$\forall 1$
	$\forall f \to fx$	$\forall$ I (*Instantiation*)
	$x \cong x$	*Ref* (*Reflexivity*)
	$(x \cong y) \to fx \to fy$	*Rep* (*Replacement*)

Figure 6: Specification of logical operations by axiomatization

be given, such that for all $x \in \mathbb{B}$, all $f \in \mathbb{B} \to \mathbb{B}$, all $y, z \in D$, and all $g \in D \to \mathbb{B}$ the following equations hold:

$x = (x = b)$	A1
$b \to x = x$	I1
$fa \to fb \to fx = 1$	BCA
$\forall(\lambda d{\in}D.b) = 1$	$\forall$1
$\forall g \to gd = 1$	$\forall$ I
$y \cong z = 1$	Ref
$(y \cong z) \to gy \to gz = 1$	Rep

Now we prove the following.

1. $b = 1$. Follows with A1: $b = (b = b) = 1$.
2. $a = 0$. Assume $a = 1$. Then we obtain a contradiction as follows:

$1 = a \to b \to 0$	BCA with $f = \lambda x{\in}\mathbb{B}.x$ and $x = 0$
$= 1 \to 1 \to 0$	$b = 1$, $a = 1$
$= 0$	I1

3. $0 \to x = 1$ for all $x \in \mathbb{B}$. Let $x \in \mathbb{B}$. Then:

$0 \to x = a \to b \to x$	$a = 0$ and I1
$= 1$	BCA with $f = \lambda x{\in}\mathbb{B}.x$

4. $\forall f = 0$ for all f, y such that $fy = 0$. Let $fy = 0$. Then $\forall f \to 0 = \forall f \to fy = 1$ by $\forall$ I. Hence $\forall f = 0$ by contradiction and I1.
5. $(y \cong z) = 0$ if $y \neq z$. Let $y \neq z$ and $(y \cong z) = 1$. Then we obtain a contradiction as follows:

$1 = (y \cong z) \to (y = y) \to (z = y)$	Rep with $g = \lambda d{\in}D.(d{=}y)$
$= 1 \to 1 \to 0$	assumptions
$= 0$	I1

3.7 Specification of the Natural Numbers

A specification of the natural numbers was first given by Giuseppe Peano in 1889. We will use the specification Nat shown in Figure 7.

We already know that every model will give 0, 1, $\to$ and $\forall$ their canonical interpretation. The standard model interprets N as $\mathbb{N}$, o as 0, S as the successor

Specification	Nat	
Sort	N	
Constants	$0, 1 : \mathbf{B}$	
	$\rightarrow : \mathbf{B} \rightarrow \mathbf{B} \rightarrow \mathbf{B}$	
	$\forall : (N \rightarrow \mathbf{B}) \rightarrow \mathbf{B}$	
	$o : N$	
	$S : N \rightarrow N$	
	$\leq : N \rightarrow N \rightarrow \mathbf{B}$	
Axioms	$x = (x = 1)$	A1
	$1 \rightarrow x = x$	I1
	$f0 \rightarrow f1 \rightarrow fx$	*BCA*
	$\forall(\lambda x.1)$	$\forall$1
	$\forall f \rightarrow fx$	$\forall$ I
	$fo \rightarrow \forall(\lambda x.fx \rightarrow f(Sx)) \rightarrow fx$	*Ind* (*Induction*)
	$(o \leq y) = 1$	
	$(Sx \leq o) = 0$	
	$(Sx \leq Sy) = (x \leq y)$	

Figure 7: Specification of the natural numbers

function $\lambda n{\in}\mathbb{N}.\, n{+}1$, and $\leq$ with the order predicate the symbol suggests. Verify that this construction in fact yields a model..

We will show that every model interprets N as an infinite set and satisfies (in mathematical notation)

$$N = \{o,\ So,\ S(So),\ \ldots\}$$

Together this ensures that N is isomorphic to $\mathbb{N}$. Once we know this, the specification of the order predicate $\leq$ does what it is supposed to do since it defines the function $\leq$ recursively by a set of terminating, exhaustive and disjoint equations that are valid in the standard model.

Let $R := \{o,\ So,\ S(So),\ \ldots\}$. From the types of o and S we know $R \subseteq N$. We prove $N \subseteq R$ by contradiction. Let $a \in N - R$. Let $f := \lambda n{\in}\mathbb{N}.\, (n{\in}R)$. Then $fo = 1$ and $\forall(\lambda x.fx \rightarrow f(Sx)) = 1$. Hence $fa = 1$ by Axiom Ind. Thus $a \in R$ by the definition of f. This is a contradiction.

It remains to show that R is infinite. Once more, we prove this claim by contradiction. Let $m, n \in \mathbb{N}$ with $n \geq 1$ and $S^{m+n}o = S^m o$, where $S^n o$ is the

result of the n-fold application of the function S to o. To have unique names, we use $\preceq$ for the order predicate $\leq$ of the specification. We obtain a contradiction as follows:

$$\begin{aligned} 1 &= (S^m o \preceq S^m o) && \text{3rd and 1st axiom for } \preceq \\ &= (S^{m+n} o \preceq S^m o) && S^{m+n} o = S^m o \\ &= (S^n o \preceq o) && \text{3rd axiom for } \preceq \\ &= 0 && \text{2nd axiom for } \preceq \text{ and } n > 0 \end{aligned}$$

3.8 Properties of Terms and Specifications

A term is called

- **closed** if it does not contain a variable.
- a **sentence** if it is a closed formula.
- **equational** if it is an equation.
- **basic** if its type is a sort.
- **functional** if its type is functional.
- **combinatorial** if none of its subterms is an abstraction.
- **normal** if none of its subterms has the form $(\lambda x.s)t$.
- **first-order** if none of its subterms is a functional variable.
- **modest** if it contains only modest names. A name is called **modest** if its type has the form $A_1 \to \cdots \to A_n \to B$ where $n \geq 0$ and $A_1, \ldots, A_n$ and B are sorts.
- **pure** if it does not depend on a fixed atom.
- **stratified** if it is pure or an equation $s = t$ where s, t are pure terms.
- **algebraic** if it is combinatorial, modest, first-order and stratified.

A specification is called **closed** [**equational**, **combinatorial**, **normal**, **first-order**, **modest**, **stratified**, **algebraic**] if all its axioms are closed [equational, combinatorial, normal, first-order, modest, stratified, algebraic]. Note that stratified and algebraic specifications are always equational.

Terms and specifications are called **higher-order** if they are not first-order.

Proposition 3.18 A modest term is combinatorial if it is basic and normal.

Proposition 3.19 If s is obtained from t by $\beta\eta$-reduction, then s contains only atoms that occur in t.

Proposition 3.20 $\beta\eta$-reduction of a term preserves closedness, modesty, first-orderness, pureness, stratification, and being licenced by a signature.

Proposition 3.21 For every modest and basic term one obtains with $\beta\eta$-reduction a unique term that is modest, basic and combinatorial.

Proof Follows with Corollary 2.15 and Propositions 3.18 and 3.20. ∎

The specifications Group and BA are typical examples of algebraic specifications. The specifications LD, LA and Nat are higher-order specifications.

Exercise 3.22 Say for each axiom of LA whether it is algebraic, stratified, modest, first-order, combinatorial or higher-order.

4 Deduction

Relations "$A \vdash s$" saying which formulas are consequences of which specifications are at the heart of logic. They are called entailment relations. A prominent example is semantic entailment. Other entailment relations can be obtained with proof systems, where $A \vdash s$ means that there is a proof of s from A. In this chapter we consider proof systems for HOL.

4.1 Entailment Relations

There exists a useful abstract notion of an entailment relation. An **entailment relation on a set** X is a set $\vdash \subseteq \mathcal{P}(X) \times X$ such that the following conditions are satisfied for all $A, B \subseteq X$ and all $x \in X$:

- $x \in A \Longrightarrow A \vdash x$ **Expansivity**
- $A \vdash x \wedge A \subseteq B \Longrightarrow B \vdash x$ **Monotonicity**
- $A \vdash B \wedge A \cup B \vdash x \Longrightarrow A \vdash x$ **Idempotence**

The formulation of the idempotence condition uses the notation

$$A \vdash B \;:\Longleftrightarrow\; \forall x \in B\colon\; A \vdash x$$

Proposition 4.1 Semantic entailment is an entailment relation on the set of all formulas.

Once we have an entailment relation, we can strengthen it by means of axioms.

Proposition 4.2 Let $\vdash$ be an entailment relation on X and $A \subseteq X$. Then

$$\overset{A}{\vdash} \;:=\; \{\, (B, x) \mid A \cup B \vdash x \,\}$$

is an entailment relation on X such that $\vdash \subseteq \overset{A}{\vdash}$.

Entailment relations obtained from proof systems have two important properties that are not satisfied by the semantic entailment relation of HOL. We define these properties for abstract entailment relations. An entailment relation $\vdash$ on X is

- **compact** if for all A, x such that $A \vdash x$ there exists a finite set $B \subseteq A$ such that $B \vdash x$.
- **effective** if X is decidable and the set $\{\, x \mid A \vdash x \,\}$ is semi-decidable for every semi-decidable set $A \subseteq X$.

Proposition 4.3 Semantic entailment is not compact.

Proof Consider the specification Nat of the natural numbers and the set Inf consisting of the equations $\neg(a = o)$, $\neg(a = So)$, $\neg(a = S(So))$, ... where a is some parameter that does not occur in Nat. Then Nat $\cup$ Inf has no model, but every finite subset of Nat $\cup$ Inf has a model. Hence Nat $\cup$ Inf $\models 0$, but $A \not\models 0$ for every finite subset $A \subseteq$ Nat $\cup$ Inf. ∎

Proposition 4.4 Semantic entailment is not effective.

Proof One can construct a finite specification T such that for every Turing machine M there is a formula s such that $A \models s$ if and only if M does not terminate on the empty tape. This is the complement of the halting problem, which is known to be not semi-decidable. ∎

Here are some notations for entailment relations ($A \vdash B$ and $A \overset{B}{\vdash} C$ were already defined).

$$\begin{aligned}
A \vdash B &:\Longleftrightarrow \forall x \in B\colon\ A \vdash x \\
A, B \vdash x &:\Longleftrightarrow A \cup B \vdash x \\
A, B \vdash C &:\Longleftrightarrow A \cup B \vdash C \\
A, x \vdash y &:\Longleftrightarrow A \cup \{x\} \vdash y \\
x_1, \ldots, x_n \vdash x &:\Longleftrightarrow \{x_1, \ldots, x_n\} \vdash x \\
A \dashv\vdash B &:\Longleftrightarrow A \vdash B \wedge B \vdash A \\
A \overset{B}{\vdash} C &:\Longleftrightarrow A, B \vdash C \\
A \overset{B}{\dashv\vdash} C &:\Longleftrightarrow A \overset{B}{\vdash} C \wedge C \overset{B}{\vdash} A
\end{aligned}$$

The notion of **entailment equivalence** that comes with the notation "$A \dashv\vdash B$" is useful. We state the properties of $\dashv\vdash$ for an arbitrary entailment relation $\vdash$. This also applies to $\overset{B}{\dashv\vdash}$ since $\overset{B}{\dashv\vdash}$ is the entailment eqivalence for the entailment relation $\overset{B}{\vdash}$.

Proposition 4.5 Let $A \dashv\vdash A'$. Then:

1. $A \vdash x \iff A' \vdash x$
2. $A, B \vdash x \iff A', B \vdash x$
3. $B \vdash A \iff B \vdash A'$
4. $\overset{A}{\vdash} = \overset{A'}{\vdash}$

We call an entailment relation $\vdash$ **sound** for an entailment relation $\vdash'$ if $\vdash \subseteq \vdash'$.

4.2 Proof Systems

Before we look at a concrete proof system for HOL, we define an abstract notion of a proof system. Let X be a set. A **rule on X** is a pair (P, x) such that P is a finite subset of X and $x \in X$. The elements of P are called the **premises of the rule**, and x is called the **conclusion of the rule**. A **proof system on X** is a set of rules on X. A **derivation of $x \in X$ from $A \subseteq X$** in a proof system S on X is a tuple $(x_1, \ldots, x_n)$ such that:

1. $x_n = x$
2. $\forall i \in \{1, \ldots, n\}\colon\ x_i \in A \ \vee\ \exists P \subseteq \{x_1, \ldots, x_{i-1}\}\colon\ (P, x_i) \in S$.

For every proof system S, we define a relation $\vdash_S$, called the **entailment relation of S**:

$$A \vdash_S x \ :\Longleftrightarrow\ \exists \text{ derivation of } x \text{ from } A \text{ in } S$$

Proposition 4.6 For every proof system S on X the relation $\vdash_S$ is a compact entailment relation on X.

As example we consider the proof system $Mul = \{\, (\{x, y\},\ x \cdot y) \mid x, y \in \mathbb{N} \,\}$ on the set $\mathbb{N}$. The rules of Mul can be described by one **schematic rule**:

$$\frac{x \quad y}{x \cdot y}\ x, y \in \mathbb{N}$$

Here is a derivation that proves $3, 7 \vdash 189$:

1)	3	assumption
2)	7	assumption
3)	9	rule with (1), (1)
4)	21	rule with (1), (2)
5)	189	rule with (3), (4)

Proposition 4.7 Let S be a proof system. Then $\vdash_S$ is semi-decidable if X is decidable and S is semi-decidable.

Let $\vdash$ be an entailment relation on X. We say that a rule (P, x) on X is

- **sound for $\vdash$** if $P \vdash x$.
- **bidirectionally sound for $\vdash$** if $P \vdash x$ and $\{x\} \vdash y$ for all $y \in P$.

We say that a schematic rule on X is **(bidirectionally) sound for $\vdash$** if each of its instances is (bidirectionally) sound for $\vdash$.

Let $\vdash$ be an entailment relation on X and S be a proof system on X. We say that

- S is **sound for** $\vdash$ if $\vdash_S \subseteq \vdash$.
- S is **complete for** $\vdash$ if $\vdash \subseteq \vdash_S$.
- a rule is **subsumed by** S if it is sound for $\vdash_S$.

Proposition 4.8 A proof system S on X is sound for an entailment relation $\vdash$ on X if and only if every rule of S is sound for $\vdash$.

Proposition 4.9 Let S be a proof system and let S' be obtained from S by adding rules that are subsumed by S. Then $\vdash_S = \vdash_{S'}$.

Let S be a proof system. The **closure of a set A with respect to S** is defined as follows:

$$S[A] = \{\, x \mid A \vdash_S x \,\}$$

For our example system *Mul* we have the following:

$$\begin{aligned} \mathit{Mul}[\emptyset] &= \emptyset \\ \mathit{Mul}[\{2\}] &= \{\, 2^n \mid n \in \mathbb{N} \wedge n \geq 1 \,\} \\ \mathit{Mul}[\{3,7\}] &= \{\, 3^m \cdot 7^n \mid m,n \in \mathbb{N} \wedge m+n \geq 1 \,\} \end{aligned}$$

Let S be a proof system. We say that a set A **is closed under** S if for every rule $(P,x) \in S$: $\quad P \subseteq A \Longrightarrow x \in A$.

Proposition 4.10 (Closure) Let S be a proof system and A, B be sets. Then:

1. $A \subseteq B \ \wedge\ B$ closed under $S \ \Longrightarrow\ S[A] \subseteq B$
2. $S[A]$ is the least set that contains A and is closed under S.
3. A closed under $S \ \Longleftrightarrow\ S[A] = A$

4.3 Replacing Equals with Equals

A basic form of mathematical reasoning is the replacement of equals with equals. For instance, if we know $a = a \cdot b$ and $b = c + a$, we may have the following chain of reasoning:

$$\begin{aligned} a &= a \cdot b && \text{since } a = a \cdot b \\ &= a \cdot (c + a) && \text{since } b = c + a \\ &= (a \cdot b) \cdot (c + a) && \text{since } a = a \cdot b \end{aligned}$$

We will call this kind of reasoning *equational deduction*. An important point about equational deduction is the fact that it is based on syntactic rules rather than semantic arguments.

Here is a schematic rule for formulas that provides a restricted form of replacement of equals with equals:

$$\rho_0 \quad \frac{s_1 = s_2 \quad [x{:=}s_1]\,t}{[x{:=}s_2]\,t} \qquad \textbf{conservative replacement}$$

This rule justifies the replacement steps of our example. Nevertheless, we need a more general replacement rule. To see this, note that

$$\frac{fx = a \quad h(\lambda x.\, fx)}{h(\lambda x.\, a)}$$

is not an instance of ρ_0 if x is a variable (since substitution does not capture). However, it is a valid replacement since semantically the variable x is universally quantified. Hence we will employ a more general replacement rule, which looks as follows:

$$\rho \quad \frac{s = t \quad Cs}{Ct} \qquad \textbf{replacement}$$

The meta-variable C represents a so-called context, a notion that we will now define. On an informal basis, we already used general replacement when we considered reduction (§ 2.4).

We first consider **untyped contexts**, which are functions $Ter \to Ter$ defined recursively as follows:

1. $\lambda s{\in}Ter.\ s$ is a context.
2. If t is a term and C is a context, then $\lambda s{\in}Ter.\ A(Cs)t$ is a context.
3. If t is a term and C is a context, then $\lambda s{\in}Ter.\ At(Cs)$ is a context.
4. If x is a variable and C is a context, then $\lambda s{\in}Ter.\ Lx(Cs)$ is a context.

Proposition 4.11 The composition $\lambda s{\in}Ter.\, C(C's)$ of two contexts C and C' is a context.

A context is **well-typed** if there exists a term s such that Cs is a well-typed term. Following the general well-typedness convention, we will only write Cs if s and Cs are well-typed terms.

Combinatorial contexts can be represented as pairs (x, t) consisting of a name x and a term t in which x occurs exactly once. The application of a context can then be simulated with substitution. For contexts involving abstractions this is not the case. Consider for instance $C = \lambda s{\in}Ter.\, Lxs$. Then $Cx = Lxx$, that is, the external variable x is captured. But substitution never captures.

Exercise 4.12 Let $C = \lambda s{\in}Ter.\, \lambda xy.s$. Give the following terms:

a) Cx

$$\rho\ \frac{s = t \quad Cs}{Ct} \qquad \beta\ \frac{}{(\lambda x.s)t = [x{:=}t]s} \qquad \eta\ \frac{}{\lambda x.sx = s}\ x \notin \mathcal{N}s$$

Figure 8: Basic deduction rules

b) Cz where $x, y \neq z$

c) $[z{:=}y](\lambda xy.z)$

Proposition 4.13 Let $s = t$ be an equation, C be a context such that Cs is a formula, and $\mathcal{A}$ be a structure. Then $\mathcal{A} \models s = t \wedge \mathcal{A} \models Cs \implies \mathcal{A} \models Ct$.

Proof By induction on the size of the context C. ∎

Proposition 4.14 (Soundness) ρ is sound for $\models$.

Proof Follows from Proposition 4.13. ∎

Exercise 4.15 Find a example that shows that Proposition 4.13 does not hold if clause (4) in the definition of contexts would allow x to be a parameter.

4.4 Basic Proof System and Deductive Entailment

A **deduction rule** is a schematic rule on the set of formulas. Figure 8 shows three deduction rules ρ, β and η that we will refer to as **the basic deduction rules**. They yield a proof system that is sound for semantic entailment. We refer to this proof system as **the basic proof system**. We write $\vdash$ for the entailment relation of the basic proof system and refer to it as **deductive entailment**. We say that a formula *s* **is deducible from a specification** *A* if $A \vdash s$.

Proposition 4.16 (Soundness) Deductive entailment is sound for semantic entailment.

Proof It suffices to show that every instance of the rules ρ, β and η is sound for semantic entailment (Proposition 4.8). This follows for the instances of ρ with Proposition 4.14, for the instances of β with Proposition 3.8, and for the instances of η with Exercise 3.9. ∎

4.5 Subsumed Deduction Rules

The basic proof system defines the deductive entailment relation for HOL. With the right axioms it allows us to prove whatever we want. However, derivations

$$\textbf{Ref}\ \frac{}{s = s} \qquad \textbf{Sym}\ \frac{s = t}{t = s} \qquad \textbf{Trans}\ \frac{s = s' \quad s' = t}{s = t}$$

$$\textbf{CL}\ \frac{s = s'}{st = s't} \qquad \textbf{CR}\ \frac{t = t'}{st = st'} \qquad \xi\ \frac{s = s'}{\lambda x.s = \lambda x.s'}\ x \in Var$$

$$\textbf{Eta}\ \frac{sx = tx}{s = t}\ x \in Var - \mathcal{N}(s{=}t) \qquad \textbf{Subst}\ \frac{s = t}{\theta s = \theta t}\ Ker\theta \subseteq Var$$

Figure 9: Subsumed deduction rules

in the basic proof system are very low level (like a machine language compared to a programming language). To arrive at a more comfortable proof system, we extend the basic proof system with the deduction rules in Figure 9. Since the additional rules are subsumed by the basic proof sytem, what we can prove does not change.

Proposition 4.17 (Subsumption) The deduction rules in Figure 9 are sound for deductive entailment.

First we show that the instances of Ref are subsumed by the basic proof system. Let s be a term. Here is a derivation of $s = s$ from $\emptyset$.

1)	$(\lambda x.s)x = s$	β
2)	$s = s$	ρ with (1), (1)

The second step is best explained by showing the instance of ρ that is used:

$$\frac{(\lambda x.s)x = s \qquad (\lambda x.s)x = s}{s = s}$$

The context employed is $\lambda t{\in}Ter.\ (t = s)$.

Here is a derivation showing that Sym is subsumed:

1)	$s = t$	assumption
2)	$s = s$	Ref
3)	$t = s$	ρ with (1), (2)

Here is a derivation showing that Trans is subsumed:

1)	$s = s'$	assumption
2)	$s' = t$	assumption

3)	$s = t$	ρ with (2), (1)

CL, CR, and ξ can be derived with Ref and ρ. Here is a derivation showing that ξ is subsumed:

1)	$s = s'$	assumption
2)	$\lambda x.s = \lambda x.s$	Ref
3)	$\lambda x.s = \lambda x.s'$	ρ with (1), (2)

Note that line (3) is only valid if x is a variable, a requirement imposed by ρ and the definition of contexts.

Exercise 4.18 Find a derivation that shows that the instances of Eta are subsumed by the basic proof system. You may use Ref, Sym and Trans.

Exercise 4.19 Show that the instances of η are subsumed by the proof sytem obtained with β and Eta.

Note that the rules β, η, Ref, and Sym are bidirectionally sound. The following propositions state that is is also true for ξ and Eta.

Proposition 4.20 (Eta) $sx = tx \mapsto s = t$ if $x \in Var - \mathcal{N}(s{=}t)$

Proposition 4.21 (Xi) $s = t \mapsto \lambda x.s = \lambda x.t$ if $x \in Var$

Proposition 4.22 (Weakening) Let s, t be formulas. Then $\overset{A}{\vdash} s{=}t \implies s \overset{A}{\mapsto} t$.

The converse of Weakening, $s \mapsto t \implies \vdash s{=}t$, does not hold. To see this, let $I = \lambda x.x$ where x is a variable of type **B**. Then $\nvdash (I = \lambda x.(I{=}I)) = (x = (I{=}I))$ by soundness. However, $I = \lambda x.(I{=}I) \mapsto x = (I{=}I)$ with Eta and β.

Exercise 4.23 Prove $s = (I{=}I) \vdash s$. We conjecture that $s \vdash s = (I{=}I)$ does not hold but don't have a proof.

Next we show that the special case of Subst where $\theta = [x{:=}u]$ is subsumed.

1)	$s = t$	assumption
2)	$(\lambda x.s)u = [x{:=}u]s$	β
3)	$(\lambda x.t)u = [x{:=}u]t$	β
4)	$(\lambda x.t)u = [x{:=}u]s$	ρ with (1), (2)
5)	$[x{:=}u]s = [x{:=}u]t$	ρ with (4), (3)

To show that every instance of Subst is subsumed, more effort is required. The claim follows with a coincidence argument (Proposition 2.2) from the following proposition, which generalizes the β-rule to n variables. We use the following notation:

$$[x_1{:=}s_1,\ldots,x_n{:=}s_n] \;:=\; \{(x_1,s_1),\ldots,(x_n,s_n)\} \cup \{\,(x,x) \mid x \in Nam - \{x_1,\ldots,x_n\}\,\}$$

Proposition 4.24 (Beta) $\emptyset \vdash (\lambda x_1 \ldots x_n.\, t)s_1 \ldots s_n = [x_1{:=}s_1,\ldots,x_n{:=}s_n]t$

Proof By induction on n. Case analysis for $n = 0$ and $n > 0$. Use induction for $(\lambda x_1 \ldots x_{n-1}.(\lambda x_n.t))s_1 \ldots s_{n-1}$. ∎

4.6 Conversion

An equation e is

- an **instance** of an equation $s = t$ if $e = (\theta t = \theta s)$ for some substitution θ with $Ker\theta \subseteq Var$. Here are two instances of $x = a$: $a = a$, $fx = a$.
- an **extension** of an equation $s = t$ if $e = (Cs = Ct)$ for some context C. Here are some extensions of $x = a$: $fx = fa$, $\lambda x.fx = \lambda x.fa$, $(\lambda x.fx)a = (\lambda x.fa)a$.
- a **β-reduction step** if it is an extension of an equation that can immediately be obtained with the β-rule.
- an **η-reduction step** if it is an extension of an equation that can immediately be obtained with the η-rule.
- an **A-reduction step** if it is an extension of an instance of an equation in A.

Proposition 4.25 (Conversion Rules) The following deduction rules are sound for deductive entailment.

$$\beta' \quad \frac{s}{t} \quad s = t \text{ or } t = s \text{ is a } \beta\text{-reduction}$$

$$\eta' \quad \frac{s}{t} \quad s = t \text{ or } t = s \text{ is an } \eta\text{-reduction}$$

$$\rho' \quad \frac{e \quad s}{t} \quad s = t \text{ or } t = s \text{ is a } \{e\}\text{-reduction}$$

The **converse of an equation** $s = t$ is the equation $t = s$. An equation is a **conversion step from** A if the equation or its converse is a β-, η or A-reduction step. A **conversion proof** of an equation e from A is a tuple $(s_1,\ldots,s_n)$ such that

1. $n \geq 1$ and $e = (s_1 = s_n)$.
2. $\forall i \in \{1, \ldots, n-1\}$: $s_i = s_{i+1}$ is a conversion step from A.

A conversion proof $(s_1, \ldots, s_n)$ is **combinatorial** if $s_1, \ldots, s_n$ are combinatorial. We say that two formulas s, t are **convertible in** A if there is a conversion proof of $s = t$ from A.

Example 4.26 Let A be the specification

$$I : V \to V$$
$$K : V \to V \to V$$
$$S : (V \to V \to V) \to (V \to V) \to V \to V$$
$$Ix = x \qquad I$$
$$Kxy = x \qquad K$$
$$Sfgx = fx(gx) \qquad S$$

where I, K, S are constants and x, y, f, g are variables. Here is a conversion proof of $SKI = I$ from A:

$$\begin{aligned} SKI &= \lambda x.\, SKIx && \eta \\ &= \lambda x.\, Kx(Ix) && S \\ &= \lambda x.\, x && K \\ &= \lambda x.\, Ix && I \\ &= I && \eta \end{aligned}$$

It seems that there is no combinatorial conversion proof of $SKI = I$ from A. Since A is combinatorial, this demonstrates that abstractions are essential for deductive entailment. □

Proposition 4.27 (Lifting) If $s = t$ is a conversion step from A, then $Cs = Ct$ is a conversion step from A. Moreover, if there is a conversion proof of $s = t$ from A, then there is a conversion proof of $Cs = Ct$ from A.

Proof Follows from the fact that an extension of an extension of an equation e is an extension of e. ■

A specification is called **equational** if all its axioms are equations. Note that stratified and algebraic specifications are equational. We define **conversion entailment** as follows:

$$A \vdash_{\text{con}} s = t \;:\Longleftrightarrow\; A \text{ equational and } s, t \text{ are convertible in } A$$

Proposition 4.28 Conversion entailment is a compact entailment relation that is sound for deductive entailment: $\vdash_{\text{con}} \subseteq \vdash$.

5 Propositional Logic

Logics restricted to the sort **B** and the Boolean connectives are usually called **propositional logics**. Traditional accounts of propositional logic restrict themselves to formulas that are combinatorial and first-order. We will not impose such restrictions. What we will consider in this chapter we call **higher-order propositional logic** to distinguish it from the usually considered subsystem that we call **first-order propositional logic**. You'll find a nice presentation of the history of first-order propositional logic at www.iep.utm.edu/p/prop-log.htm#H2.

Semantically equivalent specifications (i.e., $A \models\!\!\models B$) are usually not deductively equivalent (i.e., $A \not\vdash\!\!\vdash B$). As example, take the empty specification $\emptyset$ and Sym = $\{(x{=}y) = (y{=}x)\}$ where x, y are variables of type **B**. Semantically, $\emptyset$ and Sym are equivalent. Deductively, they don't seem to be equivalent since there doesn't seem to be a proof of $(x{=}y) = (y{=}x)$ from $\emptyset$.

In this chapter we will study the deductive aspects of a higher-order specification PL that specifies the Boolean connectives such that a strong deductive entailment relation is obtained. A semantically equivalent first-order specification looks as follows:

$$\begin{array}{ll} (1 = 1) = 1 & \neg x = x \to 0 \\ 0 \to x = 1 & x \vee y = (x \to y) \to y \\ 1 \to x = x & x \wedge y = \neg(\neg x \vee \neg y) \end{array}$$

Deductively, the first-order specification is much weaker than the higher-order specification PL. For instance, $(0 = 1) = 0$ does not seem to be deducible in the first-order specification.

5.1 Specification PL

Figure 10 shows the specification PL we will explore in this chapter. It distinguishes between the **primary constants** 0, 1, $\to$ and the **defined constants** $\neg$, $\vee$, $\wedge$, which can be expressed with the primary constants. In addition, the specification use the logical constant $\doteq_{\mathbf{B}}$ (Boolean identity). Except for BCA, all axioms are first-order. The axiom E0 is semantically redundant.

Proposition 5.1 PL has exactly one model. This model gives PL's constants their canonical meaning.

Proof Follows with the arguments used in the discussion of LA in § 3.6. ∎

The next two propositions are straightforward consequences of E1.

Proposition 5.2 (One) $\mathrm{PL} \vdash 1$

Specification	PL	
Primary Constants	$0, 1 : \mathbf{B}$	
	$\to : \mathbf{B} \to \mathbf{B} \to \mathbf{B}$	
Axioms	$(x = 1) = x$	E1
	$(1 = 0) = 0$	E0
	$1 \to x = x$	I1
	$f0 \to f1 \to fx$	BCA (Boolean case analysis)
Defined Constants	$\neg x = x \to 0$	
	$x \vee y = (x \to y) \to y$	
	$x \wedge y = \neg(\neg x \vee \neg y)$	
Notation	$\overline{x} := \neg x$	

Figure 10: Specification of propositional logic

Proposition 5.3 (Up and Down) $s{=}1 \overset{\text{PL}}{\dashv\vdash} s$

The direction $s{=}1 \overset{\text{PL}}{\vdash} s$ is called **Up**, the other direction is called **Down**.

Proposition 5.4 (Subst') $s \overset{\text{PL}}{\vdash} \theta s$ if $Ker\theta \subseteq Var$

Proof Follows with E1 and Subst. ∎

Proposition 5.5 (MP: Modus Ponens) $s \to t,\ s \overset{\text{PL}}{\vdash} t$

Proof Follows with Down and I1. ∎

Proposition 5.6 (I0) $\text{PL} \vdash 0 \to x = 1$

Proof The claim follows with the following conversion proof, which uses the equational version of the axiom BCA obtainable with Down.

$$\begin{aligned} 1 &= (\lambda x.x)0 \to (\lambda x.x)1 \to (\lambda x.x)x && \text{BCA, Down} \\ &= 0 \to 1 \to x && \beta \\ &= 0 \to x && \text{I1} \end{aligned}$$

∎

Proposition 5.7 (Exx) $\text{PL} \vdash (x{=}x) = 1$

Proof Follows with Ref and Down. ∎

Proposition 5.8 (BCA) PL $\vdash [x{:=}0]s \to [x{:=}1]s \to s$.

Proof Follows with BCA, Subst' ($f := \lambda x.s$) and β. ∎

Proposition 5.9 (BCA) $[x{:=}0]s,\ [x{:=}1]s \overset{\text{PL}}{\vdash} s$

Proof The direction from left to right follows with MP from the preceding proposition. The other direction follows with Subst'. ∎

A **zero-one instance** of a formula s is a formula θs such that:

1. $\theta x \neq x \implies x : \mathbf{B} \wedge \theta x \in \{0, 1\}$
2. θs contains no variables of type **B**.

Proposition 5.10 (Enumeration) A formula is deducible from PL if and only if all its zero-one instances are deducible from PL.

Proof One direction follows by Subst. To show the other direction, let s be a formula such that every zero-one instance of s is deducible from PL. We show that s is deducible from PL by induction on the number of variables of type **B** occurring in s.

If s contains no such variable, s is a zero-one instance of s and the claim follows by assumption.

If s contains a variable $x : \mathbf{B}$, all zero-one instances of $[x := 0]s$ and $[x := 1]s$ are zero-one instances of s. Hence we know by induction that $[x := 0]s$ and $[x := 1]s$ are deducible from PL. Thus we know by BCA that s is deducible from PL. ∎

Proposition 5.11 (Equiv) PL $\vdash (x{=}y) = (x \to y) \wedge (y \to x)$.

Proof By Enumeration it suffices to show that the zero-one instances of the equation are deducible in PL. Here they are:

$$(0{=}0) = (0 \to 0) \wedge (0 \to 0) \qquad (1{=}1) = (1 \to 1) \wedge (1 \to 1)$$
$$(0{=}1) = (0 \to 1) \wedge (1 \to 0) \qquad (1{=}0) = (1 \to 0) \wedge (0 \to 1)$$

We give a conversion proof of the last equation:

$$\begin{aligned}
(1 \to 0) \wedge (0 \to 1) &= 0 \wedge 1 && \text{I1, I0}\\
&= \neg(\neg 0 \vee \neg 1) && \text{Definition } \wedge\\
&= \neg((\neg 0 \to \neg 1) \to \neg 1) && \text{Definition } \vee\\
&= (((0 \to 0) \to (1 \to 0)) \to (1 \to 0)) \to 0 && \text{Definition } \neg\\
&= ((1 \to 0) \to 0) \to 0 && \text{I0, I1, I1}
\end{aligned}$$

Up $\dfrac{s = 1}{s}$ **Down** $\dfrac{s}{s = 1}$ **Subst'** $\dfrac{s}{\theta s}$ $Ker\theta \subseteq Var$

MP $\dfrac{s \to t \quad s}{t}$ **BCA** $\dfrac{[x{:=}0]s \quad [x{:=}1]s}{s}$

Equiv $\dfrac{s \to t \quad t \to s}{s = t}$

Figure 11: Sound deduction rules for $\overset{PL}{\vdash}$

$$
\begin{aligned}
&= (0 \to 0) \to 0 && \text{I1}\\
&= 1 \to 0 && \text{I0}\\
&= 0 && \text{I1}\\
&= (1{=}0) && \text{E0}
\end{aligned}
$$

The deducibility of the other equations follows with similar conversion proofs. ∎

Figure 11 shows some deduction rules whose soundness for $\overset{PL}{\vdash}$ follows with the preceding propositions. Note that the rules Up, Down, BCA and Equiv are bidirectionally sound.

5.2 Tautological Completeness

A formula is called

- **propositional** if it is combinatorial and contains no other names but 0, 1, $\to$, $\doteq_{\mathbf{B}}$, $\neg$, $\vee$, $\wedge$, and variables of type **B**.
- a **tautology** if it is propositional and semantically entailed by PL.
- **tautologous** if it is an instance of a tautology.

We use **TL** to denote the set of all tautologies. We will show: $\mathrm{PL} \dashv\vdash \mathrm{TL} \cup \{\mathrm{BCA}\}$. Figures 12, 13 and 14 show the most important tautologies.

Proposition 5.12 (Base) For every propositional formula s there exists a propositional formula t such that $\mathrm{PL} \vdash s = t$ and t contains no other names but 0 and $\to$ and variables that occur in s.

Proof Follows with a conversion proof from: Equiv; the defining equations for $\wedge$, $\vee$, $\neg$; and $1 = (0 \to 0)$ (the converse of an instance of I0). ∎

$0 \to x = 1 \qquad x \to 0 = \overline{x} \qquad x \to x = 1$
$1 \to x = x \qquad x \to 1 = 1$

$$x \to y \to z = y \to x \to z \qquad \text{Commutativity}$$
$$x \to x \to y = x \to y$$
$$x \to y \to z = x \wedge y \to z \qquad \text{Schönfinkel}$$
$$x \wedge (x \to y) = x \wedge y \qquad \text{Modus ponens}$$
$$x \vee y \to z = (x \to z) \wedge (y \to z) \qquad \text{Distributivity}$$
$$x \to y \wedge z = (x \to y) \wedge (x \to z) \qquad \text{Distributivity}$$
$$x \wedge \overline{y} \to z = x \to y \vee z \qquad \text{Trading}$$
$$x \to y \vee \overline{z} = x \wedge z \to y \qquad \text{Trading}$$
$$x \to y = \overline{y} \to \overline{x} \qquad \text{Contraposition}$$
$$x \to y = \overline{x} \vee y$$
$$x \to y = (x = x \wedge y) \qquad \text{Golden Rule}$$
$$x \to y = (y = x \vee y) \qquad \text{Golden Rule}$$
$$x \vee y = (x \to y) \to y$$

$$x \wedge y \to x \qquad \text{Triviality}$$
$$x \to x \vee y \qquad \text{Triviality}$$
$$x \wedge y \to x \vee z \qquad \text{Triviality}$$
$$(x \to y) \to x \wedge z \to y \wedge z \qquad \text{Monotonicity}$$
$$(x \to y) \to x \vee z \to y \vee z \qquad \text{Monotonicity}$$
$$(x \to y) \to (x \wedge x' \to y) \qquad \text{Weakening}$$
$$(x \to y) \to (x \to y \vee y') \qquad \text{Weakening}$$
$$x \vee y \to (z = (x \to z) \wedge (y \to z)) \qquad \text{Case Analysis(CA)}$$
$$z = (x \to z) \wedge (\overline{x} \to z)$$
$$(\overline{x} \to y) \wedge (x \to z) = \overline{x} \wedge y \vee x \wedge z \qquad \text{Conditional}$$
$$(\overline{x} \to y) \wedge (x \to z) = (\overline{x} \vee z) \wedge (x \vee y)$$
$$((x \to y) \to x) \to x \qquad \text{Peirce}$$

Figure 12: Tautologies for →

$$(x = 0) = \overline{x} \qquad (x = 1) = x$$
$$(x = x) = 1 \qquad (x = \overline{x}) = 0$$

$$(x = y) = (y = x) \quad \text{Symmetry}$$
$$(x = (y = z)) = ((x = y) = z) \quad \text{Associativity}$$
$$\overline{x = y} = (x = \overline{y}) \quad \text{De Morgan}$$
$$\overline{\overline{x}} = x \quad \text{Double Negation}$$
$$(x = y) = (\overline{x} = \overline{y}) \quad \text{Contraposition}$$
$$(x = y) = (x \to y) \wedge (y \to x) \quad \text{Equiv}$$
$$(x = y) = (x \wedge y) \vee (\overline{x} \wedge \overline{y})$$
$$(x = y) = (\overline{x} \vee y) \wedge (x \vee \overline{y})$$
$$(x = \overline{y}) = (x \vee y) \wedge \overline{x \wedge y} \quad \text{Uniqueness of Complements}$$

Figure 13: Tautologies for =

$0 \wedge x = 0$	$1 \vee x = 1$	Dominance
$1 \wedge x = x$	$0 \vee x = x$	Identity
$x \wedge x = x$	$x \vee x = x$	Idempotence
$x \wedge \overline{x} = 0$	$x \vee \overline{x} = 1$	Complement
$x \wedge y = y \wedge x$	$x \vee y = y \vee x$	Commutativity
$x \wedge (y \wedge z) = (x \wedge y) \wedge z$	$x \vee (y \vee z) = (x \vee y) \vee z$	Associativity
$x \wedge (x \vee y) = x$	$x \vee (x \wedge y) = x$	Absorption
$\overline{x \wedge y} = \overline{x} \vee \overline{y}$	$\overline{x \vee y} = \overline{x} \wedge \overline{y}$	DeMorgan

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z) \quad \text{Distributivity}$$
$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z) \quad \text{Distributivity}$$
$$(x \wedge y) \vee (\overline{x} \wedge z) = (x \wedge y) \vee (\overline{x} \wedge z) \vee (y \wedge z) \quad \text{Resolution}$$
$$(x \vee y) \wedge (\overline{x} \vee z) = (x \vee y) \wedge (\overline{x} \vee z) \wedge (y \vee z) \quad \text{Resolution}$$

Figure 14: Tautologies for $\wedge$ and $\vee$

Proposition 5.13 For every propositional formula s containing no other names but 0, 1, and $\to$: $\mathrm{PL} \vdash s{=}0$ or $\mathrm{PL} \vdash s{=}1$.

Proof By induction on $|s|$. If $s = 0$ or $s = 1$, the claim follows by Ref. If $s = s_1 \to s_2$, we use induction for s_1. If $\mathrm{PL} \vdash s_1{=}0$, then $\mathrm{PL} \vdash s{=}1$ with I0. If $\mathrm{PL} \vdash s_1{=}1$, then $\mathrm{PL} \vdash s{=}s_2$ with I1. Hence the claim follows with induction for s_2. ∎

Proposition 5.14 (0-1)
For every closed propositional formula s: $\mathrm{PL} \vdash s{=}0$ or $\mathrm{PL} \vdash s{=}1$.

Proof Follows with Propositions 5.12 and 5.13. ∎

Proposition 5.15 (Tautological Completeness) Every tautologous formula is deducible from PL.

Proof By Subst' it suffices to show that tautologies are deducible in PL. By Enumeration and Soundness it suffices to show that closed tautologies are deducible in PL. This holds by Proposition 5.14 and Soundness. ∎

Note that the proofs of the preceding propositions suggest a straightforward procedure that decides whether a propositional formula is a tautology. The run time of this procedure is exponential in the number of variables the formula contains. This is no surprise since deciding whether a propositional formula is not a tautology is known to be NP-complete.

Provided one knows enough tautologies, tautological completeness makes it much easier to establish deductive consequences of PL. The following propositions say why this is the case.

Proposition 5.16 (Taut) $\mathrm{PL} \dashv\vdash \mathrm{TL} \cup \{\mathrm{BCA}\}$

Proposition 5.17 The following rule is sound for $\overset{\mathrm{PL}}{\vdash}$:

Taut $\dfrac{}{s}$ s tautologous

Proposition 5.18 $\mathrm{PL} \vdash s$ if there exists a derivation of s from $\{\mathrm{BCA}\}$ using Taut and the deduction rules from Proposition 4.25 and Figures 8, 9 and 11.

Proposition 5.19 $\mathrm{PL} \vdash s$ if there exists a conversion proof of s from equational tautologies.

Exercise 5.20 Given what we already know, the proofs of the propositions 5.16–5.19 are straightforward. Make sure that you can do them and that you understand every detail.

Proposition 5.21 (And) $s \wedge t \overset{\text{PL}}{\vdash\!\dashv} s, t$

Proof The direction from left to right follows with Taut and MP using the tautologies $x \wedge y \to x$ and $x \wedge y \to y$. The direction from right to left follows with Down and the tautology $1 \wedge 1$. ∎

To sketch an algorithm that decides whether a formula is tautologous, we need two definitions. A term s is a **generalization** of a term t if t is an instance of s. A term s is **more specific** than a term t if s is an instance of t. To decide whether a formula is tautologous, we compute a most specific propositional generalization of the formula and check whether it is a tautology. The following proposition ensures the correctness of this simple algorithm.

Proposition 5.22 For every formula s the following statements are equivalent:

1. s is tautologous.
2. s has a propositional generalization that is a tautology.
3. s has a most specific propositional generalization that is a tautology.
4. Every most specific propositional generalization of s is a tautology.

5.3 BCA Equivalents

We know PL $\vdash\!\dashv$ TL $\cup$ {BCA}. This means that BCA takes a prominent position in PL. There are several interesting higher-order formulas that are deductively equivalent to BCA, provided the tautologies are available.

Proposition 5.23 Let $f : \mathbf{B} \to \mathbf{B}$ and $x, y : \mathbf{B}$ be variables. Then the following formulas are deducible from PL and are equivalent with respect to $\overset{\text{TL}}{\vdash}$:

1. $f0 \to f1 \to fx$ **Boolean Case Analysis (BCA)**
2. $(x = y) \to fx \;=\; (x = y) \to fy$ **Boolean Replacement (BRep)**
3. $fx = \overline{x} \wedge f0 \vee x \wedge f1$ **Boolean Expansion (BExp)**

Proof Since PL $\vdash\!\dashv$ TL $\cup$ {BCA}, it suffices to show that the formulas are equivalent with respect to $\overset{\text{TL}}{\vdash}$.

(1) $\Longrightarrow$ (2). Since we have TL and BCA, it suffices to prove (2) from PL. Since the zero-one instances of (2) are tautologous, (2) follows with Enumeration and Taut.

(2) $\Longrightarrow$ (3) follows with a conversion proof:

$$\begin{aligned} fx &= (x = 0) \to fx \wedge (x = 1) \to fx && \text{TL} \\ &= (x = 0) \to f0 \wedge (x = 1) \to f1 && (2)\text{ twice} \\ &= \overline{x} \wedge f0 \vee x \wedge f1 && \text{TL} \end{aligned}$$

(3) $\Longrightarrow$ (1) follows with a conversion proof and Up:

$$\begin{aligned} f0 \to f1 \to fx &= f0 \to f1 \to \overline{x} \wedge f0 \vee x \wedge f1 && (3)\\ &= 1 && \text{TL} \end{aligned}$$

■

Proposition 5.24 (Boolean Replacement) Let $f : \mathbf{B} \to \mathbf{B}$ be a variable. Then the following formulas are deducible from PL and are equivalent with respect to $\overset{\text{TL}}{\vdash}$:

1. $(x = y) \to fx \to fy$
2. $(x = y) \to (fx = fy)$
3. $(x = y) \to fx \;=\; (x = y) \to fy$
4. $(x = y) \wedge fx \;=\; (x = y) \wedge fy$

Proof Since we know by Proposition 5.23 that (3) is deducible from PL, it suffices to show that the formulas are equivalent with respect to $\overset{\text{TL}}{\vdash}$. Since the formulas (2) $\to$ (3), (3) = (4) and (3) $\to$ (1) are tautologous, it suffices to show that TL, (1) $\vdash$ (2).

$$\begin{aligned} &(x = y) \to (fx = fy)\\ &= (x = y) \to (fx = fx) \to (fx = fy) && \text{TL}\\ &= (x = y) \to (\lambda y.fx = fy)x \to (\lambda y.fx = fy)y && \beta\\ &= 1 && (1) \end{aligned}$$

■

5.4 Hypothetical Conversion Proofs

Boolean replacement provides for conservative replacement within formulas:

$$\text{PL} \vdash \; (s_1 = s_2) \to [x{:=}s_1]\,t \;=\; (s_1 = s_2) \to [x{:=}s_2]\,t$$

When we prove that a formal proof for a formula exists, it it is often convenient to perform internal replacements as external conversions. To this end, we define hypothetical conversion proofs, which may involve general replacement steps with respect to a set of *axioms* and conservative replacement steps with respect to a set of *hypotheses*.

A context C is **conservative** for an equation $s_1 = s_2$ if there exist a variable x and a term t such that $Cs_1 = [x{:=}s_1]\,t$ and $Cs_2 = [x{:=}s_2]\,t$. An equation e is a **conservative A-reduction step** if there exist an equation $(s{=}t) \in A$ and a context C that is conservative for $s = t$ such that $e = (Cs{=}Ct)$. An equation is a **conversion step from A with H** if the equation or its converse is one of the

following: a β-reduction step, an η-reduction step, an A-reduction step, or a conservative H-reduction step.

Let A and H be sets of equations and e an equation. A **conversion proof of e from axioms A with hypotheses H** is a tuple $(s_1, \ldots, s_n)$ such that

1. $n \geq 1$ and $e = (s_1 = s_n)$.
2. $\forall i \in \{1, \ldots, n-1\}$: $s_i = s_{i+1}$ is a conversion step from A with H.

Proposition 5.25 (Lifting) Suppose there exists a conversion proof of $s = t$ from A with H. Then there exists a conversion proof of $Cs = Ct$ from A with H if C is a context that is conservative for every equation in H.

A **Boolean equation** is an equation $s = t$ where s and t are formulas.

Lemma 5.26 Suppose A contains BRep and there is a conversion proof of $t = t'$ from A with $H \cup \{s\}$ where s is a Boolean equation. Then there is a conversion proof of $s \to t = s \to t'$ from A with H.

Proof By induction on the length n of the conversion proof of $t = t'$. If $n = 1$, then $t = t'$ and hence there is a conversion proof of $s \to t = s \to t'$ of length 1.

Let $n \geq 2$. Then there is a term t'' such that $t = t''$ is a conversion step from A with $H \cup \{s\}$ and there is conversion proof of $t'' = t'$ from A with $H \cup \{s\}$ of length $n-1$. By induction we know that there is a conversion proof of $s \to t'' = s \to t'$ from A with H. Hence it suffices to show that there exists a conversion proof of $s \to t = s \to t''$ from A with H.

If $t = t''$ is a conversion step from A with H, then the claim follows by Lifting (there is no problem with conservative H-steps since the required lifting does not capture). Otherwise either $t = t''$ or $t'' = t$ is a conservative $\{s\}$-reduction.

Let $t = t''$ be a a conservative $\{s\}$-reduction. Let $s = (s_1 = s_2)$, $t = [x{:=}s_1]u$, and $t'' = [x{:=}s_1]u$. Here is a conversion proof of $s \to t = s \to t''$ from {BRep}:

$$
\begin{aligned}
s \to t &= (s_1 = s_2) \to [x{:=}s_1]u && \text{Notation} \\
&= (s_1 = s_2) \to (\lambda x.u)s_1 && \beta \\
&= (s_1 = s_2) \to (\lambda x.u)s_2 && \text{BRep} \\
&= (s_1 = s_2) \to [x{:=}s_2]u && \beta \\
&= s \to t'' && \text{Notation}
\end{aligned}
$$

Let $t'' = t$ be a a conservative $\{s\}$-reduction. Then the claim follows analogously to the previous case since from a conversion proof of $s \to t'' = s \to t$ from {BRep} we can obtain a conversion proof of $s \to t = s \to t''$ from {BRep}. ∎

Proposition 5.27 (Boolean Deductivity) $A \vdash s_1 \to \cdots \to s_n \to s$ holds if the following conditions are satisfied:

1. $A \vdash$ PL and $s_1, \ldots, s_n$ are Boolean equations.
2. there is a conversion proof of s from A with $\{s_1, \ldots, s_n\}$.

The proof of the next proposition demonstrates the use of Boolean Deductivity.

Proposition 5.28 (BExt: Boolean Extensionality) Let $f, g : \mathbf{B} \to \mathbf{B}$. Then:
$\mathrm{PL} \vdash (f0 = g0) \wedge (f1 = g1) \to (f = g)$

Proof Here is a conversion proof of $f = g$ from Bexp with $f0 = g0$ and $f1 = g1$:

$$\begin{aligned}
f &= \lambda x.\, fx && \eta \\
&= \lambda x.\, \overline{x} \wedge f0 \vee x \wedge f1 && \text{BExp} \\
&= \lambda x.\, \overline{x} \wedge g0 \vee x \wedge g1 && \text{Hyp } f0 = g0,\ f1 = g1 \\
&= \lambda x.\, gx && \text{BExp} \\
&= g && \eta
\end{aligned}$$

Since BExp is deducible from PL, we know by Boolean Deductivity that $(f0 = g0) \to (f1 = g1) \to (f = g)$ is deducible from PL. Now we obtain the claim with the Schönfinkel tautology. ∎

Semantically, it is clear that Deductivity should hold for all equational hypotheses, not just Boolean equations. However, so far we lack replacement axioms for non-Boolean identities. Since we don't have adequate tools for showing that semantically entailed formulas are not deducible from PL, we will be content with the formulation of a conjecture:

Conjecture 5.29 Let $f, g : \mathbf{B} \to \mathbf{B}$ be distinct variables. Then the formula $(f = g) \to (f0 = g0)$ is not deducible from PL.

The next proposition says that there are exactly 4 functions $\mathbb{B} \to \mathbb{B}$: identity, the two constant functions, and negation. The proof of this claim demonstrates the use of BExt and the monotonicity tautology for disjunction.

Proposition 5.30 (Case Analysis for B → B)
$\mathrm{PL} \vdash (f = \lambda x.x) \vee (f = \lambda x.0) \vee (f = \lambda x.1) \vee (f = \neg)$

Proof By BExt and TL (in particular the Monotonicity tautology for $\vee$) we know that it suffices to prove that

$$
\begin{aligned}
&(f0 = (\lambda x.x)0) \wedge (f1 = (\lambda x.x)1)\\
\vee\ &(f0 = (\lambda x.0)0) \wedge (f1 = (\lambda x.0)1)\\
\vee\ &(f0 = (\lambda x.1)0) \wedge (f1 = (\lambda x.1)1)\\
\vee\ &(f0 = \neg 0) \wedge (f1 = \neg 1)
\end{aligned}
$$

is deducible from PL. By β and TL we know that this formula is deductively equivalent to

$$
\begin{aligned}
&(f0 = 0) \wedge (f1 = 1)\\
\vee\ &(f0 = 0) \wedge (f1 = 0)\\
\vee\ &(f0 = 1) \wedge (f1 = 1)\\
\vee\ &(f0 = 1) \wedge (f1 = 0)
\end{aligned}
$$

Since this formula is tautologous, we are done. ∎

5.5 Case Analysis

We want to prove PL $\vdash f(f(fx)) = fx$. Semantically, PL entails $f(f(fx)) = fx$ since each of the 4 functions $\mathbb{B} \to \mathbb{B}$ satisfies the equation. Deductively, we could prove the claim from Proposition 5.30 if we had a replacement axiom $(f = g) \to hf = hg$. Since we don't have such an axiom, we will construct a more straigtforward proof based on case analysis.

By BCA we know that $f(f(fx)) = fx$ is deducible from PL if and only if its zero-one instances $f(f(f1)) = f1$ and $f(f(f0)) = f0$ are deducible from PL. We prove the first claim and leave the second as an exercise. We base the proof on a case analysis provided by the tautology CA: $x \vee y \to (z = (x \to z) \wedge (y \to z))$.

Proposition 5.31 PL $\vdash f(f(f1)) = f1$

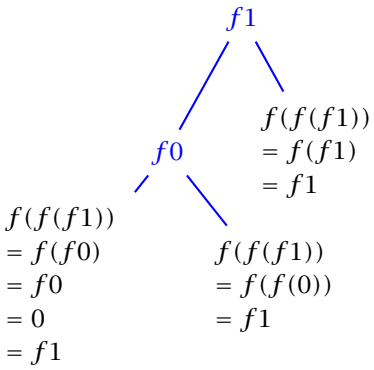
Proof By case analysis with CA.

Case $f1 = 0$, $f0 = 0$. Then $f(f(f1)) = f(f0)) = f0 = 0 = f1$ is a conversion proof from $\emptyset$ with $f1 = 0$, $f0 = 0$. Hence the claim follows with Boolean deductivity and Up.

Case $f1 = 0$, $f0 = 1$. Then $f(f(f1)) = f(f0)) = f1$.

Case $f1 = 1$. Then $f(f(f1)) = f(f1)) = f1$. ∎

The essential steps of the proof can be shown with a tree diagram:


$f1$
$f0$
$f(f(f1))$
$= f(f0)$
$= f0$
$= 0$
$= f1$
$f(f(f1))$
$= f(f(0))$
$= f1$
$f(f(f1))$
$= f(f1)$
$= f1$

6 Higher-Order Propositional Completeness

This chapter was contributed by Mark Kaminski and is based on his Master's Thesis (2006).

Given a set of terms Γ, a specification A is called **deductively complete on Γ** (**Γ-complete**) if for every formula $t \in \Gamma$: $A \models t \Longrightarrow A \vdash t$.

The first completeness result for higher-order propositional logic was obtained by Leon Henkin in 1963. Henkin worked within a framework based on a countably infinite set of axioms. We, on the other hand, will focus our attention on PL, which is clearly finite. In the following, we will identify an expressive class of formulas on which PL is deductively complete.

As we already know, PL has exactly one model. We call this model $\mathcal{T}$. $\mathcal{T}$ uniquely determines the extension of all types and closed terms that are licensed by PL. It is sometimes convenient to be able to refer to these extensions without first having to construct an interpretation $\mathcal{I} \supseteq \mathcal{T}$.

Given a structure $\mathcal{A}$, we define $\hat{\mathcal{A}}$ as the smallest function such that:

1. $\forall T\colon T$ licensed by $\mathcal{A} \Longrightarrow \exists \mathcal{I} \supseteq \mathcal{A}\colon \hat{\mathcal{A}}T = \mathcal{I}T$
2. $\forall t\colon t$ closed and licensed by $\mathcal{A} \Longrightarrow \exists \mathcal{I} \supseteq \mathcal{A}\colon \hat{\mathcal{A}}t = \hat{\mathcal{I}}t$

By Proposition 3.7, $\hat{\mathcal{A}}$ is uniquely determined for every structure $\mathcal{A}$, and satisfies the following corollary.

Corollary 6.1 For every structure $\mathcal{A}$ and every interpretation $\mathcal{I} \supseteq \mathcal{A}$:

1. $\forall T\colon T$ licensed by $\mathcal{A} \Longrightarrow \mathcal{I}T = \hat{\mathcal{A}}T$
2. $\forall t\colon t$ closed and licensed by $\mathcal{A} \Longrightarrow \hat{\mathcal{I}}t = \hat{\mathcal{A}}t$

6.1 Denotational Completeness

Unfortunately, for formulas containing higher-order identities PL seems to be incomplete without additional axioms. Luckily, higher-order identities do not increase the semantic expressiveness of our language in $\mathcal{T}$. Every formula containing higher-order identities can be equivalently reformulated using only Boolean parameters and, optionally, identity on $\mathbf{B}$. Let us now make this claim formal.

A set of terms Γ is called **denotationally complete** with respect to a structure $\mathcal{A}$ if for every type T licensed by $\mathcal{A}$ and every value $v \in \hat{\mathcal{A}}T$ there exists a closed term $t \in \Gamma$ such that $\hat{\mathcal{A}}t = v$. We call t a **reification** of v in Γ with respect to $\mathcal{A}$.

Given a specification A, we define Δ_A to be the set of all terms licensed by A and containing no other parameters but those in $\mathcal{N}A$. We claim: Δ_{PL} is denotationally complete with respect to $\mathcal{T}$.

Since we will have to deal with very large terms, it is useful to introduce some abbreviating notation. Figure 15 summarizes the new abbreviations and

$$\uparrow_{T_1\to\ldots\to T_n\to\mathbf{B}} v := \lambda x_1 \ldots x_n. \bigvee_{\substack{w_i\in\hat{\mathcal{T}}T_i \\ v\,w_1\ldots w_n}} \bigwedge_{1\le j\le n} x_j \approx_{T_j} (\uparrow_{T_j} w_j) \qquad \text{D}\uparrow$$

$$\forall_T := \lambda f. \bigwedge_{v\in\hat{\mathcal{T}}T} f(\uparrow_T v) \qquad \text{D}\forall$$

$$\approx_{\mathbf{B}} := \doteq_{\mathbf{B}} \qquad \text{D}\approx$$

$$\approx_{S\to T} := \lambda f g. \forall_S x. f x \approx_T g x \qquad \text{D}\approx$$

Figure 15: Derived notation for PL

introduces names by which these can be referred to later. The definitions range over all types licensed by PL. As we will see shortly, a term of the form $\uparrow_T v$, where $v \in \hat{\mathcal{T}}T$, is a reification of v. Hence $\uparrow$ is called a **reification operator**. $\forall_T$ and $\approx_T$ denote the universal quantification over T and the identity predicate on T, respectively.

To convince oneself that the definition of $\uparrow$ covers all types licensed by PL, the following proposition is useful.

Proposition 6.2 For every type T there exist types $T_1, \ldots, T_n$ and a sort S such that $T = T_1 \to \ldots \to T_n \to S$.

Proof Exercise. ∎

When there is no danger of confusion, we will usually omit type annotations from $\uparrow v$, $\forall$, and $\approx$. In case the type cannot be uniquely inferred from the context, the abbreviated notation implicitly assumes universal quantification of the context over all types licensed by PL.

Example 6.3

- $\uparrow 0 = \bigvee_0 1 = 0$, $\uparrow 1 = 1$
- $\uparrow(\lambda v \in \mathbb{B}.1) = \lambda x.x \doteq_{\mathbf{B}} 0 \vee x \doteq_{\mathbf{B}} 1$
- $\uparrow(\lambda v \in \mathbb{B}.\lambda w \in \mathbb{B}.v \Rightarrow w)$
  $= \lambda xy.x \doteq_{\mathbf{B}} 0 \wedge y \doteq_{\mathbf{B}} 0 \vee x \doteq_{\mathbf{B}} 0 \wedge y \doteq_{\mathbf{B}} 1 \vee x \doteq_{\mathbf{B}} 1 \wedge y \doteq_{\mathbf{B}} 1$
- $\forall_{\mathbf{B}} f = f0 \wedge f1$
- $f \approx_{\mathbf{B}\to\mathbf{B}} g = f0 \doteq_{\mathbf{B}} g0 \wedge f1 \doteq_{\mathbf{B}} g1$ □

Proposition 6.4 $\uparrow v$, $\forall$, and $\approx$ are closed.

Proof By mutual induction on types. Exercise. ∎

Proposition 6.5 (Soundness)

1. $\hat{\mathcal{T}}(\uparrow v) = v$
2. $\hat{\mathcal{T}}\,\forall_T v = (v = \lambda w \in \hat{\mathcal{T}}T.1)$
3. $\hat{\mathcal{T}}(\approx) v\,w = (v = w)$

Proof By mutual induction on types. ∎

Corollary 6.6 Δ_{PL} is denotationally complete with respect to $\mathcal{T}$.

6.2 Definitional Extensions

Recall: $\theta(\doteq_T) = (\doteq_T)$ for every θ and every T.

We call a substitution θ **stable** if $\forall x \in Ker\theta\colon\ x \in Par \ \wedge\ \theta x$ closed.

Proposition 6.7 (Stability) If θ stable, then:

1. $A \models s \implies \theta A \models \theta s$
2. $A \vdash s \implies \theta A \vdash \theta s$

A function f is called **idempotent** if $f(fx) = fx$ for every argument x.

$A_\theta \ :=\ \{\, x = \theta x \mid x \in Ker\theta \,\}$

An equation is called **trivial** if it has the form $s = s$.

Proposition 6.8 θ idempotent if and only if every equation in $\theta(A_\theta)$ is trivial.

Let A be a specification. A substitution θ is called a **definitional extension of** A if θ is stable and idempotent, and satisfies $\theta A = A$. A specification B is called a definitional extension of A if $B \overset{A}{\vdash\!\dashv} A_\theta$ such that θ is a definitional extension of A.

Example 6.9

- PL − {E1, E0, I1, BCA} is a definitional extension of {E1, E0, I1, BCA}.
- $\{x \leftrightarrow y = (x \to y) \wedge (y \to x)\}$ is a definitional extension of PL. □

Proposition 6.10 Let θ be a definitional extension of A and $\theta t = t$. Then:

1. $A \cup A_\theta \models t \iff A \models t$
2. $A \cup A_\theta \vdash t \iff A \vdash t$

Proof Follows with Stability and Proposition 6.8. ∎

Proposition 6.11 Let θ be a definitional extension of A and $\forall t \in \Gamma\colon\ \theta t = t$. Then A is Γ-complete if and only if $A \cup A_\theta$ is Γ-complete.

Proof Follows from Proposition 6.10 ∎

By Proposition 6.11, to prove a specification A Γ-complete, it suffices to show Γ-completeness of any definitional extension θ of A such that $\theta\Gamma = \Gamma$. On the other hand, knowing that a specification A is Γ-complete, we immediately obtain completeness results for all of its definitional extensions that leave Γ invariant.

If we interpret $\uparrow v$, $\forall$, and $\approx$ not as notational abbreviations but as defined constants, one can show that the defining equations as given in Figure 15 form a definitional extension of PL. Hence we know by Proposition 6.11 that PL is Δ_{PL}-complete if and only if deductive completeness on Δ_{PL} holds in the extended system.

Assuming a specification A is Γ-complete, we can naturally extend Γ while preserving deductive completeness of A as stated by the following proposition.

Proposition 6.12 A specification A is Γ-complete if and only if it is deductively complete on $\{s \mid \exists t \in \Gamma\colon A \vdash s = t\}$.

Proof Exercise. ∎

Corollary 6.13 PL is Δ_{PL}-complete if and only if it is $\Delta_{\{\mathrm{E1,E0,I1,BCA}\}}$-complete.

Proposition 6.14 PL is Δ_{PL}-complete if and only if $\{\mathrm{E1}, \mathrm{E0}, \mathrm{I1}, \mathrm{BCA}\}$ is $\Delta_{\{\mathrm{E1,E0,I1,BCA}\}}$-complete.

Proof By Corollary 6.13, is suffices to show that PL is $\Delta_{\{\mathrm{E1,E0,I1,BCA}\}}$-complete if and only if $\{\mathrm{E1}, \mathrm{E0}, \mathrm{I1}, \mathrm{BCA}\}$ is $\Delta_{\{\mathrm{E1,E0,I1,BCA}\}}$-complete. This is the case by Proposition 6.11. ∎

6.3 Deductive Completeness

Proposition 6.15 Let X, Y be finite sets. Then

$$\mathrm{TL} \vdash \bigwedge_{x \in X} \bigvee_{y \in Y} f x y = \bigvee_{g \in X \to Y} \bigwedge_{x \in X} f x (g x)$$

Proof By induction on $|X|$. ∎

Example 6.16 Let $X = Y = \{0, 1\}$, and let $f x y = t_{xy}$. Then
$(t_{00} \vee t_{01}) \wedge (t_{10} \vee t_{11}) = (t_{00} \wedge t_{10}) \vee (t_{00} \wedge t_{11}) \vee (t_{01} \wedge t_{10}) \vee (t_{01} \wedge t_{11})$ □

Corollary 6.17 $\mathrm{TL} \vdash \bigvee_{x \in X} s_x \wedge t = \left(\bigvee_{x \in X} s_x \right) \wedge t$

Proposition 6.18 Every closed, β-normal formula $t \in \Delta_{\mathrm{PL}}$ is propositional.

Proof By induction on $|t|$. Since abstractions have functional types, t cannot be of the form $\lambda x.s$. The remaining cases are:

Case $t \in Nam$. Since t is closed, $t \notin Var$. Since $t \in \Delta_{\mathrm{PL}}$ and $t : \mathbf{B}$, it must be either 0 or 1.

Case $t = t_1 t_2 \ldots t_n$ where $n \geq 2$ and t_1 not an application. Since t is β-normal, t_1 is not of the form $\lambda x.s$. And since t is closed, t_1 is not a variable. Consequently, t_1 is either $\rightarrow$, $\neg$, $\wedge$, or $\vee$. In either case $t_2, \ldots, t_n$ must be closed, β-normal formulas. By induction for $t_2, \ldots, t_n$, they are propositional. Hence t is propositional. ∎

Proposition 6.19 If $t \in \Delta_{\mathrm{PL}}$ is closed, then: $\mathrm{PL} \models t \Longrightarrow \mathrm{PL} \vdash t$.

Proof Assume $\mathrm{PL} \models t$, and let s be the β-normal form of t. By Proposition 2.13, $s : \mathbf{B}$. By Proposition 3.19, s is closed and $s \in \Delta_{\mathrm{PL}}$. Hence, by Proposition 6.18, s is propositional. So, by assumption, s is a tautology. By Tautological Completeness (Proposition 5.15), $\mathrm{PL} \vdash s$. Hence $\mathrm{PL} \vdash t$. ∎

Proposition 6.20 PL is Δ_{PL}-complete if and only if $\forall_T f \rightarrow fx$ is deducible from PL for every type T licensed by PL.

Proof

"$\Longrightarrow$" Since, by Soundness (Proposition 6.5), $PL \models \forall f \rightarrow fx$, the formula is deducible from PL by the completeness assumption.

"$\Longleftarrow$" Assume $\mathrm{PL} \vdash \forall f \rightarrow fx$ and, for some formula $t \in \Delta_{\mathrm{PL}}$, $\mathrm{PL} \models t$. To show: $\mathrm{PL} \vdash t$. Let $\{x_1, \ldots, x_n\}$ be the variables that occur in t. By Soundness (Proposition 6.5), $\mathrm{PL} \models \forall x_1 \ldots \forall x_n.t$. By Proposition 6.4, $\forall x_1 \ldots \forall x_n.t$ is closed. Hence, by Proposition 6.19, $\mathrm{PL} \vdash \forall x_1 \ldots \forall x_n.t$. By n successive applications of the assumption and Modus Ponens, we obtain $\mathrm{PL} \vdash t$. ∎

So, in order to prove PL Δ_{PL}-complete, it suffices to show $\mathrm{PL} \vdash \forall_T f \rightarrow fx$ for every type T licensed by PL.

Proposition 6.21 $\mathrm{PL} \vdash x \approx_T x$

Proof By induction on T. ∎

Proposition 6.22 $\mathrm{PL} \vdash (\uparrow_{T_1 \rightarrow T_2} v)(\uparrow_{T_1} w) = (\uparrow_{T_2} (v\,w))$

Proof We prove

1. If $T = T_1 \to T_2$, then $\mathrm{PL} \vdash (\uparrow_{T_1 \to T_2} v)(\uparrow_{T_1} w) = (\uparrow_{T_2} (v\,w))$.
2. If $v, w \in \hat{\mathcal{T}}\,T$ are disjoint, then $\mathrm{PL} \vdash (\uparrow_T v) \approx_T (\uparrow_T w) = 0$.

by mutual induction on T.

Case $T = \mathbf{B}$. (1) holds vacuously. For (2) it suffices to check that both $(0 = 1) = 0$ and $(1 = 0) = 0$ are tautologous.

Case $T = T_1 \to T_2$ where $T_2 = T_{21} \to \ldots \to T_{2n} \to \mathbf{B}$.

1. By D$\uparrow$ and β, we have

$$
\begin{aligned}
&(\uparrow_{T_1 \to T_{21} \to \ldots \to T_{2n} \to \mathbf{B}} v)(\uparrow_{T_1} w) \\
&= \left(\lambda x_1 x_{21} \ldots x_{2n}. \bigvee_{\substack{u_i \in \hat{\mathcal{T}}\,T_i \\ v\,u_1 u_{21} \ldots u_{2n}}} x_1 \approx_{T_1} (\uparrow_{T_1} u_1) \wedge \bigwedge_{1 \le j \le n} x_{2j} \approx_{T_{2j}} (\uparrow_{T_{2j}} u_{2j}) \right) (\uparrow_{T_1} w) \\
&= \lambda x_{21} \ldots x_{2n}. \bigvee_{\substack{u_i \in \hat{\mathcal{T}}\,T_i \\ v\,u_1 u_{21} \ldots u_{2n}}} (\uparrow_{T_1} w) \approx_{T_1} (\uparrow_{T_1} u_1) \wedge \bigwedge_{1 \le j \le n} x_{2j} \approx_{T_{2j}} (\uparrow_{T_{2j}} u_{2j})
\end{aligned}
$$

   By Proposition 6.21 and induction for T_1 (2), respectively, we know:

   - $\mathrm{PL} \vdash (\uparrow_{T_1} w) \approx_{T_1} (\uparrow_{T_1} w) = 1$
   - $\mathrm{PL} \vdash (\uparrow_{T_1} w) \approx_{T_1} (\uparrow_{T_1} u_1) = 0$ if $u_1 \neq w$

   Hence

$$
\begin{aligned}
&\lambda x_{21} \ldots x_{2n}. \bigvee_{\substack{u_i \in \hat{\mathcal{T}}\,T_i \\ v\,u_1 u_{21} \ldots u_{2n}}} (\uparrow_{T_1} w) \approx_{T_1} (\uparrow_{T_1} u_1) \wedge \bigwedge_{1 \le j \le n} x_{2j} \approx_{T_{2j}} (\uparrow_{T_{2j}} u_{2j}) \\
&= \lambda x_{21} \ldots x_{2n}. \bigvee_{\substack{u_i \in \hat{\mathcal{T}}\,T_i \\ v\,w\,u_{21} \ldots u_{2n}}} \bigwedge_{1 \le j \le n} x_{2j} \approx_{T_{2j}} (\uparrow_{T_{2j}} u_{2j}) \\
&= (\uparrow_{T_{21} \to \ldots \to T_{2n}} (v\,w))
\end{aligned}
$$

2. Let $v, w \in \hat{\mathcal{T}}\,T$ be distinct. Then there exists $u \in \hat{\mathcal{T}}\,T_1$ such that $v\,u, w\,u$ are distinct. Thus, by induction for T_2, $\mathrm{PL} \vdash (\uparrow_{T_2} (v\,u)) \approx_{T_2} (\uparrow_{T_2} (w\,u)) = 0$. By (1), this is equivalent to $\mathrm{PL} \vdash (\uparrow_T v)(\uparrow_{T_1} u) \approx_{T_2} (\uparrow_T w)(\uparrow_{T_1} u) = 0$. Hence

$$
\begin{aligned}
0 &= \bigwedge_{u \in \hat{\mathcal{T}}\,T_1} (\uparrow_T v)(\uparrow_{T_1} u) \approx_{T_2} (\uparrow_T w)(\uparrow_{T_1} u) && \text{TL} \\
&= \forall_{T_1} x.(\uparrow_T v)x \approx_{T_2} (\uparrow_T w)x && \text{D}\forall \\
&= (\uparrow_T v) \approx_T (\uparrow_T w) && \text{D}\approx
\end{aligned}
$$

■

Proposition 6.23 (Enum) $\mathrm{PL} \vdash \bigvee_{v \in \hat{\mathcal{I}} T} x \approx (\uparrow v)$

Proof We prove $\mathrm{PL} \vdash \bigvee_{v \in \hat{\mathcal{I}} T} x \approx (\uparrow v) = 1$ by induction on T.

Case $T = \mathbf{B}$. The claim follows by D$\approx$ on $\mathbf{B}$ and Tautological Completeness (Proposition 5.15).

Case $T = T_1 \to T_2$.

$$\begin{aligned}
\bigvee_{v \in \hat{\mathcal{I}} T} x \approx (\uparrow v) &= \bigvee_{v \in \hat{\mathcal{I}} T} \forall_{T_1} y. xy \approx (\uparrow v) y && \mathrm{D}\approx \\
&= \bigvee_{v \in \hat{\mathcal{I}} T} \bigwedge_{w \in \hat{\mathcal{I}} T_1} x(\uparrow w) \approx (\uparrow v)(\uparrow w) && \mathrm{D}\forall \\
&= \bigvee_{v \in \hat{\mathcal{I}} T_1 \to \hat{\mathcal{I}} T_2} \bigwedge_{w \in \hat{\mathcal{I}} T_1} x(\uparrow w) \approx (\uparrow (v w)) && \text{Prop. 6.22} \\
&= \bigwedge_{w \in \hat{\mathcal{I}} T_1} \bigvee_{u \in \hat{\mathcal{I}} T_2} x(\uparrow w) \approx (\uparrow u) && \text{Prop. 6.15} \\
&= \bigwedge_{w \in \hat{\mathcal{I}} T_1} 1 && \text{induction for } T_2 \\
&= 1 && \mathrm{TL}
\end{aligned}$$

■

Proposition 6.24 (MP') If $\mathrm{PL} \vdash s \to t = 1$, then there exists a conversion proof of $t = 1$ from PL with $s = 1$.

Proof $t = 1 \to t = s \to t = 1$ ■

Proposition 6.25 $x \approx_T y \to fx \to fy \overset{\mathrm{PL}}{\vdash} x \approx_T y \wedge fx = x \approx_T y \wedge fy$

Proposition 6.26

1. There exists a conversion proof of $s = t$ from PL with $s \approx_T t = 1$.
2. $\mathrm{PL} \vdash x \approx_T y \to fx \to fy$
3. $\mathrm{PL} \vdash \forall_T f \to fx$

Proof

1. We show (1) $\Longrightarrow$ (2). Assume (1). By TL, it suffices to show:

   $$\mathrm{PL} \vdash (x \approx_T y = 1) \to (fx = 1) \to (fy = 1)$$

   Therefore, by Boolean Deductivity (Proposition 5.27), it suffices to find a conversion proof of $fy = 1$ from PL with $\{x \approx_T y = 1, fx = 1\}$. Since, by (1), there exists a conversion proof of $x = y$ from PL with $x \approx_T y = 1$, it suffices to prove $fy = 1$ by conversion from PL with $\{x = y, fx = 1\}$. We do it as follows: $fy = fx = 1$.

2. We show (2) $\Longrightarrow$ (3). Assume (2) and let $v \in \hat{\mathcal{T}}\,T$. Then:

$$\begin{aligned} x \approx (\uparrow v) &= x \approx (\uparrow v) \wedge \left(\left(\bigwedge_{w \in \hat{\mathcal{T}}\,T} f(\uparrow w) \right) \to f(\uparrow v) \right) && \text{TL} \\ &= x \approx (\uparrow v) \wedge (\forall f \to f(\uparrow v)) && \text{D}\forall \\ &= x \approx (\uparrow v) \wedge (\forall f \to fx) && \text{(2), Prop. 6.25} \end{aligned}$$

By the above:

$$\begin{aligned} \forall f \to fx &= \left(\bigvee_{v \in \hat{\mathcal{T}}\,T} x \approx (\uparrow v) \right) \wedge (\forall f \to fx) && \text{Enum, TL} \\ &= \bigvee_{v \in \hat{\mathcal{T}}\,T} x \approx (\uparrow v) \wedge (\forall f \to fx) && \text{Prop. 6.17} \\ &= \bigvee_{v \in \hat{\mathcal{T}}\,T} x \approx (\uparrow v) \\ &= 1 && \text{Enum, TL} \end{aligned}$$

3. We show (1) by induction on T. By the above, the inductive hypothesis can always be weakened to a corresponding instance of (2) or (3).

Case $T = \mathbf{B}$.

$$\begin{aligned} s &= (s \approx_{\mathbf{B}} t) \wedge s && \text{TL, Hyp } s \approx_{\mathbf{B}} t = 1 \\ &= (s \doteq_{\mathbf{B}} t) \wedge s && \text{D}\approx \\ &= (s \doteq_{\mathbf{B}} t) \wedge t && \text{Prop. 5.24} \\ &= (s \approx_{\mathbf{B}} t) \wedge t && \text{D}\approx \\ &= t && \text{TL, Hyp } s \approx_{\mathbf{B}} t = 1 \end{aligned}$$

Case $T = T_1 \to T_2$. By (3) for T_1, and TL:

$$\text{PL} \vdash \forall_{T_1} (\lambda x.sx \approx_{T_2} tx) \to sx \approx_{T_2} tx = 1$$

Hence, by MP', there exists a conversion proof of $sx \approx_{T_2} tx = 1$ from PL with $\forall_{T_1} (\lambda x.sx \approx_{T_2} tx) = 1$, i.e. $s \approx_T t = 1$. By induction for T_2, there exists a conversion proof of $sx = tx$ from PL with $sx \approx_{T_2} tx = 1$. Hence, there exists a conversion proof of $sx = tx$ from PL with $s \approx_T t = 1$. Finally, by Lifting (Proposition 5.25), there exists a conversion proof of $(\lambda x.sx) = (\lambda x.tx)$ from PL with $s \approx_T t = 1$. The claim follows by β-conversion. ∎

Specification	$\mathrm{PL}_=$	
Extends	PL	
Axioms	$(x \doteq_T y) \to fx \to fy$	Rep (General Replacement)

Figure 16: Propositional logic with replacement axioms

6.4 Higher-Order Identities

In the previous section we have shown PL deductively complete on Δ_{PL}. Now we ask ourselves which axioms we need additionally to obtain deductive completeness on all formulas licensed by PL, in particular on those containing higher-order identities. It turns out that we need infinitely many axioms, one for each type T licensed by PL: $(x \doteq_T y) \to fx \to fy$.

Proposition 6.27 $\mathrm{PL}_= \vdash (\doteq_T) = (\approx_T)$

Proof By induction on T, using several lemmas. ∎

Proposition 6.28 $\mathrm{PL}_=$ is deductively complete on all terms that are licensed by PL (or, equivalently, $\mathrm{PL}_=$).

Proof By Proposition 6.27 and 6.12. ∎

Specification	HL	
Extends	PL	
Constants	$\forall_T, \exists_T : (T \to \mathbf{B}) \to \mathbf{B}$	for every type T
	$\mathrm{C}_T : (T \to \mathbf{B}) \to T$	for every type T
Notation	$\forall x.s := \forall(\lambda x.s)$	
	$\exists x.s := \exists(\lambda x.s)$	
	$\mathrm{C}x.s := \mathrm{C}(\lambda x.s)$	
	$Px\,Qy.s := Px.Qy.s$	for $P, Q \in \{\forall, \exists, \mathrm{C}\}$
Axioms	$(x = y) \to fx = (x = y) \to fy$	Rep (Replacement)
	$(\forall x.fx = gx) = (f = g)$	Ext (Extensionality)
	$\exists f = \neg(f = \lambda x.0)$	Definition $\exists$
	$f(\mathrm{C}f) = \exists f$	Choice

Figure 17: Axioms for identities, quantifiers, and choices

7 Identities and Quantifiers

We will now explore a specification HL that extends PL with the missing axioms so that deductive entailment lives up to our semantic expectations. Besides supplying the missing axioms for the identities, HL axiomatizes quantifiers and choice operators.

7.1 Specification HL

Figure 17 shows the specification HL we will explore in this chapter. HL extends the specification PL, i.e., PL $\subset$ HL. For every type T, HL axiomatizes the quantifiers $\forall_T$ and $\exists_T$ and the choice operator C_T. Note that HL is infinite since what appears in Figure 17 are in fact schemes that may be instantiated for all admissible types. The replacement axiom is semantically entailed. The job of the extensionality axiom consists mainly in supplying an important property of identities. As a side effect, it also axiomatizes the universal quantifier.

First we show that the replacement axioms yield deductivity for all types.

Lemma 7.1 Suppose A contains the replacement axiom for $\doteq_T$ and there is a conversion proof of $t = t'$ from A with $H \cup \{s\}$ where s is an equation obtained with $\doteq_T$. Then there is a conversion proof of $s \to t = s \to t'$ from A with H.

Proof Analogous to the proof of Lemma 5.26. ∎

Proposition 7.2 (Deductivity) Suppose $A \vdash \mathrm{TL}$ and A contains the replacement axioms for all identities. Then $A \vdash s_1 \to \cdots \to s_n \to s$ holds if $s_1, \ldots, s_n$ are equations and there is a conversion proof of s from A with $\{s_1, \ldots, s_n\}$.

Proposition 7.3 (Reflexivity) $\emptyset \vdash x = x$

Proof Follows with the deduction rule Ref in Figure 9. ∎

Proposition 7.4 (Symmetry) $\mathrm{HL} \vdash (x = y) = (y = x)$

Proof By Equiv it suffices to show that $(x{=}y) \to (y{=}x)$ and $(y{=}x) \to (x{=}y)$ are deducible from HL. This follows with deductivity. ∎

Proposition 7.5 (Transitivity) $\mathrm{HL} \vdash (x = y) \to (y = z) \to (x = z)$

Proof By deductivity it suffices to show that there is a conversion proof of $x = z$ from HL with $\{x = y,\ y = z\}$. This is obviously the case. ∎

Proposition 7.6 (Replacement) $\mathrm{HL} \vdash (x = y) \to (fx = fy)$

Proof

$$
\begin{array}{lll}
(x = y) \to (fx = fy) = (x = y) \to (\lambda x.\, fx = fy)x & \beta \\
\qquad = (x = y) \to (\lambda x.\, fx = fy)y & \mathrm{Rep} \\
\qquad = (x = y) \to (fy = fy) & \beta \\
\qquad = (x = y) \to 1 & \mathrm{Ref} \\
\qquad = 1 & \mathrm{TL}
\end{array}
$$
∎

Proposition 7.7 (Replacement) Let $f : T \to \mathbf{B}$ be a variable. Then the following formulas are deducible from HL and are equivalent with respect to $\overset{\mathrm{TL}}{\vdash}$:

1. $(x = y) \to fx \to fy$
2. $(x = y) \to (fx = fy)$
3. $(x = y) \to fx \;=\; (x = y) \to fy$
4. $(x = y) \wedge fx \;=\; (x = y) \wedge fy$

Note that (2) is less general than Proposition 7.6.

Proof The following formulas are tautologous: (2) → (3), (3) = (4), (3) → (1). Hence it suffices to show that TL, (1) ⊢ (2).

$$
\begin{array}{lll}
(x = y) \to (fx = fy) & \\
\quad = (x = y) \to (fx = fx) \to (fx = fy) & \mathrm{TL} \\
\quad = (x = y) \to (\lambda y.fx = fy)x \to (\lambda y.fx = fy)y & \beta \\
\quad = 1 & (1)
\end{array}
$$
∎

7.2 Quasi-Conversion

Conversion proofs are a convenient way to verify deductive claims of the form $A \vdash s$. To obtain a short proof, a claim $A \vdash s$ is typically shown with a conversion proof from $A \cup B$, where B contains so-called **lemmas** whose deducibility from A has been established before. Given the right lemmas, deductive claims can be verified with short proofs.

When we construct a proof of $A \vdash s$, we will admit obvious consequences of the assumptions in A as so-called **implicit lemmas**. For instance, if $A \vdash \mathrm{TL}$, we may admit all tautologous formulas as implicit lemmas. In the following we will define a more permissive class of implicit lemmas.

We call a term s a **λ-instance** of a term t if there exists an instance of t that reduces to s. For example:

- $(\forall x.\, s) \to s$ is a λ-instance of $\forall f \to fx$.
- $(\forall x.\, fx) \to fx$ is a λ-instance of $\forall f \to fx$.
- $[x := t]s \to \exists x.\, s$ is a λ-instance of $fx \to \exists f$.
- $[x := 0]s \to [x := 1]s \to s$ is a λ-instance of $f0 \to f1 \to fx$.

Proposition 7.8 Let s be be a λ-instance of a formula t. Then $\{t\} \vdash s$.

We call a formula s a **quasi-instance** of a formula t if there exists a λ-instance t' of t such that $t' \to s$ is tautologous. Here are examples:

- Every tautologous formula is a quasi-instance of every formula.
- $s = 1$ is a quasi-instance of s if s is a formula.
- $s \to t$ is a quasi-instance of $s = t$ if s and t are formulas.
- $s = s \wedge t$ is a quasi-instance of $s \to t$.
- $t = s \vee t$ is a quasi-instance of $s \to t$.
- $\exists x.1$ is a quasi-instance of $fx \to \exists f$.
- $(x = y) \to (fx = fy)$ is a quasi-instance of $(x = y) \to fx \to fy$.

Proposition 7.9 Let s be be a quasi-instance of a formula t. Then $\mathrm{TL} \cup \{t\} \vdash s$.

We call an equation a **quasi-conversion step from A** if it is a conversion step from A or if it has the form $Ct = Ct'$, where C is a context and $t = t'$ or $t' = t$ is a quasi-instance of a formula in A. A **quasi-conversion proof** is defined like a conversion proof except that its steps are quasi-conversion steps.

Proposition 7.10 $A \vdash s$ if there exists a quasi-conversion proof of s from A and $A \vdash \mathrm{TL}$.

We will not exploit the full power of quasi-conversion steps since they can be rather difficult to verify. It took 30 years until the decidability of the λ-instance relation could be shown (Decidability of higher-order matching, Colin Stirling, ICALP 2006). We do not know whether the quasi-instance relation is decidable.

7.3 Quantifier Laws

The quantifier laws shown in Figure 18 express important properties of the quantifiers. Except for Generalization, the laws are formulas that are deducible in HL. We will verify some of the laws and leave the verification of the others as exercises.

Proposition 7.11 (Definition ∀) $\mathrm{HL} \vdash \forall f = (f = \lambda x.1)$

Proof By a quasi-conversion proof from HL.

$$\begin{aligned}
(f = \lambda x.1) &= \forall x.\, fx = (\lambda x.1)x && \text{Ext}\\
&= \forall x.\, fx = 1 && \beta\\
&= \forall x.\, fx && \text{TL}\\
&= \forall f && \eta
\end{aligned}$$

■

Note that the first two steps of the proof could be merged into one step since $(f = \lambda x.1) = \forall x.\, fx = 1$ is a λ-instance of Ext.

Proposition 7.12 (Instantiation) $\mathrm{HL} \vdash \forall f \to fx$

Proof

$$\begin{aligned}
(\forall f \to fx) &= (f = \lambda x.1) \to fx && \text{Def } \forall\\
&= (f = \lambda x.1) \to (\lambda x.1)x && \text{Rep}\\
&= (f = \lambda x.1) \to 1 && \beta\\
&= 1 && \text{TL}
\end{aligned}$$

■

Note that the last three steps of the proof could be merged into one step since $(f = \lambda x.1) \to fx = 1$ is a quasi-instance of Rep.

Proposition 7.13 (Elimination) Let $x \neq y$. Then $\mathrm{HL} \vdash (\forall x.\, y) = y$.

Proof By Equiv and 2 quasi-conversion proofs.

$$(\forall x.\, y) \to y = 1 \qquad \text{Inst } \forall$$

$$y \to (\forall x.\, y) = (y = 1) \to \forall x.\, y \qquad \text{TL}$$

Extensionality

$$(\forall x.\, fx = gx) = (f = g)$$

Definition

$$\forall f = (f = \lambda x.1) \qquad \exists f = \neg(f = \lambda x.0)$$

Instantiation

$$\forall f \to fx \qquad fx \to \exists f$$

Elimination

$$(\forall x.y) = y \qquad (\exists x.y) = y$$

Generalization

$$\forall x.\, s \overset{\text{HL}}{\vdash} s$$

Pull for $\wedge$, $\vee$

$$\forall f \wedge x = \forall y.\, fy \wedge x \qquad \exists f \wedge x = \exists y.\, fy \wedge x$$
$$\forall f \vee x = \forall y.\, fy \vee x \qquad \exists f \vee x = \exists y.\, fy \vee x$$
$$\forall f \wedge \forall g = \forall x.\, fx \wedge gx \qquad \exists f \vee \exists g = \exists x.\, fx \vee gx$$

De Morgan

$$\neg(\forall x.\, fx) = \exists x.\, \neg(fx) \qquad \neg(\exists x.\, fx) = \forall x.\, \neg(fx)$$

Pull for $\to$

$$x \to \forall f = \forall y.\, x \to fy \qquad x \to \exists f = \exists y.\, x \to fy$$
$$\forall f \to x = \exists y.\, fy \to x \qquad \exists f \to x = \forall y.\, fy \to x$$
$$\forall f \to \exists g = \exists x.\, fx \to gx$$

Commutativity

$$(\forall x \forall y.\, s) = \forall y \forall x.\, s \qquad (\exists x \exists y.\, s) = \exists y \exists x.\, s$$

Skolem

$$(\forall x \exists y.\, fxy) = \exists g \forall x.\, fx(gx) \qquad (\exists x \forall y.\, fxy) = \forall g \exists x.\, fx(gx)$$

Figure 18: Quantifier laws (pull laws assume $x \neq y$)

$$
\begin{aligned}
&= (y = 1) \to \forall x.\, 1 && \text{Rep}\\
&= (y = 1) \to ((\lambda x.\, 1) = (\lambda x.\, 1)) && \text{Def } \forall\\
&= 1 && \text{TL}
\end{aligned}
$$
■

Convince yourself that the second quasi-conversion proof can be written more compactly by fully exploiting the power of quasi-conversion steps:

$$
\begin{aligned}
y \to (\forall x.\, y) &= y \to \forall x.\, 1 && \text{Rep}\\
&= 1 && \text{Def } \forall
\end{aligned}
$$

Proposition 7.14 (Generalization) $\forall x.\, s \overset{\text{HL}}{\vdash\!\dashv} s$

Proof By 2 quasi-conversion proofs.

$$
\begin{aligned}
(\forall x.\, s) &= \forall x.\, 1 && s\\
&= 1 && \text{Elim } \forall
\end{aligned}
$$

$$
\begin{aligned}
s &= (\forall x.\, s) \to s && \forall x.\, s\\
&= 1 && \text{Inst } \forall
\end{aligned}
$$
■

Proposition 7.15 (Pull $\forall\wedge$) Let $x \neq y$. Then $\text{HL} \vdash \forall f \wedge x = \forall y.\, f y \wedge x$.

Proof By BCA on x and 2 conversion proofs.

$$
\begin{aligned}
\forall f \wedge 0 &= 0 && \text{TL}\\
&= \forall y.\, 0 && \text{Elim } \forall\\
&= \forall y.\, f y \wedge 0 && \text{TL}
\end{aligned}
$$

$$
\begin{aligned}
\forall f \wedge 1 &= \forall f && \text{TL}\\
&= \forall y.\, f y && \eta\\
&= \forall y.\, f y \wedge 1 && \text{TL}
\end{aligned}
$$
■

Proposition 7.16 (De Morgan) $\text{HL} \vdash \overline{\forall x.\, f x} = \exists x.\, \overline{f x}$

Proof By Equiv.

$$
\begin{aligned}
\overline{\forall x.\, f x} \to \exists x.\, \overline{f x} &= \forall f \vee \exists x.\, \overline{f x} && \eta,\ \text{TL}\\
&= \forall x.\, f x \vee \exists x.\, \overline{f x} && \text{Pull}
\end{aligned}
$$

$$
\begin{aligned}
&= \forall x.1 && \text{Inst } \exists \\
&= 1 && \text{Elim } \forall
\end{aligned}
$$

$$
\begin{aligned}
(\exists x.\overline{fx}) \to \overline{\forall x.fx} &= \overline{(\exists x.\overline{fx}) \wedge \forall f} && \eta, \text{ TL} \\
&= \overline{\exists x.\overline{fx} \wedge \forall f} && \text{Pull} \\
&= \overline{\exists x.\overline{fx} \wedge \forall f \wedge fx} && \text{Inst } \forall \\
&= \overline{\exists x.0} && \text{TL} \\
&= 1 && \text{Elim } \exists
\end{aligned}
$$

■

Proposition 7.17 (Pull $\to\forall$) Let $x \neq y$. Then $\text{HL} \vdash x \to \forall f = \forall y.x \to fy$.

Proof The claim is a quasi-instance of the pull law for $\forall$ and $\vee$. A more detailed conversion proof looks as follows:

$$
\begin{aligned}
x \to \forall f &= \forall f \vee \overline{x} && \text{TL} \\
&= \forall y.fy \vee \overline{x} && \text{Pull } \forall\vee \\
&= \forall y.x \to fy && \text{TL}
\end{aligned}
$$

■

Proposition 7.18 (Pull $\exists\to$) Let $x \neq y$. Then $\text{HL} \vdash \exists f \to x = \forall y.fy \to x$.

Proof

$$
\begin{aligned}
\exists f \to x &= \overline{\exists f} \vee x && \text{TL} \\
&= (\forall y.\overline{fy}) \vee x && \text{de Morgan} \\
&= \forall y.\overline{fy} \vee x && \text{Pull} \\
&= \forall y.fy \to x && \text{TL}
\end{aligned}
$$

■

7.4 Correctness of Henkin's Reduction

Henkin discovered that the Boolean connectives can be expressed with the identities (see Figure 5 in Chapter 3). As one would expect, Henkin's equations are deducible in HL. Proposition 7.11 states the deducibility of the equation for the universal quantifier. We now prove that the equations for 0 and conjunction are deducible.

Proposition 7.19 (Definition 0) $\text{HL} \vdash 0 = ((\lambda x.1) = \lambda x.x)$

Proof

$$\begin{array}{lll} ((\lambda x.1) = \lambda x.x) = \forall x.\, 1 = x & & \text{Ext} \\ = \forall x.\, x & & \text{TL} \\ = (\forall x.\, x) \wedge 0 & & \text{Inst } \forall \\ = 0 & & \text{TL} \end{array}$$

■

Proposition 7.20 (Definition $\wedge$) Let $f : \mathbf{B} \to \mathbf{B} \to \mathbf{B}$. Then:
$\text{HL} \vdash x \wedge y = ((\lambda f.f11) = \lambda f.fxy)$.

Proof

$$\begin{array}{lll} ((\lambda f.f11) = \lambda f.fxy) = \forall f.\, f11 = fxy & & \text{Ext} \\ = (\forall f.\, f11 = fxy) \wedge (1 \wedge 1 = x \wedge y) & & \text{Inst } \forall \\ = (\forall f.\, f11 = fxy) \wedge x \wedge y & & \text{TL} \\ = x \wedge y & & * \end{array}$$

The last step $(*)$ follows with BCA (zero-one instances), Ref and Elim $\forall$. ■

7.5 Backward Proofs

To prove a formula s with quantifiers, its often helpful to attempt the construction of a quasi-conversion proof of $s = 1$, with some **backward steps** $t \dashv t'$ mixed in. Such a proof can be compiled into derivation of s. Since the compilation into derivations reverses the order of the proof steps, we call such proofs **backward proofs**. The backward steps will be justified by the deduction rules in Figure 19. To obtain a backward step, the rules are applied from bottom to top (e.g., $\exists x.\, s \dashv [x{:=}t]s$). Here is an example of a backward proof that uses the **abstraction rule** Abs:

$$\begin{array}{lll} \exists x.\, x = y & \dashv & y = y \quad \text{Abs } x := y \\ & = & 1 \quad \text{Ref} \end{array}$$

Note that Abs is a special case of the **quasi-modus ponens** rule QMP where the formula in A is the instantiation law $fx \to \exists f$.

Except for Abs, CA and QMP, the rules in Figure 19 are bidirectionally sound. The rules Equiv, And, BCA and CA are non-linear (i.e., they have more than one premise). You have seen them before in the chapter on propositional logic.

Exercise 7.21 Explain why $fx \to s \vdash \forall f \to s$ follows with QMP from the instantiation law for $\forall$.

Gen $\dfrac{s}{\forall x.\, s}$ **Abs** $\dfrac{[x{:=}t]s}{\exists x.\, s}$

Equiv $\dfrac{s \to t \quad t \to s}{s = t}$ **And** $\dfrac{s \quad t}{s \wedge t}$

BCA $\dfrac{[x{:=}0]s \quad [x{:=}1]s}{s}$ **CA** $\dfrac{t_1 \vee t_2 \quad t_1 \to s \quad t_2 \to s}{s}$

QC $\dfrac{s}{t}$ $s = t$ quasi-conversion step from A

QMP $\dfrac{s}{t}$ $s \to t$ quasi-instance of formula in A

Figure 19: Sound deduction rules for $\overset{A}{\vdash}$ where $A \vdash \mathrm{HL}$

Proposition 7.22 (Double Pull) $\mathrm{HL} \vdash \forall f \wedge \forall g = \forall x.\, fx \wedge gx$

Proof By Equiv and two linear backward proofs. We show one and leave the other as an exercise. The backward step is justified by the bidirectionally sound rule Gen, which makes it possible to drop outermost universal quantifiers.

$\forall f \wedge \forall g \to \forall x.\, fx \wedge gx$	
$= \forall x.\, \forall f \wedge \forall g \to fx \wedge gx$	Pull $\to\forall$
$\vdash \forall f \wedge \forall g \to fx \wedge gx$	Gen
$= \forall f \wedge fx \wedge \forall g \to fx \wedge gx$	Inst $\forall$
$= \forall f \wedge fx \wedge \forall g \wedge gx \to fx \wedge gx$	Inst $\forall$
$= 1$	TL

■

Proposition 7.23 (Commutativity) $\mathrm{HL} \vdash (\forall x \forall y.\, s) = \forall y \forall x.\, s$

Proof By Equiv. One proof suffices for both subgoals.

$((\forall x \forall y.\, s) \to \forall y \forall x.\, s) \vdash (\forall x \forall y.\, s) \to s$		Pull, Gen
	$= (\forall x \forall y.\, s) \wedge (\forall y.\, s) \to s$	Inst $\forall$
	$= (\forall x \forall y.\, s) \wedge (\forall y.\, s) \wedge s \to s$	Inst $\forall$
	$= 1$	TL

■

7.6 Turing's Law and Cantor's Law

We now consider two convincing examples for backward proofs that are also interesting for other reasons.

How would you answer the following questions?

1. On a small island, can there be a barber who shaves everyone who doesn't shave himself?
2. Does there exist a Turing machine that halts on the representation of a Turing machine x if and only if x does not halt on the representation of x?
3. Does there exist a set that contains a set x as element if and only if $x \notin x$?

The answer to all 3 questions is no, and this no has purely logical reasons.

Proposition 7.24 (Turing) Let $f : T \to T \to \mathbf{B}$. Then:
$\mathrm{HL} \vdash \neg(\exists f \exists x \forall y.\, fxy = \overline{fyy})$

Proof

$$\begin{aligned}
\neg(\exists f \exists x \forall y.\, fxy = \overline{fyy}) &= \forall f \forall x \exists y.\, fxy = fyy && \text{de Morgan, TL} \\
&\dashv fxx = fxx && \text{Gen, Abs } y := x \\
&= 1 && \text{TL}
\end{aligned}$$

■

Cantor's Theorem says that for no set X there is a surjective function $X \to \mathcal{P}(X)$. The non-countability of the power set of the natural numbers is a consequence of this theorem. The usual proof of the theorem uses a famous argument known as diagonalization. Nevertheless, Cantor's Theorem holds for purely logical reasons and has a straightforward formal proof. Recall that we can represent sets whose elements are of type T by their characteristic functions, which have the type $T \to \mathbf{B}$.

Proposition 7.25 (Cantor) Let $f : T \to T \to \mathbf{B}$. Then:
$\mathrm{HL} \vdash \neg(\exists f \forall g \exists x.\, fx = g)$

Proof

$$\begin{aligned}
\neg(\exists f \forall g \exists x.\, fx = g) &= \neg(\exists f \forall g \exists x \forall y.\, fxy = gy) && \text{Ext} \\
&= \forall f \exists g \forall x \exists y.\, \neg(fxy = gy) && \text{de Morgan} \\
&\dashv \forall x \exists y.\, \neg(fxy = \overline{fyy}) && \text{Gen, Abs } g := \lambda y.\, \overline{fyy} \\
&\dashv \neg(fxx = \overline{fxx}) && \text{Gen, Abs } y := x \\
&= 1 && \text{TL}
\end{aligned}$$

■

7.7 Quantified Replacement

With Rep one can replace equals with equals provided there is no capture. As one would expect, capture is ok for variables for which the supporting equation universally holds. This can be proven from Ext and Rep.

Proposition 7.26 HL $\vdash (\forall x.\, s = t) \to (f(\lambda x.gs) = f(\lambda x.gt))$

Proof

$(\forall x.\, s = t) \to (f(\lambda x.gs) = f(\lambda x.gt))$	
$= ((\lambda x.s) = \lambda x.t) \to (f(\lambda x.gs) = f(\lambda x.gt))$	Ext
$= ((\lambda x.s) = \lambda x.t) \to (\lambda h.f(\lambda x.g(hx)) = f(\lambda x.gt))\ (\lambda x.s)$	β
$= ((\lambda x.s) = \lambda x.t) \to (\lambda h.f(\lambda x.g(hx)) = f(\lambda x.gt))\ (\lambda x.t)$	Rep
$= ((\lambda x.s) = \lambda x.t) \to (f(\lambda x.gt) = f(\lambda x.gt))$	β
$= 1$	Ref

Note that the β-steps of the proof are redundant since they come for free with a quasi-conversion from Rep. ∎

7.8 Choice and Skolem

The axiom Choice of HL corresponds to the axiom of choice in set theory. So far, we have not made use of Choice. We will now prove the most prominent consequence of Choice, known as Skolem's law.

Proposition 7.27 (Skolem) HL $\vdash (\forall x \exists y.\, fxy) = \exists g \forall x.\, fx(gx)$

Proof By Equiv.

$(\forall x \exists y.\, fxy) \to \exists g \forall x.\, fx(gx)$	
$= (\forall x.\, \exists(fx)) \to \exists g \forall x.\, fx(gx)$	η
$= (\forall x.\, fx(\mathrm{C}(fx))) \to \exists g \forall x.\, fx(gx)$	Choice
$= 1$	Inst $\exists$ with $g := \lambda x.\mathrm{C}(fx)$

$(\exists g \forall x.\, fx(gx)) \to \forall x \exists y.\, fxy$	
$\vdash (\forall x.\, fx(gx)) \to \exists y.\, fxy$	Pull, Gen
$= (\forall x.\, fx(gx)) \wedge fx(gx) \to \exists y.\, fxy$	Inst $\forall$
$= 1$	Inst $\exists$

∎

A basic proposition of set theory says that there is a surjective function $X \to Y$ if and only if there is an injective function $Y \to X$, provided X and Y are not empty. In HOL, we can define surjectivity and injectivity of functions as follows:

$$\begin{aligned} \text{surj } f &:= \forall y \exists x.\, fx = y \\ \text{inj } g &:= \exists f \forall y.\, f(gy) = y \end{aligned}$$

By Skolem's law we have $\text{surj } f = \exists g \forall y.\, f(gy) = y$, which gives us the injective function as required. Hence, given $f : S \to T$ and $g : T \to S$, we have:

$$\begin{aligned} &\text{surj } f \to \exists g.\, \text{inj } g \\ &= (\forall y \exists x.\, fx = y) \to \exists g \exists f \forall y.\, f(gy) = y && \text{Def surj and inj} \\ &= (\exists g \forall y.\, f(gy) = y) \to \exists g \exists f \forall y.\, f(gy) = y && \text{Skolem} \\ &= (\exists g \forall y.\, f(gy) = y) \to \exists f \exists g \forall y.\, f(gy) = y && \text{Com} \\ &= 1 && \text{Inst } \exists \end{aligned}$$

Exercise 7.28 Let $f : S \to T$ and $g : T \to S$. Show that the following formulas are duducible from HL. Do not use Choice or Skolem.

a) $\text{inj } g \to \exists f.\, \text{surj } f$

b) $(\exists g.\, \text{inj } g) = \exists f.\, \text{surj } f$

c) $\text{inj } g \wedge (gx = gy) \to (x = y)$

d) $\text{inj } g = \forall x \forall y.\, (gx = gy) = (x = y)$

Proposition 7.29 $\text{HL} \vdash \text{C}(\lambda x.\, x = y) = y$

Proof

$$\begin{aligned} (\text{C}(\lambda x.\, x = y) = y) &= (\lambda x.\, x = y)(\text{C}(\lambda x.\, x = y) && \beta \\ &= \exists x.\, x = y && \text{Choice} \\ &\dashv y = y && \text{Abs } x := y \\ &= 1 && \text{Ref} \end{aligned}$$

■

Proposition 7.30 $\text{HL} \vdash (\forall x.\, fx = (x = y)) \to (\text{C}f = y)$

Proof

$$\begin{aligned} &(\forall x.\, fx = (x = y)) \to (\text{C}f = y) \\ &= (f = \lambda x.\, x = y) \to (\text{C}f = y) && \text{Ext} \\ &= (f = \lambda x.\, x = y) \to (\text{C}(\lambda x.\, x = y) = y) && \text{Rep} \\ &= (f = \lambda x.\, x = y) \to (y = y) && \text{Proposition 7.29} \\ &= 1 && \text{Ref} \end{aligned}$$

■

Exercise 7.31 Show that the following formulas are duducible from HL.

a) $(\forall x.(x = y) \to fx) = fy$

b) $\exists f \wedge (\forall x.fx \to (x = y)) = \forall x.fx = (x = y)$.

8 Tableaux

Tableaux are a notation for tree-structured backward proofs. They come with a proof construction method useful for humans and computers. Tableaux accommodate branching rules like BCA, CA and Equiv in a uniform manner. They also accommodate hypothetical conversion proofs. Two of the key ideas come from Gentzen's sequent calculus (1935) and are as follows:

- Stepwise decomposition of the initial formula.
- Stepwise expansion of the initial formula by adding formulas obtained by decomposition.

One can have postive and negative tableaux. In the literature one finds only negative tableaux, which are also known as refutation tableaux since their goal consists in showing $\neg s$. We will consider postive tableaux whose goal consists in showing s. Tableaux first appeared in the work of Beth (1959) and Smullyan (1966). They take a prominent role in Melvin Fittings textbook (1996) on first-order logic.

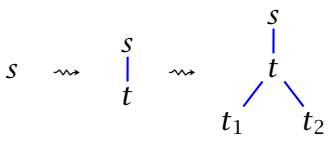
8.1 Hybrid Tableaux

Tableaux are a notation for tree-structured backward proofs. In a tableau, an initial formula is expanded by stepwise adding entailed formulas until a set of tautologous formulas is reached. An expansion step is either unary or binary:

$s \overset{A}{\vdash\!\dashv} s \vee t$ **unary expansion**

$s \overset{A}{\vdash\!\dashv} s \vee t_1,\ s \vee t_2$ **binary expansion**

The construction of a tableau may proceed as follows:



The initial formula is s. It is first expanded with a unary step to $s \vee t$. Then $s \vee t$ is expanded with a binary step to $s \vee t \vee t_1$ and $s \vee t \vee t_2$. The formulas represented by the root-to-leaf paths of a tableau are called the **branches** of the tableau. Here are the sets of branches of the above tableaux:

$$\{s\} \rightsquigarrow \{s \vee t\} \rightsquigarrow \{s \vee t \vee t_1,\ s \vee t \vee t_2\}$$

Given a tableau for an initial formula s whose set of branches is B, we have $s \overset{A}{\vdash\!\dashv} B$. Hence, we have $A \vdash s$ if all branches are tautologous and $A \vdash \mathrm{TL}$.

QC $\dfrac{s}{t}$ $s = t$ is a quasi-conversion from A

QMP $\dfrac{s_1 \ \cdots \ s_n}{t}$ $t \to s_1 \vee \cdots \vee s_n$ is a quasi-instance from A and $n \geq 1$

UI $\dfrac{\forall x.\, s}{s}$ x fresh **EI** $\dfrac{\exists x.\, s}{[x{:=}t]\, s}$

And $\dfrac{s \wedge t}{s \mid t}$ **Equiv** $\dfrac{s =_{\mathbf{B}} t}{\overline{s} \vee t \mid s \vee \overline{t}}$

Rep $\dfrac{\overline{s} \quad [x{:=}s_1]\, t}{[x{:=}s_2]\, t}$ $s \in \{s_1{=}s_2,\ s_2{=}s_1\}$

Figure 20: Hybrid expansion rules

A tableau system comes with a notion of **closedness** for branches. If a branch is closed, it must be tautologous. A tableau is **closed** if all its branches are closed. Closed tableaux represent **complete proofs** and open tableaux represent **partial proofs**.

One can have different tableau systems that differ in their closedness condition and their set of expansion rules. We consider a sytem that we call **hybrid tableaux**. With hybrid tableaux, a branch is closed if and only if it is tautologous. The expansion rules for hybrid tableaux are shown in Figure 20. The expansion rules must not be confused with deduction rules. And and Equiv provide for binary expansions, all other rules for unary expansions. A leaf of a tableau can be expanded with a rule if the premises of the rule appear on the branch to the leaf. The side condition of the rule UI (universal instantiation) states that the branch to which the rule is applied must not contain x.

We will explain the tableau expansion rules together with their correctness proofs. For unary rules we show $s \overset{A}{\vdash\!\dashv} s \vee t$, and for binary rules $s \overset{A}{\vdash\!\dashv} s \vee t_1,\ s \vee t_2$. We could assume $A \vdash \mathrm{HL}$ but prefer to show stronger equivalences if this is possible. The following known facts will be useful:

1. $A \vdash s = t \Longrightarrow s \overset{A}{\vdash\!\dashv} t$ (Weakening)
2. $A \vdash B$ and $s \overset{B}{\vdash\!\dashv} t \Longrightarrow s \overset{A}{\vdash\!\dashv} t$ (Monotonicity)
3. $s \wedge t \overset{\mathrm{TL}}{\vdash\!\dashv} s,\ t$ (And)
4. $\mathrm{HL} \vdash \mathrm{TL}$

The **disjuncts** of a formula are defined recursively:

$$\begin{aligned} \mathcal{D}s &:= \{s\} && \text{if } s \text{ is no disjunction} \\ \mathcal{D}(s \vee t) &:= \mathcal{D}s \cup \mathcal{D}t \end{aligned}$$

Proposition 8.1 (Correctness of QC) Let $A \vdash \mathrm{TL}$, $t \in \mathcal{D}s$, and $t = t'$ be a quasi-conversion from A. Then $A \vdash s = s \vee t'$.

Proof Since $t \in \mathcal{D}s$ we know that $s = s \vee t$ is tautologous. Now the claim follows by quasi-conversion:

$$\begin{aligned} s &= s \vee t && \mathrm{TL} \\ &= s \vee t' && A \end{aligned}$$

■

Proposition 8.2 (Correctness of QMP) Let $A \vdash \mathrm{TL}$, $\mathcal{D}t \subseteq \mathcal{D}s$, and $t' \to t$ be a quasi-instance from A. Then $A \vdash s = s \vee t'$.

Proof Since $\mathcal{D}t \subseteq \mathcal{D}s$ we know that $s = s \vee t$ is tautologous. Moreover, $t = t \vee t'$ is a quasi-conversion from A since $t' \to t$ is a quasi-instance from A. Now the claim follows by quasi-conversion:

$$\begin{aligned} s &= s \vee t && \mathrm{TL} \\ &= s \vee t \vee t' && A \\ &= s \vee t' && \mathrm{TL} \end{aligned}$$

■

Proposition 8.3 (Correctness of EI) Let $(\exists x.t) \in \mathcal{D}s$. Then $\mathrm{HL} \vdash s = s \vee [x{:=}t']t$.

Proof Follows with QMP since $[x{:=}t']t \to \exists x.t$ is a quasi-instance of $fx \to \exists f$ (the instantiation law for $\exists$). ■

We will use the following notation for negated equations:

$$s \neq t \;:=\; \overline{s = t}$$

Proposition 8.4 (Correctness of Rep)
Let $\{s_1{\neq}s_2, [x{:=}s_1]\,t\} \subseteq \mathcal{D}s$ or $\{s_2{\neq}s_1, [x{:=}s_1]\,t\} \subseteq \mathcal{D}s$. Then $s \overset{\mathrm{HL}}{\vdash\!\dashv} s \vee [x{:=}s_2]\,t$.

Proof Follows with QMP since the formulas $[x{:=}s_2]\,t \to s_1{\neq}s_2 \vee [x{:=}s_1]\,t$ and $[x{:=}s_2]\,t \to s_2{\neq}s_1 \vee [x{:=}s_1]\,t$ are quasi-instances of $(x{=}y) \to fx = (x{=}y) \to fy$ (the replacement axiom of HL). This follows since $(x \to y = x \to z) \to z \to \overline{x} \vee y$ and $(x \to y = x \to z) \to y \to \overline{x} \vee z$ are tautologies. ■

Proposition 8.5 (Correctness of UI) Let $(\forall x.t) \in \mathcal{D}s$ and $x \notin \mathcal{N}s$.
Then $s \overset{\text{HL}}{\vdash} s \vee t$.

Proof Since $(\forall x.t) \in \mathcal{D}s$, we know that $s = s \vee (\forall x.t)$ is tautologous. Hence:

$s = s \vee (\forall x.t)$	TL
$= \forall x.s \vee t$	Pull, x $\notin \mathcal{N}$s
$\vdash s \vee t$	Gen

■

Proposition 8.6 (Correctness of And) Let $(t_1 \wedge t_2) \in \mathcal{D}s$. Then $s \overset{\text{TL}}{\vdash} s \vee t_1, s \vee t_2$.

Proof Since $(t_1 \wedge t_2) \in \mathcal{D}s$, we know that $s = s \vee (t_1 \wedge t_2)$ is tautologous. Hence:

$s = s \vee (t_1 \wedge t_2)$	TL
$= (s \vee t_1) \wedge (s \vee t_2)$	TL

Now the claim follows with Weakening and And. ■

Proposition 8.7 (Correctness of Equiv) Let $(t_1 =_{\mathbf{B}} t_2) \in \mathcal{D}s$.
Then $s \overset{\text{TL}}{\vdash} s \vee \overline{t_1} \vee t_2, s \vee t_1 \vee \overline{t_2}$.

Proof Since $(t_1 =_{\mathbf{B}} t_2) \in \mathcal{D}s$, we know that $s = (s \vee \overline{t_1} \vee t_2) \wedge (s \vee t_1 \vee \overline{t_2})$ is tautologous. Now the claim follows with Weakening and And. ■

8.2 Hybrid Tableau Proofs

We now consider several examples of (hybrid) tableau proofs. A tableau proof is a closed tableau that is validated by the expansion rules and is annotated so that one can see which axioms are used with quasi-conversion and the quasi-modus ponens steps.

Example 8.8 Here is a tableau proof of HL $\vdash (x = y) \to (y = z) \to (x = z)$.

1. $(x = y) \to (y = z) \to (x = z)$	
2. $x \neq y,\ y \neq z,\ x = z$	TL
3. $y = z$	Rep

Every line represents a formula. The commas in the second line represent disjunctions. This notational device helps to reduce the number of parentheses. Line (2) is obtained from line (1) by a quasi-conversion from TL. Line (3) is obtained with Rep from the disjuncts $x \neq y$ and $x = z$ in line (2). Since the single branch of the tableau contains a complementary pair of formulas ($y \neq z$ and $y = z$), the tableau is closed. □

Example 8.9 Here is a tableau proof that uses the expansion rule Equiv.

$$(x = y) = (y = x)$$

$x \neq y,\ y = x$	Equiv	$x = y,\ y \neq x$	Equiv
$y = y$	Rep	$x = x$	Rep
1	Ref	1	Ref

□

Example 8.10 (BCA) Here is a tableau proof of the axiom BCA of PL.

1. $f0 \to f1 \to fx$
2. $\overline{f0},\ \overline{f1},\ fx$ TL
3. $x \neq 0 \wedge x \neq 1$ QMP TL

4. $x \neq 0$	And	4. $x \neq 1$	And
5. $f0$	Rep of (4), (2)	5. $f1$	Rep of (4), (2)

Formula (2) of the proof is obtained from formula (1) by QC. Since $(1) = (2)$ tautologous, it does not matter which assumption we use. We annotate such step with TL. Formula (3) is introduced with QMP. Since $\overline{(3)}$ is a tautology, it does not matter which assumption we use. We annotate such step with QMP TL. □

Example 8.11 (Double Instantiation) Here is a linear tableau proof that uses the instantiation rules for the quantifiers. The types are as follows: $f : T \to T \to \mathbf{B}$, $h : (T \to \mathbf{B}) \to T$, $g : T \to \mathbf{B}$.

1. $\exists g \forall x \exists h.\ h(fx) \wedge \overline{hg}$
2. $\forall x \exists h.\ h(fx) \wedge \overline{h(\lambda x.\overline{fxx})}$ EI with $g := \lambda x.\overline{fxx}$
3. $\exists h.\ h(fx) \wedge \overline{h(\lambda x.\overline{fxx})}$ UI
4. fxx EI of (3) with $h := \lambda g.gx$, TL
5. $\overline{fxx}$ EI of (3) with $h := \lambda g.\overline{gx}$, TL

Note that the existential formula (3) is instantiated twice and that the instantiation steps (4) and (5) exploit the power of quasi-conversion and simplify $fxx \wedge fxx$ and $\overline{fxx} \wedge \overline{fxx}$ to fxx and $\overline{fxx}$. □

Example 8.12 (Ping-Pong)

1.	$(\exists f \forall y.\ f(gy) = y) \to \exists f \forall y \exists x.\ fx = y$	
2.	$\overline{\exists f \forall y.\ f(gy) = y},\ \exists f \forall y \exists x.\ fx = y$	TL
3.	$\forall f \exists y.\ f(gy) \neq y$	de Morgan twice
4.	$\exists y.\ f(gy) \neq y$	UI
5.	$\forall y \exists x.\ fx = y$	EI of (2)
6.	$\exists x.\ fx = y$	UI
7.	$f(gy) = y$	EI with $x := gy$
8.	$f(gy) \neq y$	EI of (4)

Note that the universal instantiation (4) and (6) must precede the existential instantiations (5) and (7) so that f and y are fresh for the universal instantiations.□

8.3 First-order Tableaux

We now consider a simpler tableaux system that comes without quasi-conversion and quasi-modus ponens and makes no use of assumptions (i.e., $A = \emptyset$). A branch s of a first-order tableaux is **closed** if $1 \in \mathcal{D}s$ or $\exists t \in \mathcal{D}s : \overline{t} \in \mathcal{D}s$ (i.e., s contains two complementary disjuncts). The expansion rules for first-order tableaux are shown in Figure 21. As it will turn out, first-order tableaux are complete for classical first-order formulas. That is, for any classical first-order formula that is deducible from HL without Choice, there is a first-order tableau proof.

The rules exhibit some remarkable properties. For every logical constant there are at most two expansion rules. The overlined rules are obtained with the de Morgan laws and the non-overlined rules. If there are two rules for a logical constant that is not an indentity, one of the rules if overlined. If we construct a proof for a formula, we need only the rules for the logical constants that occur in the formula. In particular, if we construct proofs for tautologous formulas, we don't need the rules for quantifiers and identities.

The propositional rules yield a descision algorithm for propositial formulas. First note that they only add subformulas of existing formulas to a branch, possibly in negated form. Hence, after finitely many steps, no new formulas can be added to a branch. We call such branches **saturated**. A tableau is **complete** if each of its branches is either closed or saturated. According to what we have said, we can construct a complete tableau for every propositional formula. If the tableau is closed, the formula is deducible and hence a tautology. Otherwise the tableau contains a saturated branch that is not closed. One can show that a saturated branch is a tautology if and only if it is closed (Hintikka's Lemma).

$$\overline{\textbf{Zero}}\ \frac{\overline{0}}{1}$$

$$\overline{\textbf{Not}}\ \frac{\overline{\overline{s}}}{s}$$

$$\overline{\textbf{Or}}\ \frac{\overline{s \vee t}}{\overline{s} \mid \overline{t}}$$

$$\textbf{And}\ \frac{s \wedge t}{s \mid t} \qquad \overline{\textbf{And}}\ \frac{\overline{s \wedge t}}{\overline{s} \vee \overline{t}}$$

$$\textbf{Imp}\ \frac{s \to t}{\overline{s} \vee t} \qquad \overline{\textbf{Imp}}\ \frac{\overline{s \to t}}{s \mid \overline{t}}$$

$$\textbf{Equiv}\ \frac{s =_{\mathbf{B}} t}{\overline{s} \vee t \mid s \vee \overline{t}} \qquad \overline{\textbf{Equiv}}\ \frac{\overline{s =_{\mathbf{B}} t}}{s \vee t \mid \overline{s} \vee \overline{t}}$$

$$\textbf{UI}\ \frac{\forall x.\, s}{s}\ x \text{ fresh} \qquad \overline{\textbf{UI}}\ \frac{\overline{\forall x.\, s}}{\overline{[x{:=}t]\, s}}$$

$$\textbf{EI}\ \frac{\exists x.\, s}{[x{:=}t]\, s} \qquad \overline{\textbf{EI}}\ \frac{\overline{\exists x.\, s}}{\overline{s}}$$

$$\textbf{Ref}\ \frac{s = s}{1} \qquad \textbf{Rep}\ \frac{\overline{s} \quad [x{:=}s_1]\, t}{[x{:=}s_2]\, t}\ s \in \{s_1{=}s_2,\, s_2{=}s_1\}$$

Figure 21: First-order expansion rules

Since a formula s is a tautology if and only if every branch of a tableau for s is a tautology, the existence of a complete but non-closed tableau implies that the formula is not a tautology.

9 Prime Trees and BDDs

In this chapter we consider a special class of propositional formulas, called prime trees, such that every propositional formula is deductively equivalent to exactly one prime tree (with respect to TL). In other words, prime trees are unique normal forms for propositional formulas. We also look at an efficient graph representation for prime trees known as BDDs (binary decision diagrams). Every node of a BDD represents a prime tree.

Prime trees are a special form of decision trees. Decision trees have been in the literature for a long time. But it took until 1986 that Bryant discovered a class of decision trees that yield unique representations of propositional formulas (i.e., the class of prime trees). The BDD implementation of prime trees is of great practical importance for the computer-aided design of circuits and for the verification of finite transition systems (i.e., model checking).

9.1 Prime Trees

In this chapter we call two formulas s, t **equivalent** if $s = t$ is tautologous. We will use the following notation:

$$(s, t_0, t_1) \;:=\; \overline{s} \wedge t_0 \vee s \wedge t_1 \qquad \textbf{conditional}$$

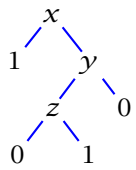
Proposition 9.1 The following formulas are tautologies:

1. $(0, x, y) = x$
2. $(1, x, y) = y$
3. $(x, y, y) = y$

We use **PF** to denote the set of all propositional formulas. The set **DT** $\subseteq$ PF of **decision trees** is defined recursively:

1. 0 and 1 are decision trees.
2. (x, s, t) is a decision tree if s and t are decision trees and $x : \mathbf{B}$ is a variable.

As the name suggests, decision trees can be thought of as trees. For instance, $(x, 1, (y, (z, 0, 1), 0))$ may be seen as the tree



A decision tree is **reduced** if none of its subtrees has the form (x, s, s). Formally, we can define reduced decision trees recursively:

1. 0 and 1 are reduced decision trees.
2. (x, s, t) is a reduced decision tree if s and t are different reduced decision trees and $x : \mathbf{B}$ is a variable.

By Proposition 9.1(3) we can compute for every decision tree an equivalent reduced decision tree.

In the following we assume a linear order on the set of all variables of type **B**. We write $x < y$ if x is smaller than y with respect to this order. The notion of prime time is defined with respect to this order.

A decision tree is **ordered** if the variables get larger as one goes down on a path from the root to a leaf. For example, the example tree drawn above is ordered if and only if $x < y < z$. Formally, we define ordered decision trees recursively:

1. 0 and 1 are ordered decision trees.
2. (x, s, t) is a ordered decision tree if s and t are ordered decision trees and x is smaller than every variable that occurs in s or t.

A **prime tree** is a reduced and ordered decision tree.

Proposition 9.2 For every propositional formula s there exists an equivalent prime tree containing only variables that occur in s.

Proof By induction on the number of variables occurring in s.

Case 1 Let s be closed. Then s is equivalent to 0 or 1 by Proposition 5.14 (0-1).

Case 2 Let x be the least variable occurring in s. By Boolean Expansion we know that s is equivalent to $(x, [x{:=}0]s, [x{:=}1]s)$. By induction there exist prime trees s_0 and s_1 equivalent to $[x{:=}0]s$ and $[x{:=}1]s$ that contain only variables larger than x. We know that s is equivalent to (x, s_0, s_1). If $s_0 \neq s_1$, then (x, s_0, s_1) is a prime tree. Otherwise, s is equivalent to the prime tree s_0. ∎

Next we will show that there is at most one prime tree equivalent to a given propositional formula. Recall that $\mathcal{T}$ denotes the unique model of PL.

Proposition 9.3 $\mathcal{I} \supset \mathcal{T} \implies \hat{\mathcal{I}}(x, s, t) =$ if $\mathcal{I}x = 0$ then $\hat{\mathcal{I}}s$ else $\hat{\mathcal{I}}t$

Lemma 9.4 For all prime trees s, t: $s \neq t \implies \exists \mathcal{I} \supset \mathcal{T} : \hat{\mathcal{I}}s \neq \hat{\mathcal{I}}t$.

Proof By induction on $|s| + |t|$. Let s, t be different prime trees. We show that there is an interpretation $\mathcal{I} \supset \mathcal{T}$ such that $\hat{\mathcal{I}}s \neq \hat{\mathcal{I}}t$.

Case $s, t \in \{0, 1\}$. Every $\mathcal{I} \supset \mathcal{T}$ does the job.

Case $s = (x, s_0, s_1)$ and $x \notin \mathcal{N}t$. By induction we have $\mathcal{I} \supset \mathcal{T}$ such that $\hat{\mathcal{I}}s_0 \neq \hat{\mathcal{I}}s_1$. Since x occurs neither in s_0 nor s_1, we have $\hat{\mathcal{I}}_{x,0}s \neq \hat{\mathcal{I}}_{x,1}s$ since $\hat{\mathcal{I}}_{x,0}s = \hat{\mathcal{I}}_{x,0}s_0 =$

$\hat{\mathcal{I}}s_0 \neq \hat{\mathcal{I}}s_1 = \hat{\mathcal{I}}_{x,1}s_1 = \hat{\mathcal{I}}_{x,1}s$. But $\hat{\mathcal{I}}_{x,0}t = \hat{\mathcal{I}}_{x,1}t$ since x does not occur in t. Hence $\hat{\mathcal{I}}_{x,0}s \neq \hat{\mathcal{I}}_{x,0}t$ or $\hat{\mathcal{I}}_{x,1}s \neq \hat{\mathcal{I}}_{x,1}t$.

Case $t = (x, t_0, t_1)$ and $x \notin \mathcal{N}s$. Analogous to previous case.

Case $s = (x, s_0, s_1)$ and $t = (x, t_0, t_1)$. Then $s_0 \neq t_0$ or $s_1 \neq t_1$. By induction there is an $\mathcal{I} \supset \mathcal{T}$ such that $\hat{\mathcal{I}}s_0 \neq \hat{\mathcal{I}}t_0$ or $\hat{\mathcal{I}}s_1 \neq \hat{\mathcal{I}}t_1$. By coincidence $\hat{\mathcal{I}}_{x,0}s_0 \neq \hat{\mathcal{I}}_{x,0}t_0$ or $\hat{\mathcal{I}}_{x,1}s_1 \neq \hat{\mathcal{I}}_{x,1}t_1$. Hence $\hat{\mathcal{I}}_{x,0}s \neq \hat{\mathcal{I}}_{x,0}t$ or $\hat{\mathcal{I}}_{x,1}s \neq \hat{\mathcal{I}}_{x,1}t$.

To see that the case analysis is exhaustive, consider that case that 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 9.5 (Prime Tree) For every propositional formula there exists exactly one equivalent prime tree.

Proof The existence follows with Proposition 9.2. The uniqueness follows with Lemma 9.4 and the fact that for equivalent prime trees there cannot be an interpretation $\mathcal{I} \supset \mathcal{T}$ that evaluates them differently (since $\mathcal{T}$ is a model of HL). ∎

For every propositional formula s we denote the unique prime tree equivalent to s with πs. We call πs the **prime tree for** s.

Proposition 9.6 Let s and t be propositional formulas.

1. s, t are equivalent if and only if $\pi s = \pi t$.
2. $s = t$ is a tautology if and only if $\pi s = \pi t$.
3. s is a tautology if and only if $\pi s = 1$.
4. $\mathcal{N}(\pi s) \cap Var \subseteq \mathcal{N}s$.

The **significant variables** of a propositional formula are the variables occurring in its prime tree:

$$\mathrm{SV}\, s \;:=\; \mathcal{N}(\pi s) \cap Var$$

Proposition 9.7 For every propositional formula s:

1. $\mathrm{SV}\, s \subseteq \mathcal{N}s$
2. A variable x is significant for s if and only if there exists an interpretation $\mathcal{I} \supset \mathcal{T}$ and a value $v \in \mathcal{I}(\tau x)$ such that $\hat{\mathcal{I}}s \neq \hat{\mathcal{I}}_{x,v}s$.

9.2 Algorithms

Next we consider algorithms for the Boolean operations on prime trees. We will develop algorithms for negation and conjunction. The algorithms for the other operations can be constructed along the same lines.

The algorithms for negation and conjunction compute the functions

$$\begin{array}{ll} \mathit{not} \in \mathrm{PT} \to \mathrm{PT} & \mathit{and} \in \mathrm{PT} \to \mathrm{PT} \to \mathrm{PT} \\ \mathit{not}\, s = \pi(\overline{s}) & \mathit{and}\, s\, t = \pi(s \wedge t) \end{array}$$

As it turns out, the algorithm can be derived elegantly together with their correctness proofs.

We base the algorithm for negation on the tautologies (verify!)

$$\begin{aligned} \overline{0} &= 1 \\ \overline{1} &= 0 \\ \overline{(x,y,z)} &= (x,\overline{y},\overline{z}) \end{aligned}$$

With these tautologies one can verify the equations

$$\begin{aligned} \pi(\overline{0}) &= 1 \\ \pi(\overline{1}) &= 0 \\ \pi(\overline{(x,s,t)}) &= (x,\pi(\overline{s}),\pi(\overline{t})) \qquad \text{if } (x,s,t) \text{ is a prime tree} \end{aligned}$$

The correctness of the first two equations is obvious. For the corectness of the last equation we show 2 things:

1. The formula on the left is equivalent to the formula on the right. To verify this, we can erase all applications of π since π always yields equivalent formulas. Now we are left with an instance of the third tautology.
2. The formula on the right is a prime tree. Let (x,s,t) be a prime tree. Then s and t are not equivalent. Hence $\overline{s}$ and $\overline{t}$ are not equivalent (by contradiction, double negation). Hence $\pi(\overline{s})$ and $\pi(\overline{t})$ are different prime trees. Since $\pi(\overline{s})$ and $\pi(\overline{t})$ contain only variables that are in s and t and (x,s,t) is a prime tree, $(x,\pi(\overline{s}),\pi(\overline{t}))$ is a prime tree.

Together, the 2 properties yield the correctness of the equation since for every formula there is only one equivalent prime tree.

Now we have the following procedure:

$$\begin{aligned} &\mathit{not} : \mathrm{PT} \to \mathrm{PT} \\ &\quad \mathit{not}\ 0 = 1 \\ &\quad \mathit{not}\ 1 = 0 \\ &\mathit{not}(x,s,t) = (x,\ \mathit{not}\ s,\ \mathit{not}\ t) \end{aligned}$$

The equation are exhaustive and terminating. Moreover, they are valid for the function *not*, since they reduce to the above equations with π by unfolding the definition of the function not. Hence the procedure *not* computes the function *not*.

Next we devise the algorithm for conjunction. This time we employ the tautologies (verify!)

$$\begin{aligned} 0 \wedge y &= 0 \\ 1 \wedge y &= y \\ (x, y, z) \wedge (x, y', z') &= (x, y \wedge y', z \wedge z') \\ (x, y, z) \wedge u &= (x, y \wedge u, z \wedge u) \end{aligned}$$

and the commutativity of $\wedge$. We also use an auxiliary function

$$\begin{aligned} &red \in \mathrm{DT} \to \mathrm{DT} \\ &red\ 0 = 0 \\ &red\ 1 = 1 \\ &red(x, s, t) = \text{if } s = t \text{ then } s \text{ else } (x, s, t) \end{aligned}$$

Now we can verify the following equations:

$$\begin{aligned} \pi(0 \wedge t) &= 0 \\ \pi(1 \wedge t) &= t \\ \pi((x, s_0, s_1) \wedge (x, t_0, t_1)) &= red(x, \pi(s_0 \wedge t_0), \pi(s_1 \wedge t_1)) \\ \pi((x, s_0, s_1) \wedge t) &= red(x, \pi(s_0 \wedge t), \pi(s_1 \wedge t)) \\ &\qquad \text{if } t = (y, t_0, t_1) \text{ and } x < y \end{aligned}$$

As for negation, the correctness of the equations is established in 2 steps. First one verifies that for each equations the formula on the left is equivalent to the formula on the right. Since π and *red* yield equivalent formulas, we can erase their applications. Now we are left with instances of the above tautologies. For the second step we show that the formulas on the right are prime trees, provided the arguments on the left hand side are prime trees. This is easy and explains the condition $x < y$ coming with the last equation.

Now we have the following procedure:

$$
\begin{aligned}
&\mathit{and} : \mathrm{PT} \to \mathrm{PT} \to \mathrm{PT} \\
&\qquad\qquad \mathit{and}\ 0\ t = 0 \\
&\qquad\qquad \mathit{and}\ 1\ t = t \\
&\qquad\qquad \mathit{and}\ s\ 0 = 0 \\
&\qquad\qquad \mathit{and}\ s\ 1 = s \\
&\mathit{and}\ (x, s_0, s_1)\ (x, t_0, t_1) = \mathit{red}(x,\ \mathit{and}\ s_0\ t_0,\ \mathit{and}\ s_1\ t_1) \\
&\qquad \mathit{and}\ (x, s_0, s_1)\ t = \mathit{red}(x,\ \mathit{and}\ s_0\ t,\ \mathit{and}\ s_1\ t) \\
&\qquad\qquad\qquad\qquad \text{if } t = (y, t_0, t_1) \text{ and } x < y \\
&\qquad \mathit{and}\ s\ (y, t_0, t_1) = \mathit{red}(y,\ \mathit{and}\ s\ t_0,\ \mathit{and}\ s\ t_1) \\
&\qquad\qquad\qquad\qquad \text{if } s = (x, s_0, s_1) \text{ and } x > y
\end{aligned}
$$

The procedure *and* computes the function *and* since the following properties are satisfied:

1. The equations are exhaustive and terminating.
2. The equations hold for the function *and*. This is the case since the equations reduce to the above tautologies (up to commutativity) using the definitions of the functions *and* and *red*.

You now know enough so that you can devise algorithms for the other Boolean operations. Things are as before since for every Boolean operation $\circ$ the following formulas are tautologies:

$$
\begin{aligned}
(x, y, z) \circ (x, y', z') &= (x, y \circ y', z \circ z') \\
u \circ (x, y', z') &= (x, u \circ y', u \circ z') \\
(x, y, z) \circ u &= (x, y \circ u, z \circ u)
\end{aligned}
$$

This follows with BCA on x and the tautologies of Proposition 9.1.

9.3 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 22 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**).

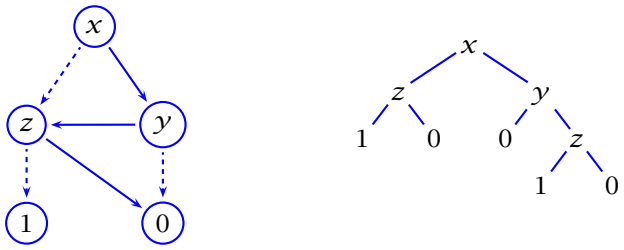


Figure 22: A BDD and the decision tree represented by the topmost node

In our example BDD the node labeled with z represents a subtree that occurs twice in the decision tree on the right.

Formally, a **BDD** is a function γ such that there exists a natural number $N \geq 1$ such that

1. $\gamma \in \{2,\ldots,N\} \to \mathit{Var} \times \{0,\ldots,N\} \times \{0,\ldots,N\}$.
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 0 and 1. 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 22 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

$$\tau_\gamma \in \{0,1\} \cup \mathit{Dom}\,\gamma \to \mathit{DT}$$

$$\tau_\gamma 0 = 0$$

$$\tau_\gamma 1 = 1$$

$$\tau_\gamma n = (x, \tau_\gamma n_0, \tau_\gamma n_1) \quad \text{if } \gamma n = (x,n_0,n_1)$$

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 22 is minimal.

Theorem 9.8 (Minimality) A BDD is minimal if and only if it 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.

You will find useful information in the English Wikipedia entry for binary decision diagram.

10 First-Order Propositional Completeness

In this chapter we consider finite sets of tautologies that are deductively equivalent to the set of all tautologies. To obtain such a set, one can either start from the axioms for Boolean algebras or from the implication-based systems pioneered by Frege (often called Hilbert systems). We will start from the axioms for Boolean algebras. For the completeness proof we have again two possibilities. Either we base it on the expansion law (a first-order version of BExp) or on a reduction to conjunctive normal form. We choose the expansion law.

10.1 BC and BC'

Recall the definition of propositional formulas and tautologies in § 5.2. We know $\mathrm{PL} \vdash \mathrm{TL} \cup \{\mathrm{BCA}\}$. PL has exactly one model. We call this model $\mathcal{T}$. The structure $\mathcal{T}$ interprets the names for the Boolean primitives in the canonical way.

Proposition 10.1 For every propositional formula s the following statements are equivalent:

1. s is a tautology.
2. s is valid in $\mathcal{T}$.
3. $s \in \mathrm{TL}$.
4. s is deducible from TL.
5. s is deducible from PL.
6. s is deducible from HL.

Proof Follows from the results in § 5.2. ∎

Figure 23 shows two specifications BC and BC′. BC is similar to BA in Figure 4 but uses the sort **B** instead of D as well as other names. This makes a significant difference semantically since the interpretation of **B** is fixed. BC′ extends BC with the axioms Impl and Equiv. Impl introduces $\rightarrow$ as defined constant. Equiv expresses a prominent property of Boolean identity that is not subsumed by the other axioms (as we will see soon).

Every axiom of BC′ is a tautology. Hence $\mathrm{BC}' \subseteq \mathrm{TL}$ and $\mathcal{T}$ is a model of BC′. We will show that BC has two models and that $\mathcal{T}$ is the only model of BC′.

In the conversion proofs to come we will make tacit use of the axioms Commutativity and Associativity.

Proposition 10.2 The following formulas are deducible from BC.

$x \wedge x = x$	$x \vee x = x$	Idempotence
$x \wedge 0 = 0$	$x \vee 1 = 1$	Dominance

Specification	BC	
Constants	$0, 1 : \mathbf{B}$ $\neg : \mathbf{B} \to \mathbf{B}$ $\wedge, \vee : \mathbf{B} \to \mathbf{B} \to \mathbf{B}$	
Notation	$\overline{x} := \neg x$	
Axioms	$x \wedge y = y \wedge x$ $x \vee y = y \vee x$	Commutativity
	$(x \wedge y) \wedge z = x \wedge (y \wedge z)$ $(x \vee y) \vee z = x \vee (y \vee z)$	Associativity
	$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$ $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$	Distributivity
	$x \wedge 1 = x$ $x \vee 0 = x$	Identity
	$x \wedge \overline{x} = 0$ $x \vee \overline{x} = 1$	Complement
Specification	BC′	
Extends	BC	
Defined Constant	$x \to y = \overline{x} \vee y$	Impl
Axiom	$(x = y) = (x \to y) \wedge (y \to x)$	Equiv

Figure 23: Specifications BC and BC′

Proposition 10.3 $\mathrm{BC} \cup \{0 = 1\} \vdash x = y$

Proof Follows with Identity and Dominance. ∎

Proposition 10.4 For every model $\mathcal{A}$ of BC: $\mathcal{A}0 \neq \mathcal{A}1$.

Proof Immediate consequence of the preceding proposition since **B** must be interpreted as the 2-element set $\mathbb{B}$. ∎

The next proposition will be very useful for showing that tautologies with negations are deducible from BC.

Proposition 10.5 (UoC: Uniqueness of Complements) For all formulas s, t:
$s = \overline{t} \overset{\mathrm{BC}}{\dashv\vdash} s \wedge t = 0,\ s \vee t = 1$

Proof The direction from left to right follows with Complement. The other direction can be shown with the following conversion proof:

$s = s \wedge 1$	Identity
$= s \wedge (t \vee \overline{t})$	Complement
$= s \wedge t \vee s \wedge \overline{t}$	Distributivity
$= 0 \vee s \wedge \overline{t}$	$s \wedge t = 0$
$= t \wedge \overline{t} \vee s \wedge \overline{t}$	Complement
$= (t \vee s) \wedge \overline{t}$	Distributivity
$= 1 \wedge \overline{t}$	$s \vee t = 1$
$= \overline{t}$	Identity

■

Proposition 10.6 The following formulas are deducible from BC.

$\overline{x \wedge y} = \overline{x} \vee \overline{y}$	$\overline{x \vee y} = \overline{x} \wedge \overline{y}$	deMorgan
$\overline{1} = 0$	$\overline{0} = 1$	Negation
$\overline{\overline{x}} = x$		Double Negation

Proof Follows with UoC. ■

Proposition 10.7 BC has exactly two models. One interprets the constants of BC as the structure $\mathcal{T}$ does, the other in the dual way: 0 as 1, 1 as 0, $\neg$ as negation, $\wedge$ as disjunction, and $\vee$ as conjunction.

Proof By Proposition 10.4 we know that every model $\mathcal{A}$ of BC must satisfy either $\mathcal{A}0 = 0$, $\mathcal{A}1 = 1$ or $\mathcal{A}0 = 1$, $\mathcal{A}1 = 0$. By Identity and Dominance we know that there is no choice for $\wedge$ and $\vee$ once the interpretation of 0 and 1 is fixed. By Negation we know the same for $\neg$. ■

A **Boolean formula** is a propositional formula that does not contain the constants $\rightarrow$ and $=$.

Proposition 10.8 (0-1)
For every closed Boolean formula s: $\mathrm{BC} \vdash s{=}0$ or $\mathrm{BC} \vdash s{=}1$.

Proof By induction on $|s|$. If $s = 0$ or $s = 1$, the claim follows by Ref. If $s = \neg t$, we use induction for t and Negation. If $s = s_1 \wedge s_2$ or $s = s_1 \vee s_2$, we use induction for s_1 and s_2 and then Identity or Dominance. ■

Proposition 10.9 For every propositional formula s there exists a Boolean formula t such that $\mathrm{BC}' \vdash s = t$.

Proof The constants $\rightarrow$ and $=$ can be eliminated with the axioms Impl and Equiv. ∎

Proposition 10.10 (E1) $(x = 1) = x$ is deducible from BC′.

Proposition 10.11 $\mathcal{T}$ is the only model of BC′.

Proof By Proposition 10.10 we know that $(1 = 1) = 1$ is deducible from BC′. Hence every model of BC′ must interpret 1 as 1 since the interpretation of the identies is fixed. Now the claim follows with Proposition 10.7 and the fact that $\mathcal{T}$ is a model of BC′. ∎

10.2 Expansion and Completeness

Lemma 10.12 Let s be a Boolean formula and x a name of type **B**. Then:
$\mathrm{BC} \vdash x \wedge s = x \wedge [x{:=}1]s$

Proof By induction on $|s|$. ∎

Lemma 10.13 Let s be a Boolean formula and x a name of type **B**. Then:
$\mathrm{BC} \vdash \overline{x} \wedge s = \overline{x} \wedge [x{:=}0]s$

Proof By induction on $|s|$. ∎

Proposition 10.14 (Expansion) Let s be a Boolean formula and x a name of type **B**. Then: $\mathrm{BC} \vdash s = \overline{x} \wedge [x{:=}0]s \vee x \wedge [x{:=}1]s$

A **Boolean tautology** is a Boolean formula that is a tautology.

Proposition 10.15 (Boolean Completeness) For every Boolean tautology s:
$\mathrm{BC} \vdash s = 1$.

Proof By induction on the number of names of type **B** occurring in s. ∎

Proposition 10.16 (Completeness) $\mathrm{TL} \dashv\vdash \mathrm{BC}'$.