1 Mathematical Formulas as Terms

Our first concern is the logical structure of mathematical formulas. We will represent formulas in the *simply typed lambda calculus*, a system that provides an abstract syntax for functions. Here are examples of mathematical formulas:

$$
\begin{array}{l}
2y + z \\
(\neg y \Rightarrow \neg x) \Rightarrow (x \Rightarrow y) \\
x \wedge (x \Rightarrow y) = x \wedge y \\
f0 \wedge (\forall x{\in}\mathbb{N}\colon fx \Rightarrow f(x+1)) \Rightarrow \forall x{\in}\mathbb{N}\colon fx \\
(\forall x{\in}X\colon fx \wedge gx) = (\forall x{\in}X\colon fx) \wedge (\forall x{\in}X\colon gx)
\end{array}
$$

1.1 Boolean Operations

Formulas often employ Boolean operations such as conjunction ($\wedge$) and implication ($\Rightarrow$). We use the numbers 0 and 1 as **Boolean values,** where 0 may be thought of as "false" and 1 as "true". We define

$$\mathbb{B} \overset{\text{def}}{=} \{0,1\}$$

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

$$
\begin{array}{l}
(3 < 7) = 1 \\
(7 \le 3) = 0 \\
(3 = 7) = 0
\end{array}
$$

The Boolean operations are defined as follows:

$$
\begin{array}{rcl}
\neg x &=& 1 - x \\
x \wedge y &=& \min\{x,y\} \\
x \vee y &=& \max\{x,y\} \\
x \Rightarrow y &=& \neg x \vee y \\
x \Leftrightarrow y &=& (x = y)
\end{array}
$$

1.2 Functions and Lambda Notation

In our analysis of mathematical language, functions will play the main role.

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 $(x, y) \in f$.

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

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

This notation describes the function $\mathbb{Z} \to \mathbb{Z}$ that doubles its argument (i.e., yields $2x$ for x).

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

$$\mathit{plus} \overset{\text{def}}{=} \lambda x{\in}\mathbb{Z}.\ \lambda y{\in}\mathbb{Z}.\ x + y$$

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 the sum $a + b$ 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 American logician Haskell Curry. They first appeared 1924 in a paper by Moses Schönfinkel on the primitives of mathematical language.

For convenience, we omit parentheses as follows:

$$\begin{array}{rcl} t_1\, t_2\, t_3 & \rightsquigarrow & (t_1\, t_2)\, t_3 \\ T_1 \to T_2 \to T_3 & \rightsquigarrow & T_1 \to (T_2 \to T_3) \end{array}$$

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

1.3 Terms

The syntax of formulas can be described at different levels of abstraction. While a **concrete syntax** is concerned with notational aspects, an **abstract syntax** is concerned with logical structure and abstains from notational aspects. It turns out that abstract syntax is the right base for talking about the meaning of formulas (i.e., semantics) and for designing algorithms manipulating formulas.

We will represent the abstract syntax of mathematical formulas with terms. There are 4 different forms of terms whose construction can be summarized with the grammar

$$t \ ::=\ c \ \mid\ x \ \mid\ t\,t \ \mid\ \lambda x{:}T.t$$

The grammar says that a term is either a **constant** c, a **variable** x, an **application** $t_1\,t_2$ formed with two term t_1 and t_2, or an **abstraction** $\lambda x{:}T.t$ formed with a variable x, a type T and a term t. Every value can be used as a constant c. Variables can be thought of as names. **Types** are formed according to the grammar

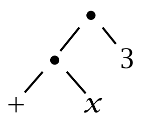
$$T \;::=\; C \;\mid\; T \to T$$

where every nonempty set can be used as a **base type** C.

Terms and types should be thought of as mathematical objects. It is helpful to see terms as the objects of an abstract data structure. Eventually, we will give a formal account of terms and types and also consider their implementation in the prgramming language Standard ML.

We will see soon that terms provide an abstract syntax for mathematical formulas. In fact, terms provide a universal abstract syntax. For instance, programs can be represented as terms and many compilers are using a term representation.

From the grammar it is clear that terms are tree-structured objects. It is instructive to describe terms with trees. The tree

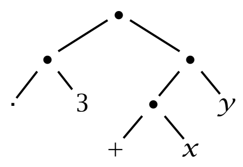


represents a term, that consists of two applications represented as bullets (•), the constants + and 3, and the variable x. The constant + should be thought of as the cascaded representation of the addition operation.

We may also describe the above term with the linear notation $x+3$. The linear description is more convenient than the tree description, but understanding it requires more notational skills.

We will take the viewpoint that the above term represents the "formula" $x+3$. Put more precisely, we will take the term as the representation of the logical structure (i.e., abstract syntax) of the formula and neglect the notation of the formula.

As a further example we consider the formula $3(x+y)$. Its abstract syntax is given by the term



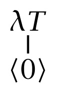
where the constants $\cdot$ and $+$ should be thought of as cascaded representations of the multiplication and addition operations for the underlying number system.

1.4 De Bruijn Representation of Bound Variables

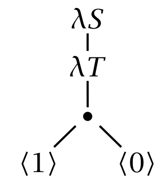
The notations $\lambda x{\in}\mathbb{N}.x$ and $\lambda y{\in}\mathbb{N}.y$ describe the same function. This tells us that the choice of the argument variable of a function description does not matter. Argument variables are merely a notational device for referring to the arguments of functions.

Similarly, the notations $\lambda x{:}T.x$ and $\lambda y{:}T.y$ describe the same term. This means that a term $\lambda x{:}T.\,t$ does not contain the particular variable x used in its description.

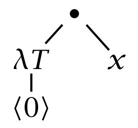
The tree description of a term represents abstractions without using argument variables. Instead, the arguments of abstractions are referred to by so-called **de Bruijn indices** invented by the Dutch logician Nicolaas de Bruijn:



$\lambda x{:}T.x$



$\lambda f{:}S.\,\lambda x{:}T.\,fx$



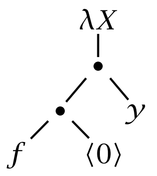
$(\lambda x{:}T.x)x$

A de Bruijn index $\langle n\rangle$ represents the argument of the abstraction whose λ-node appears on the path to the root encountered after skipping n λ-nodes.

To simplify our notation for abstractions, we will sometimes omit the type T and simply write $\lambda x.t$. This may be justified because the type is clear from the context or a certain type is fixed for the particular variable x.

1.5 Free Variables

The variables contained in a term are usually referred to as the **free variables** of the term. This is historical language based on the textual notation for terms using argument variables. For instance, the notation $\lambda x{:}X.fxy$ employs x as **bound variable** and f and y as free variables. The tree representation of the term reveals that the bound variable x is a notational device that is not part of the term:



1.6 Quantification

What is the term representation of the quantified formula $\forall x{\in}\mathbb{Z}\colon x \le y$? Since x is a bound variable and bound variables can only be introduced by abstractions, we will need an abstraction. This leads us to the term

$$\forall_{\mathbb{Z}}\,(\lambda x{:}\mathbb{Z}.\; x \le y)$$

where the constant $\forall_{\mathbb{Z}}$ is the following function:

$$\begin{aligned}&\forall_{\mathbb{Z}} \in (\mathbb{Z} \to \mathbb{B}) \to \mathbb{B}\\ &\forall_{\mathbb{Z}} f = (f = (\lambda x{\in}\mathbb{Z}.1))\end{aligned}$$

Note that $\forall_{\mathbb{Z}}$ is a very simple function: it takes a function f as argument and yields 1 if and only if f yields 1 for all arguments.

The functional representation of quantification is due to the American logician Alonzo Church.

1.7 Notational Issues

Let X be a set. We will use the following functions as constants in terms:

$$\begin{aligned}&\doteq_X \in X \to X \to \mathbb{B}\\ &x \doteq_X y = (x = y)\\[1em] &\forall_X \in (X \to \mathbb{B}) \to \mathbb{B}\\ &\forall_X f = (f = (\lambda x{\in}X.1))\\[1em] &\exists_X \in (X \to \mathbb{B}) \to \mathbb{B}\\ &\exists_X f = (f \ne (\lambda x{\in}X.0))\end{aligned}$$

We will omit the type annotations of the **polymorphic constants** $\doteq_X$, $\forall_X$ and $\exists_X$ if they are clear from the context and simply write $\doteq$, $\forall$ and $\exists$. Moreover, we will use the abbreviations

$$\begin{aligned}\forall x{:}T.\, t &\;\rightsquigarrow\; \forall_T(\lambda x{:}T.\, t)\\ \exists x{:}T.\, t &\;\rightsquigarrow\; \exists_T(\lambda x{:}T.\, t)\end{aligned}$$

Make sure that you understand the following:

- $\lambda x{\in}X.x$ describes a function.
- $\lambda x{:}X.x$ describes a term.
- $\forall x{\in}\mathbb{N}\colon 0 \le x$ describes a mathematical statement.
- $\forall x{:}\mathbb{N}.\; 0 \le x$ describes a term.

1.8 Substitution and Instances

The notation $s[x := t]$ denotes the term that is obtained from the term s by replacing all occurrences of the variable x with the term t:

$$\begin{aligned} (x+5)[x := 3] &= 3+5 \\ (x+y)[x := x+3] &= (x+3)+y \end{aligned}$$

The operation behind $s[x := t]$ is called **substitution**.

What is the result of the substitution $(\lambda x{:}C.y)[y := x]$? Here it is crucial to recall that the term $\lambda x{:}C.y$ does not contain the argument variable x used in its description. This suggests $(\lambda x{:}C.y)[y := x] \neq \lambda x{:}C.x$. If we apply the substitution to the tree description of $\lambda x{:}C.y$, the problem with the conflicting argument variable disappears:

$$(\lambda x{:}C.y)[y := x] \;=\; \begin{pmatrix} \lambda C \\ | \\ y \end{pmatrix} [y := x] \;=\; \begin{pmatrix} \lambda C \\ | \\ x \end{pmatrix} \;=\; \lambda y{:}C.x$$

Make sure that you understand every equation in this chain.

Before de Bruijn invented his indices in the 1960's, variables where the only means to refer to the argument of an abstraction. It is rather complicated to give a precise definition of substitution if terms are formalized with variables as argument references.

As a general rule keep in mind that a substitution never introduces a new reference to the argument of an abstraction.

The fact $(\lambda x{:}C.y)[y := x] \neq \lambda x{:}C.x$ is often paraphrased as "substitution must be **capture-free**".

The **instances of a term** t are defined as follows:

1. t is an instance of t
2. $x : T \;\wedge\; s : T \;\Longrightarrow\; t[x := s]$
3. s' instance of s $\wedge$ s instance of t $\Longrightarrow$ s' instance of t

1.9 Semantic Equivalence

You will certainly agree that the equation

$$(\lambda x{\in}\mathbb{N}.\,2x) = (\lambda x{\in}\mathbb{N}.\,x+x)$$

holds. On the other hand, we have

$$(\lambda x{:}\mathbb{N}.\,2x) \neq (\lambda x{:}\mathbb{N}.\,x+x)$$

since this time we compare terms rather than functions. This motivates the notation of semantic equivalence. We call two terms s and t **semantically equivalent** (written $s \equiv t$) if they describe the same value no matter how the values of their free variables are chosen. Here are examples:

$$\begin{aligned}(\lambda x{:}\mathbb{N}.\,2x) &\equiv (\lambda x{:}\mathbb{N}.\,x+x)\\(\lambda x{:}\mathbb{N}.\,x+y) &\neq (\lambda x{:}\mathbb{N}.\,y+x)\\(\lambda x{:}\mathbb{N}.\,x+y) &\equiv (\lambda x{:}\mathbb{N}.\,y+x)\end{aligned}$$

Proposition 1.1 Semantic equivalence is an equivalence relation on terms that satisfies the following properties:

Cong $\quad s \equiv s' \;\wedge\; t \equiv t' \;\Longrightarrow\; st \equiv s't'$

Sub $\quad s \equiv s' \;\Longrightarrow\; s[x:=t] \equiv s'[x:=t]$

$\xi \quad s \equiv s' \;\Longrightarrow\; \lambda x{:}T.s \equiv \lambda x{:}T.s'$

The properties Sub and ξ (read xi) result from the fact that $s \equiv t$ only holds if s and t yield the same value no matter how the values of the free variables are chosen.

1.10 The β-Law

The **β-law** says that every term $(\lambda x{:}T.s)t$ satisfies the equivalence

$$(\lambda x{:}T.s)t \equiv s[x:=t]$$

The β-law establishes substitution as the syntactic counterpart of function application. Here are instances of the law:

$$\begin{aligned}(\lambda x{:}\mathbb{N}.\ x+3)7 &\equiv 7+3\\(\lambda x{:}\mathbb{N}.\ x+3)(y+2) &\equiv (y+2)+3\end{aligned}$$

We say that a pair (s,t) is a

- **β-reduction** if t can be obtained from s by applying the β-law from left to right.
- **β-expansion** if t can be obtained from s by applying the β-law from right to left.
- **β-conversion** if (s,t) is either a β-reduction or a β-expansion.

Here are examples of β-reductions:

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

A term is called

- a **β-redex** if it is an instance of the left hand side of the β-law.
- **β-normal** if it does not contain a β-redex.

A term t is called a **β-normal form** of a term s if t is β-normal and can be obtained from s by a finite sequence of β-reductions. The sequence of β-reductions shown above demonstrates that fab is a β-normal form of $(\lambda f.fa)(\lambda xy.fxy)b$.

Theorem 1.2 (Church, Rosser, Turing) Every term has exactly one β-normal form.

Theorem 1.3 (Tait 1967) There is no infinite chain of β-reductions (i.e., an infinite sequence $t_1, t_2, t_3, \dots$ such that (t_i, t_{i+1}) is a β-reduction for all $i = 1, 2, 3, \dots$).

The proofs of the two theorems is not straightforward. Together they give us a simple algorithm for computing the β-normal form of a term: Apply β-reductions until a β-normal term is obtained. The chain of β-reductions leading to the normal form may be very long. Rick Statman [1979] showed that deciding whether two terms have the same β-normal form is not elementary recursive.

1.11 The η-Law

The **η-law** says that every term $\lambda x{:}T.fx$ where x, f are variables satisfies the equivalence

$$\lambda x{:}T.fx \equiv f$$

The η-law relies on the fact that two function $X \to Y$ are equal if they agree on all arguments $x \in X$.

The notions of η-reduction, η-expansion, η-conversion, η-redex, η-normality, and η-normal forms are defined for η the same way they are defined for β. Here are examples of η-reductions:

$$\begin{aligned} \lambda xy.fxy &= \lambda x.\lambda y.(fx)y && \\ &\equiv \lambda x.fx && \eta \\ &\equiv f && \eta \end{aligned}$$

Note that $\lambda x{:}T.sx \equiv s$ is an instance of the η-law only if x is not free in the term s. This follows from the fact that substitution is capture-free. The fact $\lambda x{:}\mathbb{N}.x \not\equiv \lambda x{:}\mathbb{N}.((+)x)x$ shows that the η-law cannot be generalized, and also that capture freeness of substitution is essential for the property Sub stated by Proposition 1.1.

Analogues of theorems 1.2 and 1.3 also hold for η-reduction.

1.12 $\beta\eta$-Normal Forms

A term is **$\beta\eta$-normal** if it contains neither a β-redex nor an η-redex. A term is called a **$\beta\eta$-normal form** of a term s if it is $\beta\eta$-normal and can be obtained from s by a finite sequence of β- and η-reductions. Here is an example:

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

Analogues of theorems 1.2 and 1.3 also hold for $\beta\eta$-reduction.

Theorem 1.4 (Friedman 1975) Let s and t be terms not containing constants. Then $s \equiv t$ if and only if s and t have the same $\beta\eta$-normal form.

Friedman's theorem says that semantic equivalence of pure functional descriptions is fully captured by the β- and η-law.

2 An Axiomatization of Terms

Terms are a complex data structure. The main complications are:

1. The choice of argument variables does not matter. For instance, $\lambda x.x = \lambda y.y$. To put it plain, we don't get what we see.
2. An application st can only be formed if the types of s and t match.

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]

The real numbers are a good example for a mathematical data structure defined with the axiomatic approach. A well-known construction of the real numbers uses so called Dedekind cuts and was devised by Richard Dedekind around 1860. Another construction of the real numbers employs equivalence classes of rational Cauchy sequences.

To validate an axiomatization, at least one model (i.e., a construction satisying the axioms) has to be devised. The added value that comes with an axiomatization is the fact that it doesn't commit us to a particular model. Rather, an axiomatization gives us the freedom to work with different models, as long as they satisfy the axioms.

An axiomatization saves as basis for proofs. Proofs based on an axiomatization are automatically valid for all models of the axiomatization.

We will now develop an axiomatization of terms that we will use as the base for all proofs and algorithms to come. The axiomatization captures the syntactic aspects of terms. The semantic interpretation of terms will be defined on top of the axiomatization.

2.1 Types

We start with an axiomatization of types.

2.1.1 Axiomatization

We assume two sets

$$Sor \subseteq Ty$$

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

whose elements we call **sorts** and **types**, respectively. Types that are not sorts are called **functional types**. Furthermore, we assume two functions:

$F \in Ty \to Ty \to Ty$	function type
$\lvert\cdot\rvert \in Ty \to \mathbb{N}$	size

and arrange the following notations:

$C, D \in Sor$	sorts
$S, T \in Ty$	types
$S \to T \quad \rightsquigarrow \quad FST$	

The first and the second notational line introduce so-called **meta-variables**. The first notational line states that the meta-variables C, D will always denote sorts, if not said otherwise. Similiarly, the second notational line states that the meta-variables S, T will always denote types. Finaly, the last line says that we may write $S \to T$ in place of FST.

We assume that types satisfy the following axioms (i.e., basic properties):

Par For every type T, exactly one of the following conditions is satisfied:
(1) $T \in Sor$ (2) $\exists S, S' \colon\ T = S \to S'$

CS $|C| = 1$

CF $|S \to T| = 1 + |S| + |T|$

IF $S \to T = S' \to T' \iff S = S' \land T = T'$

Inf $Sor \cong \mathbb{N}$

Axiom Par says that a type is either basic or functional, and that there is no type that is both basic and functional. The axiom CF says that every functional types can be obtained in finitely many steps from sorts. Together, Par and CF make it possible to prove claims about types by induction on $|T|$. Axioms IF says that two functional types are equal if both their argument types and their result types are equal. The final axiom Inf says that we may think of sorts as natural numbers. Formally, $X \cong Y$ means that there is a bijection between the set X and the set Y.

We insist on infinitly many sorts since we see sorts as names that may or may not be given a fixed meaning by an interpretation. Interpretations capture the semantic aspects of terms and will be defined later.

Every axiom holds for all possible values of the unquantified meta-variables occurring in it (so-called **universal interpretation**). For instance, the axiom CS holds for all sorts C, and the axiom CF holds for all types S and T. Written explicitly, axioms CS and CF look as follows:

CS $\forall C \in Sor\colon\ |C| = 1$

CF $\forall S, T \in Ty\colon\ |S \to T| = 1 + |S| + |T|$

2.1.2 Standard Model

We obtain a model of the axiomatization of types by defining the sets *Sor* and *Ty* as follows:

$$\begin{aligned} Sor &:= \{1\} \times \mathbb{N} \\ Ty &:= Sor \cup (\{2\} \times (Ty \times Ty)) \end{aligned}$$

The equation for *Ty* is to be interpreted recursively, that is, *Ty* contains only those pairs that can be obtained in finitely many steps from *Sor*. The definition of F and $|\cdot|$ is now straightforward:

$$\begin{aligned} FST &:= (2, (S, T)) \\ |(1, n)| &:= 1 \\ |(2, (S, T))| &:= 1 + |S| + |T| \end{aligned}$$

Note that the definition of $|\cdot|$ is recursive since the definition of *Ty* is recursive.

2.2 Terms

We continue with an axiomatization of terms.

2.2.1 Axiomatization

The axiomatization of terms assumes three sets

$$Var \subseteq Ind \subseteq Ter$$

whose elements are called **variables**, **individual names** and **terms**, respectively. Individual names that are not variables are called **constants.** We assume

$$Ty \cap Ter = \emptyset$$

and arrange the following notations:

$x, y, z \in Var$			variables
$u, v \in Ind$			individual names
$s, t \in Ter$			terms
Con	$\rightsquigarrow$	$Ind - Var$	constants
$a, b, c \in Con$			constants
Nam	$\rightsquigarrow$	$Sor \cup Ind$	names

We assume the following functions

$A \in Ter \rightharpoonup Ter \rightharpoonup Ter$	application
$L \in Var \to Ter \to Ter$	abstraction
$\tau \in Ter \to Ty$	type
$\lvert \cdot \rvert \in Ter \to \mathbb{N}$	size
$\mathcal{N} \in (Ty \cup Ter) \to \mathcal{P}(Nam)$	names
$\mathbf{S} \in Sub \to Ter \to Ter$	substitution

and arrange the following notations:

$$\begin{array}{rcll} st & \rightsquigarrow & Ast & \\ \lambda x.t & \rightsquigarrow & Lxt & \\ t{:}T & \rightsquigarrow & \tau t = T & \\ Sub & \rightsquigarrow & \{\, f \in Ind \to Ter \mid \forall u\colon\ \tau(fu) = \tau u \,\} & \textbf{substitutions} \\ \theta \in Sub & & & \\ s[x := t] & \rightsquigarrow & \mathbf{S}\ (\lambda y \in Var.\ \text{if } x = y \text{ then } t \text{ else } y)\ s & \end{array}$$

The functions A and L provide for the construction of applications and abstractions. Note that A is a partial function. This reflects the fact that applicative terms can only be formed if a type constraint is satisfied. The function τ yields the type of a term. The existence of τ formalizes the assumption that every term has exactly one type and that individual names come with a built-in type. The size function provides for the axiomatization of the fact that every term can be constructed in finitely many steps from individual names. The function $\mathcal{N}$ yields all names that occur in a term. The function $\mathbf{S}$ provides for substitution.

Here are the axioms for terms:

DA1 $Dom\, A = \{\, t \mid \tau t \text{ functional} \,\}$

DA2 $Dom\,(At) = \{\, s \mid \exists T\colon\ \tau t = \tau s \to T \,\}$

Par For every term t, exactly one of the following conditions is satisfied:
(1) $t \in Ind$ (2) $\exists s, s'\colon\ t = ss'$ (3) $\exists x, s\colon\ t = \lambda x.s$

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

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

TA $\tau t = \tau s \to \tau(ts)$

TL $\tau(\lambda x.t) = \tau x \to \tau t$

CN $|u| = 1$

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

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

NS $\mathcal{N}C = \{C\}$

NF $\mathcal{N}(S \to T) = \mathcal{N}S \cup \mathcal{N}T$

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

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

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

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

SA $\mathbf{S}\theta(ts) = (\mathbf{S}\theta t)(\mathbf{S}\theta s)$

SL $(\forall u \in \mathcal{N}(\lambda x.t)\colon\ x \notin \mathcal{N}(\theta u)) \implies \mathbf{S}\theta(\lambda x.t) = \lambda x.\mathbf{S}(\theta[x := x])t$

Inf $\forall T\colon\ \{\, x \mid \tau x = T \,\} \cong \mathbb{N} \ \wedge\ \{\, c \mid \tau c = T \,\} \cong \mathbb{N}$

Most of the axioms are obvious given an intuitive understanding of terms. We will say more about the axioms IL, SL and Inf.

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.

Inf states that there are infinitly many variables and constants for every type.

SL states how substitution applies to abstractions. As we will see, the precondition of SL can always be satisfied by choosing a suitable argument variable.

If an axiom contains an application of the partial function A or **S**, the axiom holds only for those values of the meta-variables that make the applications of A and S well-defined. For instance, the axiom TA holds only for those terms s and t such that t is functional and $s \in \mathit{Dom}\,(At)$. Written explicitly, axioms

TA $\tau t = \tau s \to \tau(ts)$

TL $\tau(\lambda x.t) = \tau x \to \tau t$

look as follows:

TA $\forall s,t \in \mathit{Ter}\colon\ t \in \mathit{Dom}\,A \ \wedge\ s \in \mathit{Dom}\,(At) \implies \tau t = F(\tau s)(\tau(Ats))$

TL $\forall x \in \mathit{Var}\ \forall t \in \mathit{Ter}\colon\ \tau(Lxt) = F(\tau x)(\tau t)$

Note that the meta-variable x used in the compact formulation of TL is not bound by the preceeding λ. Make sure you understand why this is the case.

2.2.2 Standard Model

The standard model for terms extends the standard model for types. The definition of variables, constants and individual names is easy:

$$\begin{aligned} Con &:= \{3\} \times (\mathbb{N} \times Ty) \\ Var &:= \{4\} \times (\mathbb{N} \times Ty) \\ Ind &:= Con \cup Var \end{aligned}$$

The definition of the set Ter requires more ingenuity. We will first define recursively a larger set whose elements we call **quasi-terms**:

$$QT := Ind \cup (\{5\} \times (QT \times QT)) \cup (\{6\} \times (Ty \times QT)) \cup (\{7\} \times \mathbb{N})$$

Applications are modeled as pairs starting with 5, abstractions as pairs starting with 6, and argument references (de Bruijn indices) as pairs starting with 7.

Let X be a set. We use X^* denote the set of all tuples whose components are in X. The elements of X^* may be thought of as strings or vectors. We use the following notations for $v \in X^*$:

- $v.n$ denotes the $(n+1)$-th component of v. For instance, $\langle 3,7,4\rangle.1 = 7$.
- $x :: v$ denotes the tuple obtained from v by adding x as leftmost component. For instance, $9 :: \langle 3,7,4\rangle = \langle 9,3,7,4\rangle$.

To define the set $Ter \subseteq QT$ of terms, we will first define an **admissibiliy relation** $R \subseteq Ty^* \times QT \times Ty$. Given R, we define the set of terms as follows:

$$Ter := \{\, t \in QT \mid \exists T\colon\ (\langle\rangle, t, T) \in R \,\}$$

Intuitively, this definition may be read as follows: a quasi-term t is a term if it doesn't contain dangling de Bruijn indices and if it is well-typed.

For the definition of R we introduce the following notations:

$$\begin{aligned} st &\rightsquigarrow (5,(s,t)) \\ \lambda T.t &\rightsquigarrow (6,(T,t)) \end{aligned}$$

We define R recursively by the following inference rules, where we assume $\Gamma \in Ty^*$, $n \in \mathbb{N}$, $S,T \in Ty$, and $s,t \in Ter$.

$$\frac{c = (3,(n,T))}{(\Gamma,c,T) \in R} \qquad \frac{x = (4,(n,T))}{(\Gamma,x,T) \in R} \qquad \frac{i = (7,n) \quad \Gamma.n = T}{(\Gamma,i,T) \in R}$$

$$\frac{(\Gamma,\ t,\ S \to T) \in R \quad (\Gamma,\ s,\ S) \in R}{(\Gamma,\ ts,\ T) \in R} \qquad \frac{(S :: \Gamma,\ t,\ T) \in R}{(\Gamma,\ \lambda S.t,\ S \to T) \in R}$$

It remains to define the functions A, L, τ, $|\cdot|$, $\mathcal{N}$, and $\mathbf{S}$. The definition of A, τ, $|\cdot|$, and $\mathcal{N}$ is straightforward, while L and $\mathbf{S}$ require more ingenuity. It also remains to verify that the axioms are satisfied, which in some cases is tedious.

2.3 Proof of Basic Properties

Based on the axioms, we can now prove properties that are satisfied by every model of axiomatization of terms.

In the following, we assume that some model is given. Since we don't make any special assumptions about this model, the following propositions will hold for all models.

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

Proof Exercise. ∎

Proposition 2.2 $\mathcal{N}t$ is a finite set.

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

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

Case $t = ss'$. Then $\mathcal{N}t = \mathcal{N}s \cup \mathcal{N}s'$ by NA. Hence $\mathcal{N}t$ finite by induction hypothesis.

Case $t = \lambda x.s$. Then $\mathcal{N}t = \mathcal{N}(\tau x) \cup (\mathcal{N}s - \{x\})$ by NL. Hence $\mathcal{N}t$ finite by Proposition 2.1 and induction hypothesis. ∎

Proposition 2.3 (Type Preservation) $\tau(\mathbf{S}\theta t) = \tau t$

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

Case $t = u$. Then $\tau(\mathbf{S}\theta t) = \tau(\theta u) = \tau u = \tau t$ by SN.

Case $t = ss'$. Then

$$\begin{array}{lr}
\mathbf{S}\theta t = (\mathbf{S}\theta s)(\mathbf{S}\theta s') & \text{SA} \\
\tau(\mathbf{S}\theta s) = \tau(\mathbf{S}\theta s') \to \tau(\mathbf{S}\theta t) & \text{TA} \\
\tau s = \tau s' \to \tau(\mathbf{S}\theta t) & \text{induction hypothesis}
\end{array}$$

Since $\tau s = \tau s' \to \tau t$ by TA, we have $\tau(\mathbf{S}\theta t) = \tau t$ by IF.

Case $t = \lambda x.s$. By Proposition 2.2, Inf and IL we can assume $x \notin \mathcal{N}(\theta u)$ for all $u \in \mathcal{N}t$. Hence

$$\begin{array}{rlr}
\tau(\mathbf{S}\theta t) & = \tau(\lambda x.\mathbf{S}(\theta[x := x])s) & \text{SL} \\
& = \tau x \to \tau(\mathbf{S}(\theta[x := x])s) & \text{TL} \\
& = \tau x \to \tau s & \text{induction hypothesis} \\
& = \tau t & \text{TL}
\end{array}$$

∎

Proposition 2.4 (Coincidence) $(\forall u \in \mathcal{N}t\colon\ \theta u = \theta' u) \implies \mathbf{S}\theta t = \mathbf{S}\theta' t$

Proof Exercise. ∎

Proposition 2.5 $\mathbf{S}(\lambda u{\in}Ind.u)t = t$

Proof Exercise. ∎

2.4 Renaming

Proposition 2.6 (Renaming) $y \notin \mathcal{N}t \implies (t[x{:=}y])[y{:=}s] = t[x{:=}s]$

Proposition 2.7 $v \in \mathcal{N}(\mathbf{S}\theta t) \iff (\exists u \in \mathcal{N}t\colon\ v \in \mathcal{N}(\mathbf{S}\theta u))$

Proposition 2.8 $t' = t[x{:=}x'] \implies (t = t'[x'{:=}x] \iff x' \notin \mathcal{N}(\lambda x.t))$

Proposition 2.9 $\lambda x.t = \lambda x'.t' \iff t = t'[x'{:=}x] \wedge t' = t[x{:=}x']$

2.5 Beta and Phi

We fix a function

$$\beta \in Ter \to Ter \to Ter$$

such that

$$\beta(\lambda x.t)s = t[x{:=}s]$$

for every variable x and all terms s, t such that $\tau x = \tau s$. The existence of β follows from IL and Proposition 2.6.

Exercise 2.10 Explain why the existence of β is not obvious.

Proposition 2.11 (Beta) For every abstraction $t : S \to T$ and every variable $x : S$, the following equivalence holds:

$$t = \lambda x.s \iff x \notin \mathcal{N}t \wedge s = \beta t x$$

Proof Let $t : S \to T$ be an abstraction and $x{:}S$ be a variable. Since t is an abstraction, there exist x' and s' such that $t = \lambda x'.s'$. Since $\beta t x = s'[x'{:=}x]$ it remains to show that

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

for ever term s. This property is an instance of Axiom IL. ∎

There are exactly as many ways to describe an abstraction $t : S \to T$ with λ as there are variables $x : S$ such that $x \notin \mathcal{N}t$. Often it is convenient to have a fixed λ-description for every abstraction. To this purpose we fix a function

$$\varphi \in \mathit{Ter} \to \mathit{Var}$$

such that $\varphi t \notin \mathcal{N}t$ and $\varphi t : S$ for every abstraction $t : S \to T$. The existence of such a function follows from Proposition 2.2 and Inf.

Proposition 2.12 (Unique Decomposition) Let t be an abstraction and $x = \varphi t$. Then there exists one and only one term s such that $t = \lambda x.s$.

3 Structures and Specifications

Terms provide us with a formal specification language for set-theoretic structures. In this language, a specification is a set of equations, and a structure satisfies a specification if it satisfies each of its equations. The idea is well-known from algebra: The axioms for groups are a specification, and the groups are the structures satisfying this specification.

3.1 Evaluation

We start with the evaluation of terms. As an example, consider the term $x + 3$. It evaluates to 5 if x takes the value 2 and the names + and 3 take the values the symbols + and 3 suggest. The example tells us that the evaluation of a term requires a function that assigns values to names. We call such functions *interpretations* and define them as follows.

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

1. $\mathit{Dom}\,\mathcal{I} = \mathit{Ty} \cup \mathit{Con} \cup \mathit{Var}$
2. $\mathcal{I}u \in \mathcal{I}(\tau u)$
3. $\mathcal{I}(S \to T) = \{\, f \mid f \text{ function } \mathcal{I}S \to \mathcal{I}T \,\}$

We require interpretations to be defined on all types, on all constants, and all variables since this is convenient and serves the purpose. Condition (3) ensures that functional types are interpreted as one would expect. Thus we know how an interpretation behaves on functional types if we know how it behaves on sorts. Conditation (2) says that the values of constants and variables must be taken from the interpretation of their types.

Proposition 3.1 (Coincidence) If $\mathcal{I}$ and $\mathcal{I}'$ agree on all names, 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 3.2 $\mathcal{I}T \neq \emptyset$.

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

Given an interpretation $\mathcal{I}$, a variable 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:

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

Proposition 3.3 (Evaluation) For every interpretation $\mathcal{I}$ there exists one and only one function $\hat{\mathcal{I}}$ such that:

1. $\mathit{Dom}\,(\hat{\mathcal{I}}) = \mathit{Ter}$
2. $\hat{\mathcal{I}}t \in \mathcal{I}(\tau t)$
3. $\hat{\mathcal{I}}u = \mathcal{I}u$
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 modified such that it requires $x = \varphi(\lambda x.t)$. The properties (1), (2) and the uniqueness of $\hat{\mathcal{I}}$ are immediate consequences of this definition. The unmodified version of (5) can be shown with the Proposition 3.4 whose proof can be based on our 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:

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

Proposition 3.4 (Substitution) $\hat{\mathcal{I}}(\mathsf{S}\theta t) = \hat{\mathcal{I}}_\theta t$.

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

Proposition 3.5 (Coincidence) If $\mathcal{I}$ and $\mathcal{I}'$ agree on $\mathcal{N}t$, then $\hat{\mathcal{I}}t = \hat{\mathcal{I}}'t$.

3.2 Signatures and Structures

When we use terms as specification language, we consider only certain sorts and certain constants. A collection of relevant sorts and constants will be called a *signature*, and an interpretation for the names of a signature will be called a *structure*. The precise definitions are as follows.

A **signature** is a set $\Sigma \subseteq \mathit{Sor} \cup \mathit{Con}$ such that $\forall c \in \Sigma\colon\ \mathcal{N}(\tau c) \subseteq \Sigma$. Note that we require that a signature is closed in the sense that if it contains a constant, it must also contain the sorts in the type of the constant.

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** $\mathcal{A}$.

A type T is **licensed** by a signature Σ if $\mathcal{N}T \subseteq \Sigma$. A term t is **licensed** by a signature Σ if the following conditions hold:

1. $\mathcal{N}t - Var \subseteq \Sigma$.
2. $\forall x \in \mathcal{N}t\colon\ \mathcal{N}(\tau x) \subseteq \Sigma$

An interpretation $\mathcal{I}$ is **licensed** by a structure $\mathcal{A}$ if $\mathcal{A} \subseteq \mathcal{I}$. A type or a term are **licensed** by a structure if they are licensed by the signature of the structure.

Proposition 3.6 (Coincidence) Let $\mathcal{I}$ and $\mathcal{I}'$ be licensed by $\mathcal{A}$. Then:

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

Example 3.7 We present examples for a signature and a structure. We start with the description of a signature Σ:

$$\begin{aligned} 0, 1 &: B \\ \rightarrow &: B \rightarrow B \rightarrow B \end{aligned}$$

The described signature Σ consists of a sort B and three different constants 0, 1, and $\rightarrow$. Since we base everything on the axiomatization of terms, we cannot name concrete sorts and constants. However, we can assume that the symbol B denotes a concrete sort, and that the symbols 0, 1, and $\rightarrow$ denote concrete constants of the types specified in the description of the signature (existence guaranteed by Inf). This way we get what we want together with a nice notation. We can now define a structure $\mathcal{B}$ that interprets the names of Σ:

$$\begin{aligned} \mathcal{B}B &= \mathbb{B} \\ \mathcal{B}0 &= 0 \\ \mathcal{B}1 &= 1 \\ \mathcal{B}(\rightarrow) &= \lambda v \in \mathbb{B}.\, \lambda w \in \mathbb{B}.\, \max\{1 - v, w\} \end{aligned}$$

■

Two structures $\mathcal{A}$ and $\mathcal{B}$ are **isomorphic** if $\Sigma_{\mathcal{A}} = \Sigma_{\mathcal{B}}$ and for every type T licensed by $\Sigma_{\mathcal{A}}$ there exists a bijection $\gamma_T\colon \mathcal{A}T \rightarrow \mathcal{B}T$ such that:

1. For every constant $c \in \Sigma_{\mathcal{A}}$: $\gamma_{\tau c}(\mathcal{A}c) = \mathcal{B}c$.
2. For every type $T_1 \rightarrow T_2$ licensed by $\Sigma_{\mathcal{A}}$ and ever function $f \in \mathcal{A}(T_1 \rightarrow T_2)$: $(\gamma_{T_1 \rightarrow T_2})f = \{\, (\gamma_{T_1} v_1, \gamma_{T_2} v_2) \mid (v_1, v_2) \in f \,\}$.

To show that two structures $\mathcal{A}$ and $\mathcal{B}$ are isomorphic, it suffices to exhibit a bijection $\gamma_C\colon \mathcal{A}C \rightarrow \mathcal{B}C$ for every sort $C \in \Sigma_{\mathcal{A}}$. The bijections for the functional types can then obtained by recursion according to condition (2). Of course, one has to check that condition (1) is satisfied.

3.3 Equations

An **equation** of type T is a pair (s, t) of two terms $s{:}T$ and $t{:}T$. If there is no danger of confusion, we will write an equation (s, t) as $s{=}t$. An equation $s{=}t$ is

licensed by a signature Σ if s and t are licensed by Σ. We arrange the following notations:

$$
\begin{array}{rcll}
e \in Equ & := & \{\, (s,t) \mid \tau s = \tau t \,\} & \text{equations} \\
\mathcal{N} e & := & \mathcal{N} s \cup \mathcal{N} t \quad \text{if} \;\; e = (s,t) & \text{names} \\
\mathcal{I} \vDash e & :\Longleftrightarrow & \hat{\mathcal{I}} s = \hat{\mathcal{I}} t \quad \text{if} \;\; e = (s,t) & \mathcal{I}\ \textbf{satisfies}\ e \\
\mathcal{A} \vDash e & :\Longleftrightarrow & \forall \mathcal{I}\colon\ \mathcal{A} \subseteq \mathcal{I} \Rightarrow \mathcal{I} \vDash e & \mathcal{A}\ \textbf{satisfies}\ e \\
VE\,\mathcal{A} & := & \{\, e \mid \mathcal{A} \vDash e \,\} & \textbf{valid equations}
\end{array}
$$

Note that a structure $\mathcal{A}$ satisfies an equations e if and only if all interpretations licensed by $\mathcal{A}$ satisfy e. Instead of $\mathcal{A}$ satisfies e we also say that e is **valid in** $\mathcal{A}$.

For the structure $\mathcal{B}$ from Example 3.7 we have the following:

$$
\begin{array}{l}
\mathcal{B} \vDash 1 \to x{=}x \\
\mathcal{B} \nvDash 0 \to x{=}x
\end{array}
$$

Proposition 3.8 Let $\mathcal{A}$ and $\mathcal{B}$ be isomorphic structures. Then $\mathcal{A} \vDash e \iff \mathcal{B} \vDash e$.

3.4 Specifications and Models

A **specification** is a set of equations. The equations of a specification are called the **axioms** of the specification. A **model** of a specification is a structure that satisfies all axioms of the specification. The **signature** of a specification is the least signature that licenses all axioms of the specification. We use Σ_A to denote the signature of a specification A and arrange the following notations:

$$
\begin{array}{rcll}
A, E \subseteq Equ & & & \text{specifications} \\
\mathcal{A} \vDash A & :\Longleftrightarrow & \forall e{\in}A\colon\ \mathcal{A} \vDash e & \mathcal{A}\ \text{model of}\ A,\ \mathcal{A}\ \textbf{satisfies}\ A \\
\mathcal{N} A & := & \bigcup\{\, \mathcal{N} e \mid e \in A \,\} &
\end{array}
$$

Figure **??** shows the description of a specification Bool. Both the signature and the axioms are described. The explicit description of the signature provides notations for the sorts and constants of the specification. A declaration of the variables used in the axioms is not necessary since their types can be inferred from the axioms:

$$
\begin{array}{l}
x : B \\
f : B \to B
\end{array}
$$

Convince yourself that the structure $\mathcal{B}$ from Example 3.7 is a model of Bool.

Specification	Bool	
Sorts	B	
Constants	$0, 1 \colon B$	
	$\to \colon B \to B \to B$	
Axioms	$0 \to x = 1$	I0
	$1 \to x = x$	I1
	$f0 \to f1 \to fx = 1$	BCA (Boolean case analysis)

Figure 1: Specification Bool

A specification A entails an equation e semantically if every model of A satisfies e:

$$A \vDash e \;:\Longleftrightarrow\; \forall \text{ model } \mathcal{A} \text{ of } A\colon\; \mathcal{A} \vDash e \qquad\qquad A \textbf{ entails } e \textbf{ semantically}$$

Our definition of models is quite liberal. In particular it admits models that interpret a sort C with a one-element set. Such models satisfy all equations of type C. In fact, every structure that interprets all sorts with one-element sets will be a model of every specification.

A **proper model** of a specification A is a model $\mathcal{A}$ of A such that $\Sigma_{\mathcal{A}} = \Sigma_A$ and $\mathcal{A}C$ has at least 2 elements for every sort $C \in \mathit{Dom}\,\mathcal{A}$.

A specification is **categorical** if it has a proper model and all its proper models are isomorphic.

Proposition 3.9 The specification Bool from Figure **??** is categorical.

Proof Let $\mathcal{A}$ be a proper model of Bool. It suffices to show that $\mathcal{A}B \subseteq \{\mathcal{A}0, \mathcal{A}1\}$ since then $\mathcal{A}B = \{\mathcal{A}0, \mathcal{A}1\}$ and $\mathcal{A}0 \neq \mathcal{A}1$ by the properness of $\mathcal{A}$ and hence $\mathcal{A}(\to)$ is determined by the axioms I0, I1.

Suppose there exists a value $v \in \mathcal{A}B - \{\mathcal{A}0, \mathcal{A}1\}$. Then there exists an interpretation $\mathcal{I}$ such that $\mathcal{A} \subseteq \mathcal{I}$ and

$$\begin{aligned} \mathcal{I}x &= v \\ \mathcal{I}fv &= v \\ \mathcal{I}f(\mathcal{A}0) &= \mathcal{A}1 \\ \mathcal{I}f(\mathcal{A}1) &= \mathcal{A}1 \end{aligned}$$

Since $\mathcal{I}$ satisfies Axiom BCA and I1, we know $\hat{\mathcal{I}}(fx) = \hat{\mathcal{I}}1$. Hence $v = \mathcal{A}1$, which contradicts our assumption. ∎

We now know that the structure $\mathcal{B}$ from Example 3.7 is the only proper model of Bool, up to isomorphism. This means that the specification Bool specifies everything that is essential about $\mathcal{B}$.

We arrange the following notations:

$A \vDash A' \;:\Longleftrightarrow\; \forall e \in A' \colon\; A \vDash e$	A **entails** A' **semantically**
$A \,\rightleftharpoons\, A' \;:\Longleftrightarrow\; A \vDash A' \;\wedge\; A' \vDash A$	A, E **semantically equivalent**

Proposition 3.10

- $A \vDash A' \iff$ every model of A is a model of A'
- $A \,\rightleftharpoons\, A' \iff A$ and A' have the same models

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

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

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

Figure 2: Deduction rules

4 Equational Deduction

Given an equational specification, one can infer semantically entailed equations by “replacing equals with equals”, a proof method known as equational deduction. Equational deduction is a syntactic proof method since it is based on syntactic rules rather than semantic arguments.

Figure 2 shows the so-called **deduction rules**. Each deduction rule states a pattern according to which an equation (the **conclusion** below the bar) can be obtained from given equations (the **premises** above the bar). Formally, each rule describes a set of pairs (E, e) (the **instances of the rule**) where E is the set of premises and e is the conclusion. The rules ξ and η, for instance, describe the following sets of instances:

$$\xi\colon\quad \{\, (\{s = s'\},\ \lambda x.s = \lambda x.s') \mid x \in Var\ \wedge\ s, s' \in Ter\ \wedge\ \tau s = \tau s' \,\}$$
$$\eta\colon\quad \{\, (\emptyset,\ \lambda x.sx = s) \mid s \in Ter\ \wedge\ x \notin \mathcal{N}s \,\}$$

The rules Ref, Sym and Trans provide the equivalence properties of equality. The rules CL, CR and ξ provide the so-called congruence properties of equality. They make it possible to replace equals with equals within a term. Note that rule ξ exploits the fact that variables are universally quantified (x may occur in s and s'). Rule β and η provide basic equational properties of abstractions we have discussed before. The fundamental property of the deuction rules is soundness:

Proposition 4.1 (Soundness) If (E, e) is an instance of a deduction rule, then $E \vDash e$.

A **derivation of** e **from** A is a tuple $(e_1, \dots, e_n)$ such that $e = e_n$ and for every $i \in \{1, \dots, n\}$: $e_i \in A$ or there exists a set $E \subseteq \{e_1, \dots, e_{i-1}\}$ such that (E, e_i)

is an instance of a deduction rule. We can now define **deductive entailment** as follows:

$A \vdash e$	$:\iff$ $\exists$ derivation of e from A	A **entails** e **deductively**
$A \vdash E$	$:\iff$ $\forall e \in E\colon\ A \vdash e$	A **entails** E **deductively**

Proposition 4.2 (Soundness) $A \vdash e \implies A \vDash e$

Deductive equivalence of specifications is defined as follows:

$A \mathrel{\vdash\!\dashv} A' \ :\iff\ A \vdash A' \ \wedge\ A' \vdash A$ $\qquad$ A, A' **deductively equivalent**

By the soundness property we know that deductive equivalence implies semantic equivalence:

Proposition 4.3 $A \mathrel{\vdash\!\dashv} A' \implies A \mathrel{\vDash\!\Dashv} A'$

Proposition 4.4 (Extensionality)
1. $\{\lambda x.s = \lambda x.t\} \mathrel{\vdash\!\dashv} \{s = t\}$
2. $x \notin \mathcal{N}(s = t) \implies \{sx = tx\} \mathrel{\vdash\!\dashv} \{s = t\}$

Proof Here is a derivation that proves $\vdash$ of (1):

$\lambda x.s = \lambda x.t$	
$(\lambda x.s)x = (\lambda x.t)x$	CL
$(\lambda x.s)x = s$	β
$s = (\lambda x.s)x$	Sym
$(\lambda x.t)x = t$	β
$s = (\lambda x.t)x$	Trans
$s = t$	Trans

The other proofs are similar. Exercise! ■

Proposition 4.5 (Finiteness) If $A \vdash e$, then there exists a finite subset $A' \subseteq A$ such that $A' \vdash e$.

Example 4.6 Let $fyx = a$ be an equation where f and a are constants and x and y are variables such that $\tau x = \tau y$. The following outlines a derivation of

$fyx = a$ from $\{fxy = a\}$.

$$\begin{array}{lll}
& fxy = a & \\
\vdash & \lambda x.fxy = \lambda x.a & \xi \\
\vdash & \lambda yx.fxy = \lambda yx.a & \xi \\
\vdash & (\lambda yx.fxy)x = (\lambda yx.a)x & \text{CL} \\
\vdash & \lambda x'.fx'x = \lambda x.a & \beta,\ \text{Sym},\ \text{Trans} \\
\vdash & (\lambda x'.fx'x)y = (\lambda x.a)y & \text{CL} \\
\vdash & fyx = a & \beta,\ \text{Sym},\ \text{Trans}
\end{array}$$ ■

The example suggests that we can deduce from e every instance of e that is obtained by instantiation of some variables of e. This property is called *generativity*. We will make use of the following notation:

$Ker\theta \ := \ \{\, u \in Ind \mid \theta u \neq u \,\}$ **Kernel of θ**

Proposition 4.7 (Generativity) $Ker\theta \subseteq Var \implies \{e\} \vdash \mathbf{S}\theta e$

This proposition can be proven with the following lemma:

Lemma 4.8 $\emptyset \vdash \mathbf{S}\{x_1{:=}s_1, \dots, x_n{:=}s_n\}t = (\lambda x_1 \dots x_n.\, t)s_1 \dots s_n$

Proof By induction on n. ■

Deductive generativity implies semantic generativity (by soundness):

Proposition 4.9 (Generativity) $Ker\theta \subseteq Var \implies \{e\} \vDash \mathbf{S}\theta e$

A substitution θ is **invertible** if there exists a substitution ψ such that $\mathbf{S}\psi(\mathbf{S}\theta s) = s$ for all terms s. A **variable renaming** is an invertible substitution θ such that $Ker\theta \subseteq Var$.

Proposition 4.10 (Variable Renaming) θ variable renaming $\implies \{e\} \dashv\vdash \{\mathbf{S}\theta e\}$

Proof Easy consequence of Generativity. ■

Another important property of the entailment relations is *stability*. We say that a deduction rule is **stable** if for every instance (E, e) of the rule and every substitution θ the pair $(\mathbf{S}\theta E,\ \mathbf{S}\theta e)$ is an instance of the rule.

Proposition 4.11 All deduction rules but ξ are stable.

We say that a substitution θ is **stable** for an equation e if it satisfies the following conditions:

1. $Ker\theta \subseteq Con$
2. $\forall c \in \mathcal{N}e\ \forall x \in \mathcal{N}(\mathbf{S}\theta c){:}\ \ x \notin \mathcal{N}e$

We say that a substitution θ is **stable** for a set of equations E if θ is stable for every equation in E.

Proposition 4.12 If $Ker\theta \subseteq Con$ and θc is closed for all constants c, then θ is stable for every equation.

Proposition 4.13 (Stability) Let θ be stable for A. Then:

1. $A \vdash e \implies \mathbf{S}\theta A \vdash \mathbf{S}\theta e$
2. $A \vDash e \implies \mathbf{S}\theta A \vDash \mathbf{S}\theta e$

The proof of this proposition is not straightforward.

Example 4.14 By Generativity we know $\{fax = x\} \vdash fay = y$. The substitution $\theta = \{a := x\}$ is not stable for $\{fax = x\}$ and in fact $\{fxx = x\} \nvdash fxy = y$ since there is structure $\mathcal{A}$ such that $\mathcal{A} \vDash fxx = x$ and $\mathcal{A} \nvDash fxy = y$. Exercise: Find such a structure. ■

A **duality** for a specification A is a substitution δ such that:

1. δ stable for A
2. $\forall s{:}\ \ A \vdash \mathbf{S}\delta(\mathbf{S}\delta s) = s$
3. $A \vdash \mathbf{S}\delta A$

Proposition 4.15 (Duality) Let δ be a duality for A. Then:

1. $A \vdash e \iff A \vdash \mathbf{S}\delta e$
2. $A \vDash e \iff A \vDash \mathbf{S}\delta e$

Proof We proof (1) as follows:

$$\begin{array}{rll}
A \vdash e & \implies \mathbf{S}\delta A \vdash \mathbf{S}\delta e & \text{stability} \\
 & \implies A \vdash \mathbf{S}\delta e & \delta \text{ duality, } (3) \\
 & \implies \mathbf{S}\delta A \vdash \mathbf{S}\delta(\mathbf{S}\delta e) & \text{stability} \\
 & \implies A \vdash e & \delta \text{ duality, } (3) \text{ and } (2)
\end{array}$$

The proof of (2) is similar and exploits soundness. ■

Example 4.16 $\delta = \{0{:=}1,\ 1{:=}0,\ +{:=}\cdot,\ \cdot{:=}+\}$ is a duality for BA that satisfies $\mathbf{S}\delta(\text{BA}) = \text{BA}$ and $\mathbf{S}\delta(\mathbf{S}\delta s) = s$. ■

Specification	BA		
Sorts	B		
Constants	$0, 1 \; : \; \mathrm{B}$		
	$\bar{} \; : \; \mathrm{B} \to \mathrm{B}$		
	$+, \cdot \; : \; \mathrm{B} \to \mathrm{B} \to \mathrm{B}$		
Axioms			
Commutativity		$xy = yx$	$x + y = y + x$
Assocativity		$(xy)z = x(yz)$	$(x + y) + z = x + (y + z)$
Distributivity		$x(y + z) = xy + xz$	$x + yz = (x + y)(x + z)$
Identity		$x1 = x$	$x + 0 = x$
Complement		$x\bar{x} = 0$	$x + \bar{x} = 1$

Figure 3: Specification BA

5 Boolean Algebra

Boolean Algebra is a theory that originated with the work of George Boole (Laws of Thought, 1854). Boole's goal was an axiomatization of the logical operations conjunction, disjunction and negation. 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.

5.1 The Specification

Our starting point is the specification BA in Figure 3, which employs the following constants:

$0, 1 : B$	
$\bar{} : B \to B$	negation
$+, \cdot : B \to B \to B$	disjunction, conjunction

For $+$ and $\cdot$ we use infix notation, where $\cdot$ takes precedence over $+$. We also write st for $s \cdot t$, which gives us $x + yz = x + (y \cdot z) = (+)x((\cdot)yz)$. This somewhat daring notation will work fine as long as all variables have type B, which usually will be the case.

The models of BA are known as **Boolean Algebras**. The **two-valued Boolean algebra $\mathcal{T}$** is the structure that interprets the sort B 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 proper 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 B 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 $\cdot$ 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 5.1 How would you prove BA $\nvdash 0 = 1$?

The symmetric presentation of the axioms of BA in Figure 3 exhibits a prominent duality of BA:

Proposition 5.2 The substitution $\delta = \{0{:=}1,\ 1{:=}0,\ +{:=}\cdot,\ \cdot{:=}+\}$ is a duality of BA that satisfies $\mathbf{S}\delta(\mathrm{BA}) = \mathrm{BA}$ and $\mathbf{S}\delta(\mathbf{S}\delta s) = s$.

We will use the notation $\hat{s} := \mathbf{S}\delta s$ and call $\hat{s}$ the **dual** of s. For equations, the duals $\hat{e}$ are defined analogously. Observe, that BA contains e if and only if it contains $\hat{e}$.

A **Boolean variable** is a variable of type B. A **Boolean parameter** is a constant of type B that are different from 0, 1. In this chapter, we adopt the following conventions:

- x, y, z denote Boolean variables.
- a, b, c denote Boolean parameters.

The set BT of **Boolean terms** is defined recursively as follows:

$$s \in BT \subseteq Ter \ ::= \ x \ | \ a \ | \ 0 \ | \ 1 \ | \ \bar{s} \ | \ s + s \ | \ s \cdot s$$

A **Boolean equation** is an equation $s = t$ where s and t are Boolean terms. Note that every axiom of BA is a Boolean equation. A **tautology** is a Boolean equation that is deducible from BA (i.e., BA $\vdash e$). Two Boolean terms s, t are **equivalent** if $s = t$ is a tautology.

Eventually, we will prove that

$$\mathrm{BA} \vDash e \iff \mathrm{BA} \vdash e \iff \mathcal{B} \vDash e$$

holds for every Boolean equation e and every proper model $\mathcal{B}$ of BA. This surprising result says that BA axiomatizes exactly those Boolean equations that are valid in any non-trivial power set algebra, and also exactly those Boolean equations that are valid in the two-valued Boolean algebra $\mathcal{T}$. Since $\mathcal{T}$ is finite (i.e.,

Algebraic Specifications

BA is a typical example of an algebraic specification. Algebraic specifications make only restricted use of functional types. They have limited expressivity and enjoy special properties. We define algebraic specifications as follows.

An **algebraic constant** is a constant whose type has the form

$$C_1 \to \cdots \to C_n \to C$$

where $n \geq 0$ and $C_1, \dots, C_n$ and C are sorts. In other words, an algebraic constant is a constant that doesn't take functional arguments. An **algebraic variable** is a variable with a non-functional type (i.e., a sort). The set of **algebraic terms** is defined recursively:

1. Every algebraic variable is an algebraic term.
2. If $c : C_1 \to \cdots \to C_n \to C$ is an algebraic constant and $s_1 : C_1, \dots, s_n : C_n$ are algebraic terms, then $cs_1 \dots s_n$ is an algebraic term.

An **algebraic equation** is an equation $s = t$ where s and t are algebraic terms. An **algebraic specification** is a specification whose axioms are algebraic.

only two values for Boolean variables), the equivalences also provide us with an algorithm that decides BA $\vDash e$ and BA $\vdash e$.

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 5.3 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 assocativity axioms are deductive consequences of 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:

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

Idempotence	$xx = x$	$x + x = x$
Dominance	$0x = 0$	$1 + x = 1$
Absorption	$x(x + y) = x$	$x + xy = x$
Negation	$\bar{1} = 0$	$\bar{0} = 1$
de Morgan	$\overline{xy} = \bar{x} + \bar{y}$	$\overline{x + y} = \bar{x}\bar{y}$
Resolution	$xy + \bar{x}z = xy + \bar{x}z + yz$	$(x + y)(\bar{x} + z) = (x + y)(\bar{x} + z)(y + z)$
Involution	$\bar{\bar{x}} = x$	

Figure 4: Some useful tautologies

Let's call this specification HR. It's easy to see that BA $\vdash$ HR (see next section). However, it took until 1996 that William McCune could prove the other direction HR $\vdash$ BA with the help of an automated theorem prover. From this we learn that deciding whether two specifications are equivalent can be extremely difficult.

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

5.2 Boolean Laws

The more tautologies one knows the easier it becomes to deduce new tautologies. Figure 4 collects some useful tautologies that together with the axioms in Figure 3 form a collection of equations we call **Boolean laws**.

This section will show you how the tautologies in Figure 4 can be deduced from the axioms of BA. This way you get familiar with the deductive structure of BA. We start with a conversion proof for BA $\vdash xx = x$:

$$\begin{aligned} xx &= xx + 0 && \text{Identity} \\ &= xx + x\bar{x} && \text{Complement} \\ &= x(x + \bar{x}) && \text{Distributivity} \\ &= x1 && \text{Complement} \\ &= x && \text{Identity} \end{aligned}$$

The proof uses the Commutativity and Assocativity tacitly and mentions the use of the other axioms explicitly. By duality (Proposition 4.15), the proof also shows that $x + x = x$ is a tautology (since $x + x = x$ is the dual of $xx = x$).

Exercise 5.4 Show that Dominance, Absorption, and Resolution are deducible from BA. We offer the following hints:

a) To show $0x = 0$, start with $0x = 0x + 0$, then use complements.

b) To show $x = x(x + y)$, start with $x = x + 0$, then use Dominance and Distributivity.
c) To show $xy + \bar{x}z = xy + \bar{x}z + yz$, start from left and use Absorption in the form of $x = x(x + y)$ and $\bar{x} = \bar{x}(\bar{x} + y)$, then use Idempotence and Complement.

Proposition 5.5 BA, $0{=}1 \vdash x{=}y$

Proof Follows with Identity and Dominance. ∎

The proposition implies that no proper Boolean Algebra equates 0 and 1.

To prove that Negation, de Morgan, and Involution are tautologies, we will employ a notion of deductive equivalence:

$$E \overset{\text{BA}}{\vdash\dashv} E' \;:\Longleftrightarrow\; \text{BA} \cup E \vdash E' \;\wedge\; \text{BA} \cup E' \vdash E \qquad \textbf{deductive equivalence in } \text{BA}$$

Proposition 5.6 (Uniqueness of Complements (UoC))
$ab = 0,\; a + b = 1 \;\overset{\text{BA}}{\vdash\dashv}\; \bar{a} = b$

The proposition is formulated with a notational convenience that omits the curly braces in the official formulation $\{ab = 0,\; a + b = 1\} \overset{\text{BA}}{\vdash\dashv} \{\bar{a} = b\}$.

Exercise 5.7 Find a proof for UoC.

With UoC, we can prove BA $\vdash \bar{s} = t$ by proving BA $\vdash st = 0$ and BA $\vdash s + t = 1$ (by Stability, Proposition 4.13). Thus to prove that the involution law $\bar{\bar{x}} = x$ is a tautology it suffices to show that $\bar{x}x = 0$ and $\bar{x} + x = 1$ are tautologies. This can be done with the complement laws.

Exercise 5.8 Prove that Negation and de Morgan are tautologies (see Figure 4).

Proposition 5.9 (Zero-One (0-1)) If s is a Boolean term that contains neither Boolean variables nor Boolean parameters, then BA $\vdash s = 0$ or BA $\vdash s = 1$.

Proof By induction on $|s|$ with Negation, Identity, Dominance and Commutativity. ∎

We arrange the following notations:

$s \to t$	$\rightsquigarrow$	$\bar{s} + t$	**implication**
$s \leftrightarrow t$	$\rightsquigarrow$	$(s \to t)(t \to s)$	**equivalence**

Note that $\leftrightarrow$ describes the identity function for $\mathbb{B}$ if we take the two-valued Boolean algebra $\mathcal{T}$ as interpretation. To save parentheses, we employ the operator precedence $\cdot \succ + \succ \to \succ \leftrightarrow$.

Equation Systems

We can see $\{ab = 0,\ a + b = 1\}$ as an *equation system* where the parameters a and b take the role of *unknowns*. Because of their generative nature, variables are not suited as unknowns if more than one equation is involved.

A deductive equivalence $E \overset{\text{BA}}{\vdash\dashv} E'$ tells us that the equation systems E and E' have the same solutions for the parameters in a given Boolean algebra. UoC tells us that the system $\{ab = 0,\ a + b = 1\}$ has the same solutions as $\{\bar{a} = b\}$. The equivalences (a) and (c) of Exercise 5.11 give us a method that allows us to transform a finite Boolean equation system E into a single term s such that $E \overset{\text{BA}}{\vdash\dashv} \{s = 1\}$.

Exercise 5.10 Show that the following equations are tautologies:

a) $1 \to x = x, \quad x \to 0 = \bar{x}$

b) $0 \to x = 1, \quad x \to 1 = 1$

c) $x \to x = 1$ reflexivity

d) $x \to y = \bar{y} \to \bar{x}$ contraposition

e) $x = \bar{x} \to 0$ contradiction

f) $xy \to z = x \to y \to z$ Schönfinkel

g) $x + y = (x \to y) \to y$

h) $x \leftrightarrow y = xy + \bar{x}\bar{y}$

Exercise 5.11 Prove the following deductive equivalences. You may use all Boolean laws.

a) $a = 1,\ b = 1 \overset{\text{BA}}{\vdash\dashv} ab = 1$

b) $a \to b = 1 \overset{\text{BA}}{\vdash\dashv} a = ab$

c) $a \leftrightarrow b = 1 \overset{\text{BA}}{\vdash\dashv} a = b$

Equivalences (a) and (c) state interesting properties of Boolean algebras. They say that in a Boolean algebra an equation $s = t$ is equivalent to the "normalized" equation $s \leftrightarrow t = 1$, and that two normalized equations $s = 1$ and $t = 1$ can be combined into the normalized equation $s \cdot t = 1$.