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 $\mathit{plus}\ 3\ 7 = 10$ 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

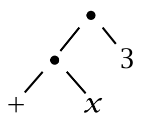
$$T \;::=\; C \;\mid\; T \to T$$

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

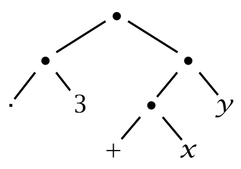


represents a term, that consists of two applications represented as bullets (•), the constants + and 3, and the variable x. The constant + should be thought of as the cascaded representation of the addition operation.

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



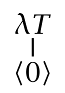
where the constants $\cdot$ and $+$ should be thought of as cascaded representations of the multiplication and addition operations for the underlying number system.

1.4 De Bruijn Representation of Bound Variables

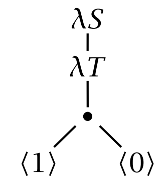
The notations $\lambda x{\in}\mathbb{N}.x$ and $\lambda y{\in}\mathbb{N}.y$ describe the same function. This tells us that the choice of the argument variable of a function description does not matter. Argument variables are merely a notational device for referring to the arguments of functions.

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.

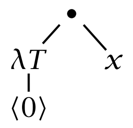
The tree description of a term represents abstractions without using argument variables. Instead, the arguments of abstractions are referred to by so-called **de Bruijn indices** invented by the Dutch logician Nicolaas de Bruijn:



$\lambda x{:}T.x$



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



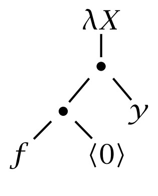
$(\lambda x{:}T.x)x$

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 $\forall x{\in}\mathbb{Z}\colon x \leq y$? 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

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

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

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

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:

$$\doteq_X \in X \to X \to \mathbb{B}$$
$$x \doteq_X y = (x = y)$$

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

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

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

$$\forall x{:}T.\, t \quad\rightsquigarrow\quad \forall_T(\lambda x{:}T.\, t)$$
$$\exists x{:}T.\, t \quad\rightsquigarrow\quad \exists_T(\lambda x{:}T.\, t)$$

Make sure that you understand the following:

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

$$(\lambda x{:}C.y)[y := x] \;=\; \left(\begin{array}{c}\lambda C\\ | \\ y\end{array}\right)[y := x] \;=\; \left(\begin{array}{c}\lambda C\\ | \\ x\end{array}\right) \;=\; \lambda y{:}C.x$$

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.