Theoretical Computer Science, Logic

Zoltán Gábor Molnár

Contents

Introduction

Logic is a formal science which investigates formal arguments and reasoning represented in both formal and natural languages. Hence, one is able to apply to logic C. S. Peirce' threefold categorization of semiotics (theory of signs). Firstly, *syntax* deals with solely the formal, explicit properties of signs. Secondly, *semantics* gives extensions to sings, in a referential sense. So, sentences have propositional values (either the Booleans, or a distinguished truth value or any others), and functors and names have functions and mathematical objects as reference values. The third account is *pragmatics,* which deals with the *use* of signs in communications. These are not only branches of general semiotics. In logic "use" tells us, how to reason about signs and apply them in formal, scientific, one way communications, and what are the situations when we are in a position to apply them in meaningful ways. In computer science and mathematics, syntax and formal pragmatics presuppose constructive, algorithmic, and explicit methods. While, formal semantics deals with models, interpretations, and algebraic structures, mainly within the realms of set theory. This lecture notes consists of the theory of, in a sense, the ultimate logical and programming language called the Typed Lambda Calculus.

1 Data Types and Syntax

1.1 Inductive data types from the set theoretic point of view

In constructive mathematics, the notions of recursion and induction are primitive ones, they are not needed explanation. As Christine Paulin-Mohring had written:

> Intuitively, an inductively defined type is given by a complete list of constructors for terms of the type. We reason about the type with an appropriate induction principle, and we write functions over the type using iteration, which is powerful enough to define primitive recursive functionals over elements of the type. Pfenning and Paulin-Mohring. (1990)

However, it is a standard way of mathematics to set bases—seemed solid—within the framework of set theory. So, the following is a definition of a restricted notion of inductive type, as a set theorist may do it.

Some auxiliary notions.

Signature: $(\tau_n)_{n\in\omega}$ where τ_n's are pairwise disjoint sets (not necessarily non-empty). (ω is the set of finite von Neumann ordinals i.e. $\omega = \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}, \dots\}$).

Arity. When $f \in \tau_n$, then $n = \mathrm{arr}(f)$, the arity of f.

Definition 1.1. The **language** generated by the set X over the signature $(\tau_n)_{n\in\omega}$ (in notation $\langle X, \langle\tau_n\rangle_{n\in\omega}\rangle$) is the *smallest* set E with the properties:

$$X \cup \tau_0 \subseteq E, \qquad f \in \tau_n \wedge (x_1, \dots, x_n) \in E^n \Rightarrow (f, x_1, \dots, x_n) \in E \qquad (n \neq 0).$$

provided that $X \cup \tau_0 \neq \varnothing$ and $X \cap \tau_0 = \varnothing$.

Here, $(x_1, \dots, x_n)$ denotes n element tuple consists of the members $x_1, \dots, x_n$ in this order. Language in this sense is the same as the so called *simple inductive types.*

Example 1.1. a) The signature of the Boolean Algebras (BA's): $\tau_0 = \{\top, \bot\}$, $\tau_1 = \{-\}$, $\tau_2 = \{\wedge, \vee\}$. The *language of the Boolean Algebras* is generated by the infinite set $X = \{A_0, A_1, \dots\}$ over τ. (In case of *Heyting Algebras* (HA's) $\tau_2 = \{\wedge, \vee, \to\}$)
b) The signature of the finite dimensional linear spaces over $\mathbf{R}$: $\tau_0 = \{0\}$, $\tau_1 = \{-, \lambda_r\}_{r\in\mathbf{R}}$, $\tau_2 = \{+\}$. The *language of the finite dimensional linear spaces over* $\mathbf{R}$ is generated by the finite set $X = \{e_1, \dots, e_m\}$ over τ.
c) The signature of the type of natural numbers (nat), as an inductive type, is $\tau_0 = \{0\}$, $\tau_1 = \{S\}$. The *language of nat* is generated by the empty set $X = \{\}$ over τ. Note that over nat, we can define functions by the well-known ordinary recursion.

Theorem 1.1. The $\langle X, \langle\tau_n\rangle_{n\in\omega}\rangle$ always exists.

Proof. Let $(H_k)_{k\in\omega}$ be the following sequence, defined by recursion:

$$H_0 := X \cup \tau_0$$
$$H_{k+1} := H_k \cup \{(f, x_1, \dots, x_n) \mid n \in \omega \setminus \{0\}, f \in \tau_n, x_1, \dots, x_n \in H_k\}$$

Then (i) $H := \bigcup_{k\in\omega} H_k$ is an E above and (ii) for every E above, $H \subseteq E$.

(i) Note that, $\{H_k\}_{k\in\omega}$ is a chain with respect to the partial order $\subseteq$. $X \cup \tau_0 \subseteq H$ is trivial. Suppose $x_1, \dots, x_n \in H$ and $f \in \tau_n$. Then there is a $K \in \omega$ such that $x_1, \dots, x_n \in H_K$, hence $(f, x_1, \dots, x_n) \in H_K \subseteq H$.
(ii) It is enough to show that for all k, $H_k \subseteq E$, because, if $e \in \bigcup_{k\in\omega} H_k$, then $e \in H_K$ for a K, so $e \in E$ too. By mathematical induction, first we know that $H_0 = X \cup \tau_0 \subseteq E$. On the other hand, suppose $H_k \subseteq E$. Then, if $e \in H_{k+1}$, then wlog we assume that $e = (f, x_1, \dots, x_n)$, where $f \in \tau_n, x_1, \dots, x_n \in H_k \subseteq E$. So, by the inductive rules of E, $(f, x_1, \dots, x_n) \in E$, i.e. $H_{k+1} \subseteq E$.
Therefore, if the language is L, then $H \subseteq L$, hence $L = H$. □

Theorem 1.2. Let $H = \langle X, \langle\tau_n\rangle_{n\in\omega}\rangle$. Then for every $e \in H$ there is a unique $k \in \omega$ such that $e \in H_k \setminus H_{k-1}$ (with H_k above and $(H_{-1} = \varnothing)$).

Proof. We will prove that for any $K \in \omega$, for every $e \in H_K$ there is a unique $k \leq K$ such that $e \in H_k \setminus H_{k-1}$. The initial case is trivial. Suppose, the proposition is true for K. For $f(e_1, \ldots, e_n)$ wlog suppose the maximum of k_i's such that $e_i \in H_{k_i} \setminus H_{k_i-1}$ is K. Hence, $f(e_1, \ldots, e_n) \in H_{K+1}$ but not in H_K, since otherwise the maximum of k_i's would not be K. The same last argument shows that the proposition holds for L. □

When $e \in H_k \setminus H_{k-1}$, then k is the *depth* of e.

The following theorem allow us to define functions over languages (over simple inductive types). This kind of definition is the *primitive recursive definition.*

Theorem 1.3. If $H = \langle X, \langle \tau_n \rangle_{n\in\omega} \rangle$, $R \neq \varnothing$, $G_0 : X \cup \tau_0 \to R$, $G \in \prod_{f\in\cup_{n\in\omega\setminus\{0\}}\tau_n} R^{R^{\mathrm{ar}(f)}}$. Then there exists uniquely the function $F : H \to R$ such that

$$F(e) = G_0(e),$$
$$F((f, e_1, \ldots, e_{\mathrm{ar(f)}})) = G_f(F(e_1), \ldots, F(e_{\mathrm{ar(f)}}))$$

where $f \in \cup_{n\in\omega\setminus\{0\}}\tau_n$, $e_1, \ldots, e_{\mathrm{arr(f)}} \in H$, $e \in X \cup \tau_0$.

Remark 1.1. (i) $B^A := \{f : A \to B\}$ $\quad (= A \to B)$ (independent function space)

(ii) $\prod_{i\in A} B_i = \{f : A \to \bigcup_{i\in A} B_i \mid \forall i \in A : f(i) \in B_i\}$ (dependent function space, or product set).

(iii) Let C be a nonempty set of sets and A a set. If f is an $A \to C$ function, and f is fixed, then f sometimes is denoted by $f = (B_i)_{i\in A}$ where $B_i = f(i)$, for $i \in A$.

Proof. Let H be the set defined recursively above. F is defined appropriately on the set

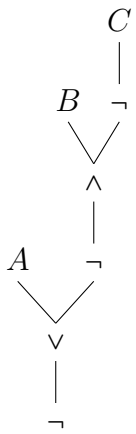
$$H_0 \cup \bigcup_{k\in\omega} H_{k+1} \setminus H_k$$

recursively, i.e. $F(e) := F_0(e)$, $e \in X \cup \tau_0$ and $F((f, e_1, \ldots, e_n)) := G_f(F(e_1), \ldots, F(e_{\mathrm{ar}(f)}))$, $(f, (e_1, \ldots, e_{\mathrm{ar}(f)})) \in H_{k+1} \setminus H_k$, $k \in \omega$. □

The element of H (i.e. $\langle X, \langle \tau_n \rangle_{n\in\omega} \rangle$) are *labeled finitary planar trees.* When the last n such that $\tau_n \neq \varnothing$ exists, then these trees are n-ary trees; in case of $n = 2$, they are binary trees.

Definition 1.2. The element of H above are the *Abstract Syntax Trees* generated by X with respect to the signature $(\tau_n)_{n\in\omega}$.

Example 1.2. In the language of BA's, here is an AST:



In *linear style,* the code of the above tree is:

$$(\neg, (\vee, A, (\neg, (\wedge, B, (\neg, C))))).$$

When we keep in mind, what are the arities of the operators, and erase the brackets and comas, we get the *prefix notation:*

$$\neg \vee A \neg \wedge B \neg C.$$

It is very easy to convert the prefix notation into infix notation:

$$\neg(A \vee (\neg(B \wedge (\neg C)))).$$

Remark 1.2. The prefix and infix notations can be considered as list type of data, so they are *lists,* while AST's are trees. As we've seen, trees are a kind of list of list of list of ..., however, we will strictly distinguish them from each other. Also an important fact is that, trees here are always *ordered or planar* trees, so not only trees in the sense of graph theory.

Remark 1.3. Note, that the Latex code of the AST above mimics the linear style a.k.a. the structure of the labeled tree datatype:

```
\begin{tikzpicture}[grow'=up]
\tikzset{frontier/.style={distance from root=200pt}}
\Tree [.$\neg$ [.$\vee$ [.$A$ ] [.$\neg$ [.$\wedge$ [.$B$ ] [.$\neg$ [.$C$ ] ] ] ] ]
\end{tikzpicture}
```

$$(\neg, (\vee, A, (\neg, (\wedge, B, (\neg, C))))).$$

1.2 Category (Realm of Kittens)

To get rid of the nuisances due to the representation of the syntax, it is worth looking for a categorical definition. But first, let us define the notion of category. It will be way more abstract, however easy in some sense.

> Category theory is extreme in the sense that it actively discourages us from looking inside the objects. An object in category theory is an abstract nebulous entity. All you can ever know about it is how it relates to other objects — how it connects with them using arrows. [...] The moment you have to dig into the implementation of the object in order to understand how to compose it with other objects, you've lost the advantages of your programming paradigm. Milewski (2019)

Every definition in category theory can be divided into two parts. The *declaration part,* in which we fix the things we manipulates and the *computational part,* in which we set the rules describing how to manipulate the things we declared.

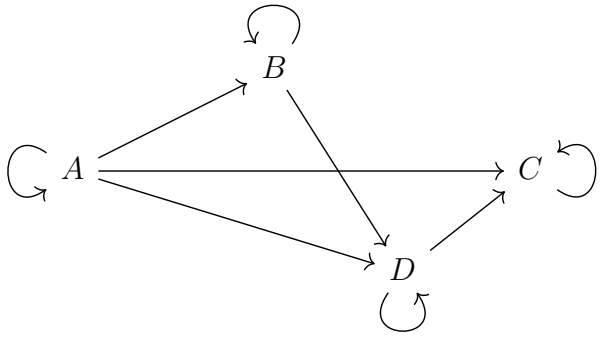
Definition 1.3. A category C consists of the following things:

- $\mathrm{Ob}_C : \mathrm{Class}$ (class of all objects)
- $\mathrm{UHom}_C : \mathrm{Class}$ (class of all morphism or arrows)
- $\mathrm{Hom} : \mathrm{Ob}_C \to \mathrm{Ob}_C \to \mathrm{UHom}_C$ ($\mathrm{Hom}(a,b)$ is the class of arrows from a to b, if $f : \mathrm{Hom}(a,b)$, then $f : a \to b$)
- $\mathrm{dom}, \mathrm{codom} : \prod_{a,b\,:\,\mathrm{Ob}_C} \mathrm{Hom}(a,b) \to \mathrm{Ob}_C$
- $\mathrm{id} : \prod_{a\,:\,\mathrm{Ob}_C} \mathrm{Hom}(a,a)$ (identity morphism)
- $\mathrm{comp} : \prod_{a,b,c\,:\,\mathrm{Ob}_C} \mathrm{Hom}(a,b) \to \mathrm{Hom}(b,c) \to \mathrm{Hom}(a,c)$ (composition ($g : a \to b$, $f : b \to c$, then $f \circ g : a \to c$))

with the rules

$$\mathrm{dom}(a,b,f) = a, \quad \mathrm{codom}(a,b,f) = b, \quad f \circ \mathrm{id} = f, \quad \mathrm{id} \circ g = g, \quad f \circ (g \circ h) = (f \circ g) \circ h.$$

Example 1.3. a) *Directed graphs* provided that they possess all the loops and the edges from $e : A \to C$ when there are the edges $e_1 : A \to B$ and $e_2 : B \to C$. Here $\mathrm{Ob}(C) =$ Vertices, $\mathrm{UHom}(C) =$ Edges.



b) What is equivalent: *pre-ordered sets* (posets), in their directed graph representations:

$$a \leq b \qquad \Longleftrightarrow \qquad \exists f : a \to b$$

(Binary relations over a set, where the relation is reflexive and transitive.)
c) Associative algebraic structure $(M, +)$ with identity element,

$$\forall a, b, c \in M \qquad (a + b) + c = a + (b + c), \qquad a + 0 = 0 + a = a$$

that is *monoids*. Here $\mathrm{Ob}(C) = \{*\}$ (a singleton) and $M = \mathrm{UHom} = \mathrm{Hom}(*, *)$.

For us one of the most frequently applied notion is the initial object in a category.

Definition 1.4. The object I is *initial* in the category C, if for every object A in C, there is a unique morphism $u : I \to A$.

$$I \xrightarrow{u} A$$

(The *dual* is *terminal* $T : C$, that is, if for every $A : C$, there is a unique morphism $v : A \to T$.

$$A \xrightarrow{v} T$$

)

Example 1.4. a) In Set (objects: sets, morphisms: functions), the empty set have the prperty that

$$\varnothing = u : \varnothing \to S$$

is unique, and for every set S, the function

$$v : S \to \{*\}$$

is unique. The singletons are terminal objects, the empty set is the initial object. Both are based on fortiori arguments.
b) Initial object in a poset $(P, \leq)$ is the least element m

$$\forall a \in P \qquad m \leq a$$

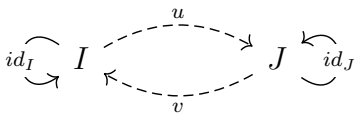
c) In Grp the trivial group

$$\{0\}$$

is both initial and terminal object (it is called a *zero object*). The latter is trivial, the former is due to the fact that the only group-morphism which maps $\{0\}$ to a group G is the $0 \mapsto 0$.

Definition 1.5. The morphism $f : X \to Y$ is isomorphism from X to Y, if there is a morphism $g : Y \to X$ such that $f \circ g = id_Y$ and $g \circ f = id_X$. In this case we write $X \overset{f}{\cong} Y$ or if the morphism is known, $X \cong Y$.

Remark 1.4. The initial object in a category is *unique up to isomorphism.*



If I and J are initial objects as well, then id_I and id_J are unique, hence $u \circ v = id_J$ and $v \circ u = id_I$. What means that u is an isomorphism.

Now, we turn to a "hybrid" definition of syntax, still incorporated the universal algebraic notion of algebraic structure.

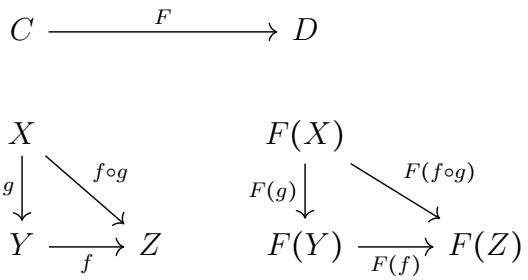
Definition 1.6. The *syntax* generated by the set X over the signature τ is the initial object of the category of all algebras generated by the set X (also called variables) over the signature τ. (Also called anarchic algebra or term algebra.)

The pure categorical syntax definition is based on the notion of F-algebras. But first, let us define functors. Functors are homomorphisms between categories.

Definition 1.7. Let C, D be categories. A functor $F : C \to D$ is a pair (F_O, F_H) of mappings $F_O : \mathrm{Ob}(C) \to \mathrm{Ob}(D)$ and $F_M : \mathrm{UHom}(C) \to \mathrm{UHom}(D)$ (indexes are not used further) such that

$$F(id_X) = id_{F(X)}$$
$$F(f \circ g) = F(f) \circ F(g)$$

$X, Y, Z : \mathrm{Ob}(C), g : X \to Y, f : Y \to Z$.
Hence, the following diagram commutes.



There are incredibly many functors, from power set functor to dual functor of linear spaces.

Definition 1.8. 1. If C is a category, and $F : C \to C$ is an endofunctor of C, then an *F F-algebra* is a pair (A, α), where $A : \mathrm{Ob}(C)$ and

$$\alpha : F(A) \to A$$

(A is called the *carrier* of the F-algebra).

2. The *category of F-algebras* in C consists of the the F-algebras as objects and morphisms $f : A \to B$ such that the following diagram commutes.

$$\begin{array}{ccc} F(A) & \xrightarrow{\alpha} & A \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle f} \\ F(B) & \xrightarrow[\beta]{} & B \end{array}$$

Example 1.5. a) If C has a terminal object 1, then by the functor

$$F(X) = 1$$

with the morphism

$$\alpha : 1 \to X$$

$(1, \alpha : 1 \to X)$ is an F-algebra (1 is the date type True in typed programming language). Especially, in Set, $1 = \{*\}$, and $\alpha = \{(*, x)\}$.
b) *Sum object* (Motivation: $F(X) = 1 \coprod 1$ the bi-pointed object, which is the data type Boole in the typed programming language.) In Set, for all set A, B, the set

$$A \coprod B = \{(n, x) \mid (n = 1 \wedge x \in A) \vee (n = 2 \wedge x \in B)\}$$

and the functions $l : A \to A \coprod B, a \mapsto (1, a)$ and $r : B \to A \coprod B, b \mapsto (2, b)$ possess the property that for any set C and functions $l' : A \to C$, $r' : B \to C$ there exists a unique function $u : A \coprod B \to C$ such that the following diagram commutes

$$\begin{array}{ccccc} A & \xrightarrow{l} & A \coprod B & \xleftarrow{r} & B \\ & {\scriptstyle l'}\searrow & {\scriptstyle u}\downarrow & \swarrow{\scriptstyle r'} & \\ & & C & & \end{array}$$

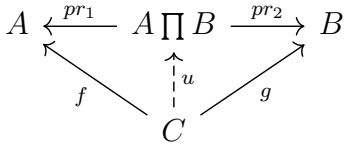
$u \circ l = l' \qquad u \circ r = r'$

Since, $u(x) = \begin{cases} l'(a) & x = (1, a) \\ r'(b) & x = (2, b) \end{cases}$ uniquely defined.

c) *Product object* (Motivation: $F(X) = 1 \coprod (X \prod X)$ is the Tree data type in typed programming languages.) In Set, for all set A, B, the set

$$A \prod B = \{(a, b) \mid a \in A \wedge b \in B\}$$

and the functions $pr_1 : A \prod B \to A, (a, b) \mapsto a$ and $pr_2 : A \prod B \to B, (a, b) \mapsto b$ possess the property that for any set C and functions $f : C \to A$, $g : C \to B$ there exists a unique function $u : C \to A \prod B$ such that the following diagram commutes



$u(x) = (f(x), g(x))$

Definition 1.9. Let F be an endofunctor in the category C. The *syntax* of the signature F is the initial algebra (object)

$$(X, \alpha : F(X) \to X)$$

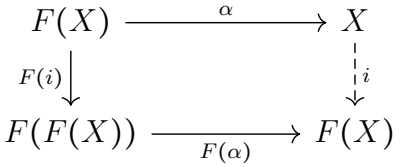
in the category of F-algebras of F in C. (Here α is called the system of constructors of X.)

Theorem 1.4 (Lambek's Theorem)**.** If $(X, \alpha : F(X) \to X)$ is the initial F-algebra of the endofunctor F, then α is an isomorphism from $F(X)$ to X:

$$F(X) \cong X.$$

Hence, X is the fixpoint of F.

Proof. Let $I = (X, \alpha : F(X) \to X)$ be the initial F-algebra. Let us consider the F-algebra $(F(X), F(\alpha) : F(F(X)) \to F(X))$. I is initial, so there is a unique $i : X \to F(X)$ such the following diagram commutes:



Now,

$$\alpha \circ i = id_X, \quad (*)$$

since id_X and $\alpha \circ i$ are both $X \to X$ morphisms and F-algebras, and X is initial. Hence, by (*) functority and F-algebra morphism properties:

$$id_{F(X)} = F(id_X) = F(\alpha \circ i) = F(\alpha) \circ F(i) = i \circ \alpha$$

□

1.3 Some applications

However, we've already known that what is a product and a sum in the sense of category theory, we are going to define explicitely the notions of categories closed under product and coproduct. This will be elaborated by the so called *equational* presentation which makes the *computational rules* of the definitions apparent.

> [T]he equational presentation of cartesian closed categories and other structured categories had already been emphasized by Lambek, who regarded them as certain kinds of deductive systems with an equivalence relation between proofs. Prawitz had also studied an equivalence relation between proofs in intuitionistic logic for completely different motives. It was shown by Mann and rediscovered by Seely that, for intuitionistic propositional calculus, the two equivalence relations are essentially the same. This confirms our view that category theory may serve as useful motivation for many constructions in logic. Lambek and Scott (1986)

Definition 1.10. A *bi-cartesian* cartegory is the following.

1. For all objects X_i $(i = 1, 2)$ there is an object $X_1 \prod X_2$, there are morphisms $pr_i : X_1 \prod X_2 \to X_i$ (the canonical projections), and for all object Z and morphisms $f_i : Z \to X_i$ there is a morphism $f_1 \prod f_2 : Z \to X_1 \prod X_2$ such that for all $g : Z \to X_1 \prod X_2$

   i. $pr_i \circ (f_1 \prod f_2) = f_i \quad (i = 1, 2)$

   ii. $(pr_1 \circ g) \prod (pr_2 \circ g) = g.$

   $$\left[\begin{array}{ccccc} X_1 & \xleftarrow{pr_1} & X_1 \times X_2 & \xrightarrow{pr_2} & X_2 \\ & {\scriptstyle f_1} \nwarrow & \uparrow {\scriptstyle f_1 \prod f_2} & \nearrow {\scriptstyle f_2} & \\ & & Z & & \end{array}\right]$$

2. For all objects X_i $(i = 1, 2)$ there is an object $X_1 \coprod X_2$, there are morphisms $in_i : X_i \to X_1 \coprod X_2$ (the canonical inclusions), and for all object Z and morphisms $f_i : X_i \to Z$ there is a morphism $f_1 \coprod f_2 : X_1 \coprod X_2 \to Z$ such that for all $g : X_1 \coprod X_2 \to Z$

   i. $(f_1 \coprod f_2) \circ in_i = f_i \quad (i = 1, 2)$

   ii. $(g \circ in_1) \coprod (g \circ in_2) = g.$

   $$\left[\begin{array}{ccccc} X_1 & \xrightarrow{in_1} & X_1 + X_2 & \xleftarrow{in_2} & X_2 \\ & {\scriptstyle f_1} \searrow & \downarrow {\scriptstyle f_1 \coprod f_2} & \swarrow {\scriptstyle f_2} & \\ & & Z & & \end{array}\right]$$

Remark 1.5. The definition above is equivalent to the definition of product (and coproduct). The *commutative property* follows from equation (i), since it is literally the statement that the left and right triangles commute.

On the other hand, equation (ii) implicates the *uniqueness* of the product map (and the coproduct map too). Let $g : Z \to X_1 \prod X_2$ be such that $pr_i \circ g = f_i$. Then

$$g = (pr_1 \circ g) \prod (pr_2 \circ g) = f_1 \prod f_2$$

Remark 1.6. Equation (ii) also claims that all arrows $g : Z \to X_1 \prod X_2$, mapping to the product, themselves are always product maps, in other words, they always decompose into a product map of two morphisms:

$$g = g_1 \prod g_2.$$

Remark 1.7. What's more, equation (ii) also claims that an arrow mapping to a product is always decomposed into a *unique* product map. Let $g : Z \to X_1 \prod X_2$. Suppose, there is an other pair f_i with the product map $f_1 \prod f_2$, that is, $f_i = pr_i \circ f_1 \prod f_2$. Then by a fortiori, if $g = f_1 \prod f_2$, then

$$f_i = pr_i \circ g.$$

Remark 1.8. If X and Y are objects, then sometimes $X \prod Y$ is denoted by $X \times Y$ and $X \coprod Y$ is denoted by $X + Y$.

Example 1.6 (Component-wise Composition Lemma)**.** Let X, Y, Z, W objects in a bi-cartesian category, $f_1 : X \to Z$, $f_1 : Y \to Z$ and $h : Z \to W$. Then

$$h \circ (f_1 \coprod f_2) = (h \circ f_1) \coprod (h \circ f_2).$$

(The composition distributes over the sum, from the left.)

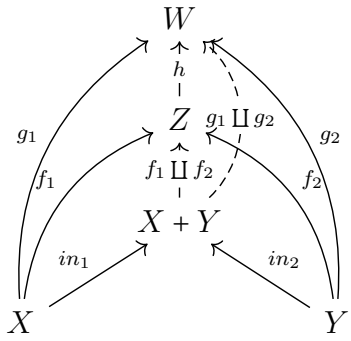
Proof. So, we know that $f_1 \coprod f_2 : X + Y \to Z$. Let

$$g_1 \coloneqq h \circ f_1 \text{ and } g_2 \coloneqq h \circ f_2,$$

hence $g_1 : X \to W$, $g_2 : Y \to W$ and $g_1 \coprod g_2 : X + Y \to W$. So, we have to show that

$$\boxed{h \circ (f_1 \coprod f_2) = (g_1 \coprod g_2)}.$$

Notice that, $h \circ (f_1 \coprod f_2)$ satisfies the commutative properties of the definition of $g_1 \coprod g_2$.



By the definition of g_i and the definition of the $f_1 \coprod f_2$:

$$g_i = h \circ f_i = h \circ (f_1 \coprod f_2) \circ in_i$$

But, such a morphism, which is here $h \circ (f_1 \coprod f_2)$, is unique by the definition of $g_1 \coprod g_2$, hence $h \circ (f_1 \coprod f_2) = g_1 \coprod g_2$. □

Remark 1.9. Let us rewrite the notion of categories of F-algebras. In the categories of F-algebras in category C, the objects are pairs: $(X, \alpha : F(X) \to X)$ and the morphisms are $f : X \to Y$ such that

$$\begin{array}{ccc} F(X) & \xrightarrow{\alpha} & X \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle f} \\ F(Y) & \xrightarrow[\beta]{} & Y \end{array}$$

commutes.

1.3.1 Natural number data type

Definition 1.11. In a category with coproduct and with terminal object (1), natural number object is a tuple $(N, 0 : 1 \to N, s : N \to N)$ such that for every object M and morphisms: $z : 1 \to M, f : M \to M$ there is a unique morphism $u : N \to M$ such that the

$$\begin{array}{ccccc} 1 & \xrightarrow{0} & N & \xrightarrow{s} & N \\ & \searrow^{z} & \downarrow{\scriptstyle u} & & \downarrow{\scriptstyle u} \\ & & M & \xrightarrow{f} & M \end{array}$$

diagram commutes.

Theorem 1.5. In a category with coproduct and terminal object, the initial algebra of the functor

$$F(X) = 1 + X$$

is the natural number object (up to isomorphism).

Proof. An F-algebra of F is a pair $(X, \alpha : 1 + X \to X)$. The F above is not a complete functor, since it is defined only over the objects, so the solution must contain the definition of the complete functor itself.

Hint and motivation: when you intend to find a morphism, "the first one, which you have in mind" is the needed one. In the present case, you have to build an appropriate morphism from $1 + X$ to $1 + Y$ and you have a morphism $f : X \to Y$. So, $F(f)$ is defined on a sum, that is, $F(f)$ is itself a coproduct $g_1 \coprod g_2$ composed of the one from 1 (the g_1)

and an other from X (it is g_2). Now, $F(f)$ maps into the object $1+Y$, so g_1 must map 1 into $1+Y$ and g_2 must map X into Y. What we have in mind about the first one (g_1) is the canonical inclusion map in_1^{1+Y} the second one (g_2) is first of all the f from X to Y, then the canonical inclusion map in_2^{1+Y}. Hence, let

$$F(f) := in_1^{1+Y} \coprod (in_2^{1+Y} \circ f).$$

F is a functor, since
1)

$$F(id_X) = in_1^{1+X} \coprod (in_2^{1+X} \circ id_X) = in_1^{1+X} \coprod in_2^{1+X} = (id_{1+X} \circ in_1^{1+X}) \coprod (id_{1+X} \circ in_2^{1+X} \circ id_{1+X}) \overset{ii}{=} id_{1+X}$$

2) We have to show that, if $g : X \to Y$, $f : Y \to Z$, then

$$(in_1^{1+Z} \coprod (in_2^{1+Z} \circ f)) \circ (in_1^{1+Y} \coprod (in_2^{1+Y} \circ g)) = in_1^{1+Z} \coprod (in_2^{1+Z} \circ f \circ g).$$

Indeed, it is enough to show this by component-wise:

$$(in_1^{1+Z} \coprod (in_2^{1+Z} \circ f)) \circ in_1^{1+Y} = in_1^{1+Z}$$

and

$$(in_1^{1+Z} \coprod (in_2^{1+Z} \circ f)) \circ (in_2^{1+Y} \circ g) = in_2^{1+Z} \circ f \circ g.$$

But, these hold by the *in* equations (i in the definition), and in the case of equation two furthermore by the associative property of $\circ$. (When you apply in_1 from the left to a coproduct, the result is going to be first component, and the other way around.)

By the definiton of coproduct with equations (equation (ii)), there are unique morphisms $0_X : 1 \to X$ and $s_X : X \to X$ such that $\alpha = 0_X \coprod s_X$. Let's suppose that $(X, 0_X \coprod s_X)$ is initial. Then for all $(Y, 0_Y \coprod s_Y)$ F-algebra there is a unique morphisms such that the

$$\begin{array}{ccc} 1+X & \xrightarrow{0_X \coprod s_X} & X \\ {\scriptstyle in_1^{1+Y} \coprod in_2^{1+Y} \circ u}\downarrow & & \vdots\,{\scriptstyle u} \\ 1+Y & \xrightarrow[0_Y \coprod s_Y]{} & Y \end{array}$$

diagram commutes. It means that

$$u \circ (0_X \coprod s_X) = (0_Y \coprod s_Y) \circ (in_1^{1+Y} \coprod in_2^{1+Y} \circ u)$$

Then after circling them from the right by in_i^{1+X}:

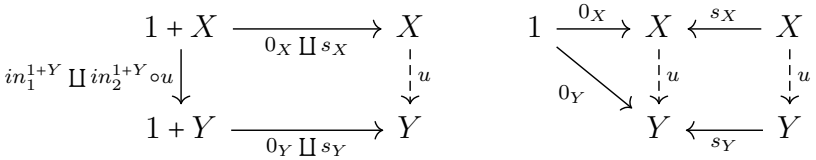
$$u \circ 0_X = (0_Y \coprod s_Y) \circ (in_1^{1+Y} \coprod in_2^{1+Y} \circ u) \circ in_1^{1+X}$$

$$u \circ s_X = (0_Y \coprod s_Y) \circ (in_1^{1+Y} \coprod in_2^{1+Y} \circ u) \circ in_2^{1+X}$$

Then by applying the composition lemma in left circles, and again applying the inclusion rules

$$u \circ 0_X = (0_Y \coprod s_Y \circ u) \circ in_1^{1+X} = 0_Y$$
$$u \circ s_X = (0_Y \coprod s_Y \circ u) \circ in_2^{1+X} = s_Y \circ u$$

Then the above diagram decomposes into that of the natural number object case:



$$u \circ 0_X = 0_Y \text{ and } u \circ s_X = s_Y \circ u$$

□

1.3.2 Truth values, Booleans, list, trees

True	$F(X) = 1$
False	$F(X) =$
Booleans	$F(X) = 1 + 1$
nat	$F(X) = 1 + X$
list of A's	$F(X) = 1 + A \times X$
binary labeled trees	$F(X) = A + (A \times X)^2$

1.4 Examples, motivation

1.5 Examples

Note that, by duality

$$(f_1 \prod f_2) \circ h = (f_1 \circ h) \prod (f_2 \circ h)$$

that is: *composition distributes over the product, from the right.*

Example 1.7. In a category with product

$$\boxed{X \times Y \cong Y \times X}$$

Proof. We have to show that there are morphisms $i : X \times Y \to Y \times X$ and $j : Y \times X \to X \times Y$ such that

$$i \circ j = id_{X \times Y}$$

(it is enough, by symmetry.) Note that the choice

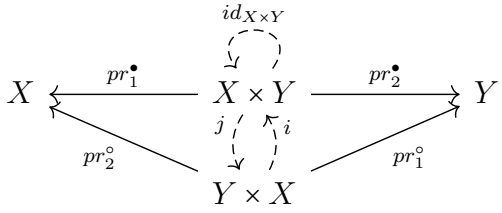
$$i = pr_2^\circ \prod pr_1^\circ : Y \times X \to X \times Y$$

$$j = pr_2^\bullet \prod pr_1^\bullet : X \times Y \to Y \times X$$

works, where $^\circ$ denotes the projections with respect to object $X \times Y$ and $^\bullet$ denotes the projections with respect to object $Y \times X$.
Solution 1—with computational rules (equations).

$$\begin{aligned}(pr_2^\circ \prod pr_1^\circ) \circ (pr_2^\bullet \prod pr_1^\bullet) &\overset{\text{comp.}}{=} (pr_2^\circ \circ (pr_2^\bullet \prod pr_1^\bullet)) \prod (pr_1^\circ \circ (pr_2^\bullet \prod pr_1^\bullet)) = \\ &\overset{\text{i}}{=} pr_1^\bullet \prod pr_2^\bullet = \\ &= (pr_1^\bullet \circ id_{X\times Y}) \prod (pr_2^\bullet \circ id_{X\times Y}) = \\ &\overset{\text{ii}}{=} id_{X\times Y}\end{aligned}$$

Solution 2—with diagrams.



Using commutative triangles. We show that $i \circ j$ satisfies the property of the product map of $X \times Y$.

$$(pr_1^\bullet \circ i) \circ j = pr_2^\circ \circ j = pr_1^\bullet$$
$$(pr_2^\bullet \circ i) \circ j = pr_1^\circ \circ j = pr_2^\bullet$$

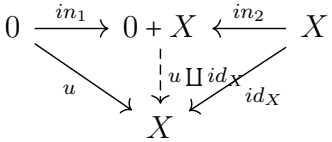
The first computation is follows from the definition of $X \times Y$, the second one is from the definition of $Y \times X$. Since, there is only one $u : X \times Y \to X \times Y$ such that $pr_1^\bullet \circ u = pr_1^\bullet$ and $pr_2^\bullet \circ u = pr_2^\bullet$, then $u = id_{X\times Y} = i \circ j$. □

Example 1.8. If 0 is the initial object, then

$$\boxed{0 + X \cong X}$$

Proof. Clearly,

$$i = u \coprod id_X : 0 + X \to X, \qquad j = in_2 : X \to 0 + X$$



By equations.

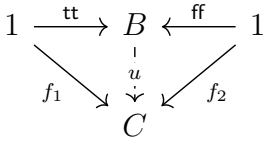
$$\begin{aligned} in_2 \circ (u \coprod id_X) &\overset{\text{comp.}}{=} (in_2 \circ u) \coprod (in_2 \circ id_X) = \\ &\overset{id}{=} (in_2 \circ u) \coprod in_2 = \\ &\overset{\text{init.}}{=} in_1 \coprod in_2 = \\ &\overset{\text{id}}{=} (id_{0+X} \circ in_1) \coprod (id_{0+X} \circ in_2) = \\ &\overset{\text{ii}}{=} id_{0+X}. \\ (u \coprod id_X) \circ in_2 &\overset{\text{i}}{=} id_X \end{aligned}$$

By commutative diagram. $(u \coprod id_X) \circ in_2 = id_X$ holds by the right triangle. Clearly $pr_1 \coprod pr_2 = id_{0+X}$, and

$$in_2 \circ (u \coprod id_X) \circ in_1 = in_2 \circ u = in_1 : 0 \to 0 + X$$

by uniqueness of maps from initial object. And by uniqueness of coproduct it is $in_2 \circ (u \coprod id_X) = id_{0+X}$. □

Example 1.9. Show that $F = 1 + 1$ is a functor. Let us define the notion of bi-pointed object B in a bi-cartesian category with terminal object 1 by the following commuting diagram.



Show that the initial F-algebra over $F(X) = 1 + 1$ is the bi-pointed object. (tt is the constructor 'true', and ff is the constructor 'false'.)

Proof. $F(X)$ is a constant map, however the morphism part is not defined, so the best choice is $F(f) \coloneqq id_{1+1}$. F is indeed a functor, since by its own definition:

$$F(id_X) = id_{1+1} = id_{F(X)}$$

and

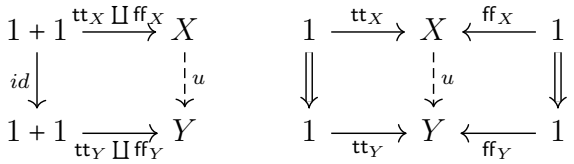
$$F(f \circ g) = id_{1+1} = id_{1+1} \circ id_{1+1} = F(f) \circ F(g)$$

Then, the coproduct can be divided into two components.

$$(\mathsf{tt}_Y \coprod \mathsf{ff}_Y) \circ in_1 = \mathsf{tt}_Y$$

$$(u \circ (\mathsf{tt}_X \coprod \mathsf{ff}_X)) \circ in_1 = (u \circ \mathsf{tt}_X) \coprod (u \circ \mathsf{ff}_X) \circ in_1 = u \circ \mathsf{tt}_X$$

And, since $\mathsf{tt}_Y \coprod \mathsf{ff}_Y = u \circ (\mathsf{tt}_X \coprod \mathsf{ff}_X$, then $\mathsf{tt}_Y = u \circ \mathsf{tt}_X$

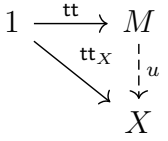


□

HW 1.1. Suppose, $\mathcal{C}$ is a category with co-product. Show that $X + Y$ is isomorphic to $Y + X$.

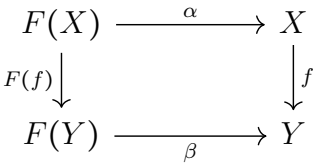
HW 1.2. Suppose, $\mathcal{C}$ is a category with product and terminal object 1. Show that $1 \times X$ is isomorphic to X.

HW 1.3. Suppose, $\mathcal{C}$ is a category with terminal object 1. Let M be the one-pointed object with inclusion $\mathsf{tt} : 1 \to M$ such that for all X and tt_X there is a unique u such that



commutes. Show that $F(X) = 1$ is an endofunctor. Show that the initial algebra of the endofunctor F is the one-pointed object. Is it true, that M is isomorphic to 1?

HW 1.4. Prove that, if F is an endofunctor of the category $\mathcal{C}$, then the F-algebras form a category where the objects are the F-algebras and the morphisms are the f-s such that



commutes.

1.6 Summary

2 Proof Theory without Proof Codes

> Mathematics is regarded as the most certain of all the sciences. That it could lead to results which contradict one another seems impossible. This faith in the indubitable certainty of mathematical proofs was sadly shaken around 1900 by the discovery of the 'antinomies' (or 'paradoxes') of set theory. It turned out that in this specialized branch of mathematics contradictions arise without our being able to recognize any specific error in our reasoning.(Gentzen, 1969, p. 132)

> It is, however, important to observe that no appeal has been made to the principle of consistency, and that the logical laws do not imply it. We may know our language to be such that not every atomic statement can be true; but logic does not know that. As far as it is concerned, they might form a consistent set, as they are assumed to do in Wittgenstein's *Tractatus.* The principle of consistency is not a logical principle: logic does not require it, and no logical laws could be framed that would entail it. (Dummett, 1991, p. 295)

2.1 Introduction

First of all, the language of *judgments* must be defined. There are two ways to do it. The first one is when the provability relation is a binary one

$$\Gamma \vdash A$$

here Γ is a set of sentences, and A is a sentence. Intuitively, this means that "the sentence A is provable from the set of sentences Γ". This is an implicit way, since the structure of the proof is not revealed. The ternary relation

$$\Gamma \vdash \Pi : A$$

where Π is the proof tree or proof code makes the proof transparent in the notation, and allows us to state explicit theorems about proofs. Before we turn to the latter one, let us do some proof theory in the binary approach.

It is important to note that, the way of presentation above is neither the first historically, nor the way non-proof-theorists or non-computer-scientists define it. The first and common definition of formal proofs is the sequential one, what is known as the Hilbert style definition. In Hilbert's style there are axioms, and only one deduction rule which is the so called modus ponens

$$\frac{A \to B \qquad A}{B}$$

that is, if we know that $A \to B$ is true and we know that A is also true, then B is true too.[1] The problem with axioms, is that they do not reflect the computational characteristics of the objects they talk about. In computer science, data are presented in a recursive or algorithmic way, hence, they already have computational computational characteristics at the beginning. However, singular axioms are just declarations of features, without any constructive or computational meaning. The interesting fact is that one of the most important structure, the natural numbers have these properties. Nat is an inductive

[1] In first order logic, where quantifiers are allowed, there is a second deduction rule, the universal generalization: when $A(x)$ is true for an arbitrary individual x, then $A(x)$ is true for all individuals.

data type with constructors and we can reason about nat inductively and we can define functions on it recursively, hence nat has full computational description.

The sort of propositions behaves similar to nat. When a data type is of the sort proposition, it can be defined in the context of computation. The *introduction rules* generate the inhabitants of that type and the *elimination rules* are allow us to reason about them inductively and define function on them recursively. Furthermore, the way we will stipulate their rules are the inference rules we use in the everyday mathematical activities.

2.2 Binary provability relation

The inductive definition of the Prop sort (propositions). (Sorts are just basic types of types.)

$$\text{Prop} ::= A \mid \bot \mid \text{Prop} \wedge \text{Prop} \mid \text{Prop} \vee \text{Prop} \mid \text{Prop} \to \text{Prop}$$

The inductive definition of the Cont type. (Contexts or premise sets.)

$$\text{Cont} ::= \text{nil} \mid \text{Prop} :: \text{Cont}$$

Here $::$ is different to $::=$. $x :: l$ means "extending the list l with the element x (from the left)". While nil is the "empty list".

Now, we define the binary relation $\Gamma \vdash A$.

$$\frac{}{\Gamma_1 :: \overset{nth}{A} :: \Gamma_2 \vdash A}\,(p_n) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\,(\bot^I)$$

$$\frac{A :: \Gamma \vdash B}{\Gamma \vdash A \to B}\,(\to^I) \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\,(\to^E)$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,(\wedge^I) \qquad \frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_i}\,(\wedge^E)$$

$$\frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 \vee A_2}\,(\vee^I) \qquad \frac{\Gamma \vdash A \vee B \quad A :: \Gamma \vdash C \quad B :: \Gamma \vdash C}{\Gamma \vdash C}\,(\vee^E)$$

We do not apply the "substructural" rules like the invariance of contexts over permutations. Contexts are just (ordered) lists. p_n above stands for the n^{th} term, from the left in the list, starting the counting form 0. So, if $\Gamma = A :: B :: B :: \text{nil}$, then the 0^{th} premise is A, the 1^{st} and 2^{nd} premises are B's, and there are no further premises, nil is not a term of the list, it is just the end mark of the list. I stands for introduction rule, E stands for elimination rule (they are also "induction rules" in the general case of inductive data

types). $\bot^I$ is not an introduction rule, it is the intuitionistic inference rule for absurdity. Without $(\bot^I)$ it is the *minimal logic,* that is

$$\text{minimal logic} = (p_n)(\to^I)(\to^E)(\wedge^I)(\wedge^E)(\vee^I)(\vee^E)$$

With $(\bot^I)$ it is the *intuitionistic logic*

$$\text{intuitionistic logic} = (p_n)(\bot^I)(\to^I)(\to^E)(\wedge^I)(\wedge^E)(\vee^I)(\vee^E)$$

With both $(\bot^I)$ and $(\bot^C)$ it is the *classical logic*

$$\text{classical logic} = (p_n)(\bot^I)(\bot^C)(\to^I)(\to^E)(\wedge^I)(\wedge^E)(\vee^I)(\vee^E)$$

where

$$\boxed{\frac{(A \to \bot) \to \bot}{A}(\bot^C)}$$

When we define negation as

$$\neg A := A \to \bot$$

the classical rule becomes

$$\frac{\neg\neg A}{A}$$

2.3 Examples

Example 2.1. $(A \wedge B) \to C \vdash A \to B \to C$

($\to$ is right-associative, while $\wedge$ is left-associative (however the latter is uninteresting).)

$$\dfrac{\dfrac{\dfrac{}{(A \wedge B) \to C}\, p_2 \quad \dfrac{\dfrac{}{A}\, p_1 \quad \dfrac{}{B}\, p_0}{A \wedge B}\wedge^I}{C}\to^E}{\begin{array}{l}B::A::(A\wedge B)\to C::\bullet\\ A::(A\wedge B)\to C::\bullet\end{array}\quad \dfrac{\dfrac{C}{B \to C}\to^I}{A \to B \to C}\to^I}$$

Example 2.2. $A \to B \to C \vdash (A \wedge B) \to C$

$$\dfrac{\dfrac{\dfrac{}{A \to B \to C}\, p_1 \quad \dfrac{\dfrac{}{A \wedge B}\, p_0}{A}\wedge_1^E}{B \to C}\to^E \quad \dfrac{\dfrac{}{A \wedge B}\, p_0}{B}\wedge_2^E}{\begin{array}{l}A\wedge B::A\to B\to C::\bullet\end{array}\quad \dfrac{C}{(A \wedge B) \to C}\to^I}$$

Example 2.3. $(A \wedge C) \vee (B \wedge C) \vdash (A \vee B) \wedge C$

Recall that

$$\neg A := A \to \bot$$

Example 2.4. $\vdash \neg\neg(A \vee \neg A)$

Example 2.5. $A \vee \neg A,\ A \to B \vdash \neg A \vee B$

Example 2.6. $\neg A \vee B \vdash A \to B$

2.4 More on implicational, minimal, and intuitionistic logic

There are trivial cases, when the proof by contraposition is provable in any logic:

Example 2.7. $\vdash A \to B \to (B \to \bot) \to (A \to \bot)$

Let us rewrite the inference rule associated with $\bot$, which is the so called **ex falso quodlibet** (from contradiction everything follows (latin))

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}$$

Note that, in the proof the ex falso quodlibet is not required.

Example 2.8. $\vdash (A \to B) \to (B \to C) \to (A \to C)$

However, the reverse case does not follows even in the intuitionistical logic. It can be checked by the "naive decision algorithm" we know: 1) in order to prove $A*B$ we have to use the introduction rule of * (or use a proof by cases method finding a disjunction in the context and then appeal to the introduction rule), 2) in case of atomic sentences, search for a conjunction or implication with the atomic sentence as an end type.

Example 2.9. $\nvdash ((B \to C) \to (A \to C)) \to A \to B$

To fix the problem we switch to intuitionisical logic and we will see two solutions.

Example 2.10. $\vdash (B \vee \neg B) \to ((B \to \bot) \to (A \to \bot)) \to A \to B$

or

Example 2.11. $\vdash ((B \to \bot) \to (A \to \bot)) \to ((A \to B) \to \bot) \to \bot$

3 Proof Theory with Proof Codes

3.1 Ternary provability relation

Now, we turned toward the proof trees as codes of proofs. To do this, it is necessary to define recursively the ternary relation $\Gamma \vdash P : A$. The Abstract Syntax Trees of *proof codes* have nodes and leaves and they are labeled by the symbols corresponding to the rules of inference. Proof codes are not real proofs, they are a kind of blind symbols. They will have meanings (become proper codes) after they participate in a real proof.

Definition 3.1. The language of *proof codes* and defined by recursion:

$$\text{ProofCode} ::= p_n \mid \bot^I A\ P \mid \to^I A\ P \mid \to^E P_1 P_2 \mid \wedge^I P_1\ P_2 \mid \wedge_i^E P \mid \vee_i^I A\ P \mid \vee^E P_1 P_2 P_3$$

where P, P_1, P_2, P_3 are ProofCode's, A is a Prop (sometimes called Type or Form), n is a nat and $i = 1, 2$.

Definition 3.2. The ternary provability relation $\Gamma \vdash P : A$ is defined by the recursive structure of proof codes

$$\frac{}{A :: \Gamma \vdash p_0 : A} \qquad \frac{\Gamma \vdash P : \bot}{\Gamma \vdash \bot^I A\ P : A}$$

$$\frac{\Gamma \vdash p_n : A}{B :: \Gamma \vdash p_{n+1} : A}$$

$$\frac{A :: \Gamma \vdash P : B}{\Gamma \vdash \to^I A\ P : A \to B} \qquad \frac{\Gamma \vdash P_1 : A \to B \qquad \Gamma \vdash P_2 : A}{\Gamma \vdash \to^E P_1\ P_2 : B}$$

$$\frac{\Gamma \vdash P_1 : A \quad \Gamma \vdash P_2 : B}{\Gamma \vdash \wedge^I P_1\ P_2 : A \wedge B} \qquad \frac{\Gamma \vdash P : A_1 \wedge A_2}{\Gamma \vdash \wedge_i^E P : A_i}$$

$$\frac{\Gamma \vdash P : A_i}{\Gamma \vdash \vee_i^I A_i P : A_1 \vee A_2} \qquad \frac{\Gamma \vdash P_1 : A \vee B \quad A :: \Gamma \vdash P_2 : C \quad P_3 : B :: \Gamma \vdash C}{\Gamma \vdash \vee^E P_1\ P_2\ P_3 : C}$$

Remark 3.1. Now, we see the role of A in $\bot^I A\ P \to^I A\ P$, and $\vee_i^I A_i P$. When the type of the conclusion consists a new type not, which is not a type of any proof code in the premises, then this new type must be marked in the conclusion's code.

Remark 3.2. A proof tree is an AST's of a proof code P, which is in relation with a context Γ and a type or a proposition A by the provability relation $\Gamma \vdash P : A$.

3.2 Proof reduction and normalization

Without detours in proofs do not just ease the reasoning, but they give immediate connections between introductions and elimination rules. The simplest such detour is when the introduction rule of conjunction follows a conjunction elimination rule immediately.

$$\dfrac{\dfrac{P_1 \,:\, A_1 \quad P_2 \,:\, A_2}{\wedge^I P_1\ P_2 \,:\, A_1 \wedge A_2}\wedge^I}{\wedge_i^E \wedge^I P_1\ P_2 : A_i}\wedge_1^E$$

Intuitively, we introduce a now meta-notation:

$$\Gamma \vdash P \equiv Q : A$$

means that "the proof code P is (external or definitional) equivalent to Q and they are typed by A in the context Γ". Recursively, $\equiv$ is defined by the *computational rules*

$$\dfrac{\Gamma \vdash P_1 \,:\, A_1 \qquad \Gamma \vdash P_2 \,:\, A_2}{\Gamma \vdash \wedge_i^E \wedge^I P_1\ P_2 \equiv P_i : A_i}$$

(or for short $\wedge_i^E \wedge^I P_1\ P_2 \equiv P_i$ or $\wedge_i^E \wedge^I P_1\ P_2 \to_\beta P_i$ where $\to_\beta$ is the so-called "one step beta reduction") and the *structural law*

$$\dfrac{\Gamma \vdash P \equiv Q : A \qquad \Gamma \vdash P' \equiv Q' : B}{\Gamma \vdash \wedge^I P\ P' \ \equiv \wedge^I Q\ Q' :\ A \wedge B}$$

which means that conjunction respects definitional equivalence.

Remark 3.3. Note that, when we think of $\Gamma \vdash P : A$ as the claim $f_i : C \to A_i$ (with f_i as P_i, C as Γ, X_i as A_i, $\wedge_i^E$ as pr_i and $A_1 \wedge A_2$ as $X_i \times X_2$), then $pr_i \circ (f_1 \prod f_2) = f_i$ means the same as the computational rule 1 of product object.

Remark 3.4. The reverse case also makes sense. One can assume that all the proofs of the type $A \wedge B$ has the same pattern:

$$\dfrac{\dfrac{P \,:\, A_1 \wedge A_2}{\wedge_1^E P \,:\, A_1}\wedge_1^E \qquad \dfrac{P \,:\, A_1 \wedge A_2}{\wedge_2^E P \,:\, A_2}\wedge_2^E}{\wedge^I \wedge_1^E P \wedge_2^E P : A_1 \wedge A_2}\wedge^I$$

Accordingly, the so called η-rule is the following:

$$\dfrac{\Gamma \vdash P : A_1 \wedge A_2}{\Gamma \vdash \wedge^I \wedge_1^E P \wedge_2^E P \equiv P : A_1 \wedge A_2}$$

it is the analogue of the computational rule 2 of the product object. In proof theory, we don't use this, however the software implementations are extensively apply it.

Turning to the implicational case, the picture becomes more complex. Before that, we will discuss the Subject Reduction lemma, which is the main intuition of the substitution of proofs. Subject Reduction is a structural reduction mechanisms to merge two subsequent lemmas into one theorem.

Theorem 3.1 (Subject Reduction)**.** For every sentences A, B, and context Γ,

$$\text{if } A :: \Gamma \vdash P : B \text{ and } \Gamma \vdash Q : A, \text{ then } \Gamma \vdash P[Q] : B.$$

(Here, $P[Q]$ is the de Bruijn substitution, and defined later.)

First, let us consider an example in order to see the well-known method "using a lemma" in a proof.

Example 3.1. Solve the problem $\neg A \wedge \neg B :: nil \vdash ? : \neg(A \vee B)$ successively, first proving that

$$\neg B :: \neg A \wedge \neg B :: nil \vdash \ ? \ : \neg(A \vee B)$$

holds, since

$$\neg A :: \neg B :: \neg A \wedge \neg B :: nil \vdash \ ? \ : \neg(A \vee B)$$

holds and

$$\neg B :: \neg A \wedge \neg B :: nil \vdash \ ? \ : \neg A$$

holds too. Then

$$\neg A \wedge \neg B :: nil \vdash ? : \neg(A \vee B)$$

holds, since

$$\neg A \wedge \neg B :: nil \vdash \ ? \ : \neg B$$

holds.

Proof. Let $\Gamma := \neg B :: \neg A \wedge \neg B :: nil$. Let the type of the lemma is $\neg A$, and the type of the conclusion $(A \vee B) \to \bot$.

1. It is clear that

$$\neg B :: \neg A \wedge \neg B :: nil \vdash \ \wedge_1^E p_1 \ : \neg A$$

hence, $Q := \wedge_1^E p_1$.

2.

$$\dfrac{\dfrac{}{A \vee B}\, p_0 \qquad \overset{A::\ldots}{\dfrac{\dfrac{}{A \to \bot}\, p_2 \quad \dfrac{}{A}\, p_0}{\bot}\, {\to}E} \qquad \overset{B::\ldots}{\dfrac{\dfrac{}{B \to \bot}\, p_3 \quad \dfrac{}{B}\, p_0}{\bot}\, {\to}E}}{\bot}\, {\vee}E$$

$$\overset{A\vee B::\neg A::\neg B::\neg A\wedge\neg B::\bullet}{\dfrac{\bot}{P : (A \vee B) \to \bot}}\, {\to}I$$

First, we can stick the proof of $\neg A$ to the leaf p_0, which is colored by green. However, both P and Q must be modified in the new proof. Second, p_1 in Q points to an open premise $\neg A \wedge \neg B$, hence we have to count the context extender inference rules, colored by red, and lift the index of p_1 by this number: $\wedge_1^E p_3$. Third, since $\neg A$ is canceled in the new context (Γ), the indices of the leaves pointing to open premises must be lower by 1: $p_3 \to p_2$. Hence,

$$\dfrac{\dfrac{}{A \vee B}\,p_0 \qquad \dfrac{\dfrac{\dfrac{}{\neg A \wedge \neg B}\,p_3}{A \to \bot}\wedge_1^E \quad \dfrac{}{A}\,p_0}{\bot}{\scriptstyle A::\ldots}\to^E \qquad \dfrac{\dfrac{}{B \to \bot}\,p_2 \quad \dfrac{}{B}\,p_0}{\bot}{\scriptstyle B::\ldots}\to^E}{\bot}\vee^E$$

$$\dfrac{\bot}{P : (A \vee B) \to \bot}{\scriptstyle A\vee B::\neg B::\neg A\wedge\neg B::\bullet}\to^I$$

This is $P[Q]$. □

Definition 3.3 (de Bruijn Substitution)**.** Let P and Q be proof codes such that $A :: \Gamma \vdash P : B$ and $\Gamma \vdash Q : A$ in a context Γ and for sentences A and B. Then $P[Q]$ ("the proof obtained by substituted Q into P-s first open premise") constructed as follows

1. find all the leaves p_n in P which point to A;
2. decrease all the p_m's in P by 1, which point to the element of Γ (the open premises except A), the result will be P';
3. put one instance of Q to the p_n's above in P', but before increase the open p_l's in these Q's by the number of context extender inference rules, which are under the leaves p_n, respectively.

(Cf. de Bruijn (1994).)

p_n's will be the places of change, initially they all point to A. p_m's are the non-A open leaves. Since, the new context lacks A they must be decreased by 1. Finally, the instances of Q which will be placed into the places of p_m's may has open leaves p_l's. They must be increased by the times of application of context extending inference rules, such as $\to^I$ and $\vee^E$.

We skip the proof of the subject reduction (it goes by structural induction on the height of P).

3.3 Implicational case

The reduction concerning "if, ... then ..." sentences (implication or conditional) is based on the following situation:

$$\dfrac{A::\ldots\dfrac{\overline{B}}{A\to B}\to^I \quad \overline{A}}{B}\to^E$$

In proof code formalism the β *redex* is

$$\dfrac{\dfrac{A :: \Gamma \vdash P : B}{\Gamma \vdash \to^I A\, P : A \to B}\to^I \quad \dfrac{}{\Gamma \vdash Q : A}}{\Gamma \vdash \to^E \to^I A\, P\, Q : B}\to^E$$

and the β *reduct* is

$$\to^E \to^I A\, P\, Q \quad \to_\beta \quad P[Q] : B$$

or in conversion style:

$$\dfrac{A :: \Gamma \vdash P : B \qquad \Gamma \vdash Q : A}{\Gamma \vdash \to^E \to^I A\, P\, Q \equiv P[Q] : B}\beta$$

which is precisely the subject reduction situation.

Note that, the η reduction is the following:

$$\to I A \to^E P\, p_0 \to_\eta P.$$

In order to find a place for the implication in the categorical semantics, take a look at the exponential object.

Definition 3.4. In a Cartesian Category $\mathcal{C}$ the object Y^X equipped with the morphism $eval : Y^X \times X \to Y$ is an *exponential object,* if for all Z and morphism $f : Z \times X \to X$ there is a unique morphism $\lambda f : Z \to Y^X$ such that

$$\begin{array}{ccc} Z & & Z \times X \\ \Big\downarrow{\scriptstyle \lambda f} & & \Big\downarrow{\scriptstyle ((\lambda f)\circ pr_1)\prod(id_X \circ pr_2)} \\ Y^X & & Y^X \times X \xrightarrow[eval]{} Y \end{array} \qquad f : Z \times X \to Y$$

commutes.

Hence, the computation rules are the following:

1. $eval \circ ((\lambda f) \circ pr_1) \prod (id_X \circ pr_2)) = f$, for all $f : Z \times X \to Y$
2. $\lambda(eval \circ ((g \circ pr_1) \prod (id_X \circ pr_2))) = g$, for all $g : Z \to Y^X$

Remark 3.5. Product of two morphisms had defined when both have a common domain. However, often two morphisms to connect with product, $f : X \to Z \times W$ and $g : Y \to Z \times W$ are transformed into two morphisms with common domain

$$f \circ pr_1, \quad g \circ pr_2 : X \times Y \to Z \times W$$

and then they become two terms of one product morphism: $(f \circ pr_1) \prod (g \circ pr_2)$. In order to simplify the notation sometimes a short denotation is used:

$$f \prod g \coloneqq (f \circ pr_1) \prod (g \circ pr_2)$$

Here we use the symbol $\prod$ (torii) to denote this operation. Hence, the previous equations can be rewrite as

1. $eval \circ ((\lambda f) \prod id_X) = f$, for all $f : Z \times X \to Y$
2. $\lambda(eval \circ (g \prod id_X)) = g$, for all $g : Z \to Y^X$

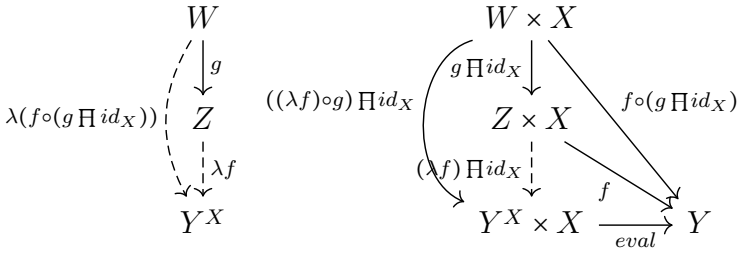
Of course, the exponential morphism λ has also a distributive law with respect to composition.

Lemma 3.1. For $f : Z \times X \to Y$ and $g : W \to Z$:

$$(\lambda f) \circ g = \lambda(f \circ (g \prod id_X))$$

Proof. Indeed, for $f \circ (g \prod id_X) : W \times X \to Y$, $h = (\lambda f) \circ g$ has the unique property $eval \circ (h \prod id_X) = f \circ (g \prod id_X)$, since

$$eval \circ (((\lambda f) \circ g) \prod id_X) = eval \circ ((\lambda f) \prod id_X) \circ (g \prod id_X) = f \circ (g \prod id_X)$$



□

Remark 3.6. In *locally small* categories (where all the $\mathrm{Hom}(A; B)$ classes are sets), the Currying/un-Currying phenomenon occurs. Two sets are isomorphic in the *set-theoretical sense,* i.e. there is a bijection between them $\mathrm{Hom}(Z \times X, Y) \cong \mathrm{Hom}(Z, Y^X)$. It is clear that this relationship is a higher order one, since the isomorphism $\cong$ is between sets (or classes) and not between object, as morphisms. When the category is not locally small, this relation is just a function-like generalized relation between (Hom-)classes. What is more, in Z and Y this correspondence is natural. However, we do not need natural transformations, so we skip them. This close connection of $\_ \times X$ and $\_^X$ makes the category closed.

Definition 3.5. A Cartesian Closed Category (CCC) is a category equipped with terminal object, product and exponential objects. A Bi-Cartesian Closed Category (BCCC) is a CCC with co-product and initial element.

Theorem 3.2 (Currying)**.** Let $\mathcal{C}$ be a *locally small* CCC. Then, for every object X, Y, Z, there is bijective function showing that

$$\mathrm{Hom}(Z \times X, Y) \cong \mathrm{Hom}(Z, Y^X).$$

Proof. Indeed, the map

$$\lambda : \mathrm{Hom}(Z \times X, Y) \to \mathrm{Hom}(Z, Y^X),\ f \mapsto \lambda f$$

is a bijection. Firstly, suppose that $\lambda f_1 = \lambda f_2$. Then, by the commutative property (Lambek's equation 1)

$$f_1 = eval \circ ((\lambda f_1) \prod id_X)) = eval \circ ((\lambda f_2) \prod id_X)) = f_2$$

Secondly, suppose $g : Z \to Y^X$, then by Lambek's equation 2 for exponential objects, the map $eval \circ (g \prod id_X) : Z \times X \to Y$ satisfies

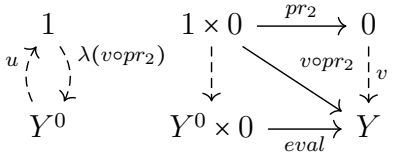
$$\lambda(eval \circ (g \prod id_X)) = g.$$

□

Note, that, if $\mathcal{C}$ is not locally small, then λ can be modeled as a class of pairs (for example in Gödel-Bernays-von Neumann Set Theory), which is surjective and injective.

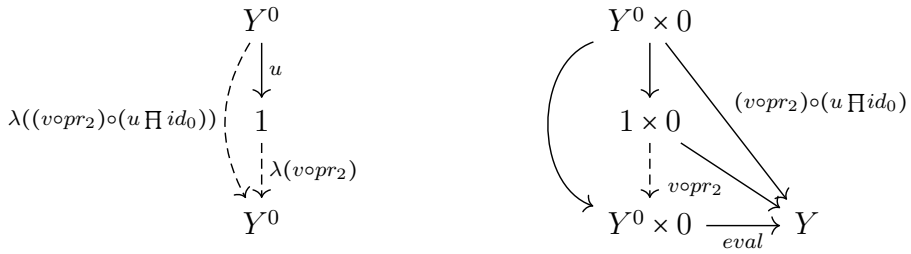
Example 3.2. Prove that $Y^0 \cong 1$, $Y^1 \cong Y$, $Y^{X_1+X_2} \cong Y^{X_1} \times Y^{X_2}$,...

Proof. See the diagram!



$$u \circ \lambda(v \circ pr_2) = id_1$$

by the universal property of the terminal object. Secondly,



$$\begin{aligned}
\lambda(v \circ pr_2) \circ u = \lambda((v \circ pr_2) \circ (u \prod id_0)) = & \qquad [\text{distributive law}] \\
= \lambda \,\text{eval} = & \qquad [\text{By Currying: } \{\bullet\} = \text{Hom}(0, Y^{Y^0}) \cong \\
& \qquad \cong \text{Hom}(Y^0 \times 0, Y) = \{\text{eval}\}] \\
= \lambda(\text{eval} \circ id_{Y^0 \times 0}) = & \qquad [\{\bullet\} = \text{Hom}(0, (Y^0 \times 0)^{Y^0}) \cong \text{Hom}(Y^0 \times 0, Y^0 \times 0)] \\
= \lambda(\text{eval} \circ (id_{Y^0} \prod id_0)) = & \\
= id_{Y^0} & \qquad [\text{Lambek 2}]
\end{aligned}$$

Where we used the fact that by initiality, $\text{Hom}(0, \_)$ is a singleton. □

3.4 Structural theorems in implicational logic

Recall that implicational logic is defined as follows:

$$\text{Prop} ::= A \mid \text{Prop} \to \text{Prop}$$

where A's are the atomic formulas or propositions or sentences, or *types*, or formula-, proposition-, sentence-, or *type*variables.

$$\text{Context} ::= nil \mid \text{Prop} :: \text{Context}$$

here nil is the empty context and $::$ is the context extending operator.

$$\text{ProofCode} ::= p_n \mid \to^I \text{Prop ProofCode} \mid \to^E \text{ProofCode ProofCode}$$

are the proofs and the inference rules are:

$$\frac{}{A :: \Gamma \vdash p_0 : A} \qquad \frac{\Gamma \vdash p_n : A}{B :: \Gamma \vdash p_{n+1} : A}$$

$$\frac{A :: \Gamma \vdash P : B}{\Gamma \vdash \to^I A\, P : A \to B} \qquad \frac{\Gamma \vdash P : A \to B \quad \Gamma \vdash Q : B}{\Gamma \vdash \to^E P\, Q : B}$$

And the computational rule is:

$$\underbrace{\to^E \to^I A\, P\, Q}_{\beta \text{ redex}} \to_\beta \underbrace{P[Q]}_{\beta \text{ reduct}}$$

Theorem 3.3 (Weak Normalization)**.** Consider context Γ and proposition A in the implicational logic. If $\Gamma \vdash P : A$, then there is a proof code Q without any β-redex in it, such that $\Gamma \vdash Q : A$. (Cf. Prawitz (1965).)

Definition 3.6. If a proof has no β redex in it, then it is a *normal* proof.

Remark 3.7. Such a Q is a β reduct of P, i.e.

$$P \twoheadrightarrow_\beta Q.$$

Proof. Let R be a β redex in P. It is clear that in the proof tree of P there is a situation of the form

$$\frac{\Gamma' \vdash \to^I A'\, S : A' \to B' \quad \Gamma' \vdash T : A'}{\Gamma' \vdash R : B'}.$$

Let $\deg R$ (the reduction degree of R in $\Gamma \vdash P : A$) be the number of $\to$ symbols in $A' \to B'$. And let $\deg P$ ("the reduction degree of P in $\Gamma \vdash P : A$ ") be 0, if P has no redex in it, else $\max_{R \in P} \deg R$.
We will perform call-by-value (cbv) reduction among the maximal redexes of P, i.e. we reduce the right-most redexes with maximal degree, so that, in the proof code $R = \to^E \to^I A'\, S\, T$ there is no proper subterm, with degree $\deg P$.
The proof goes by double induction on the index (d, m) where d is $\deg P$, m is the number of redexes of the degree d (the maximal reduction degree).

$$\dfrac{\dfrac{\overline{S : B'}}{\to^I A'\, S : \boxed{A' \to B'}}\to^I \quad \overline{T : A'}}{\to^E \to^I A'\, S\, T : B'}\to^E \qquad A' ::: \ldots$$

The boxed one is the so-called *peek* or *maximal* formula, because it has the maximal number of $\to$ in its types, comparing the neighboring nodes' types. The degree of the redex is the number of $\to$ symbols in the corresponding maximal peek.

Lemma 3.2 (Right-most engine)**.** After the reduction of the "right-most" redex R, the reduct $R' = S[T]$ has strictly less reduction degree then that of R:

$$R \to_\beta R' \;\Rightarrow\; \deg R < \deg R'$$

This holds because of the fact that after the reduction, R is replaced by its reduct $S[T] : B'$, which has $\to$ symbols fewer by one in its type B' than in $A' \to B'$, the type of the peek, and because "right-most" means that there is maximal degree redexes neither in S nor in T. Hence, the number m of the maximal redexes or even the degree d of the proof strictly lowers. Note also that, after performing a reduction, finite redexes may be created, but all of its degree are less than d. □

Remark 3.8. We found that the cbv reduction strategy terminates in finite steps. The implicational logic has also the *strong normalization property.* That is, whatever reduction strategy one uses to reduce proof terms, it leads to a normal form in finite steps. In other words, there is no infinite sequence $(P_i)_{i \in \omega}$ containing pairwise distinct proof terms, such that

$$P = P_0 \to_\beta P_1 \to_\beta P_2 \to_\beta \ldots$$

(Here $P' \to_\beta P''$ means that one redex is reduced in P' and resulted in P''.) The proof of the Strong Normalization Theorem goes by using Kőnig's Infinity Lemma.

The height of the AST of P is denoted by $\mathrm{hght}\, P$. The following theorem is a sharpening of the Normalization Theorem by adding an upper bound for the height of the normal form.

Theorem 3.4 (Upper bound for height of normal proofs)**.** If $\deg P \leq d$ $(d \geq 1)$ in $\Gamma \vdash P : A$, $\mathrm{hght}\, P \leq h$, then there is a normal proof N such that $P \to_\beta P_1 \to_\beta P_2 \to_\beta \cdots \to_\beta N$, and

$$\mathrm{hght}\, N \leq \underbrace{2^{2^{\cdot^{\cdot^{2^h}}}}}_{d}$$

(Cf. Fortune et al. (1983).)

Proof. It is enough to prove that for P with $\deg P \leq d$ $(d \geq 1)$, there is a proof M such that $P \to_\beta P_1 \to_\beta P_2 \to_\beta \cdots \to_\beta P_n \to_\beta M$, $\deg M \leq d-1$ in $\Gamma \vdash P_n : A$, and $\mathrm{hght}\, M \leq 2^h$.

By structural induction on the structure of P. The $P = p_n$ case is trivial. Let $P =\!\to^I A'\, Q$ then Q reduce to an M' with $\deg M' \leq d-1$ and $\mathrm{hght}\, M' \leq 2^{h-1}$. Hence, P reduces to $M =\!\to^I A'\, M'$ and $\deg M \leq d-1$ and $\mathrm{hght}\, M \leq 2^{h-1}+1 \leq 2^h$.

Let $P =\!\to^E Q\, R$ then Q, R reduce to Q', R' with $\deg Q', \deg R' \leq d-1$ and $\mathrm{hght}\, Q', \mathrm{hght}\, R' \leq 2^{h-1}$. Then even if P is a redex with degree d, the reduct will be such that $\mathrm{hght}\, M \leq 2^{h-1}+2^{h-1} \leq 2^h$, by the "Right-most engine" lemma.
Then, by induction on d shows the main claim. □

Now, we define the important auxiliary notion of *branch*. The main feature of a branch is that, it is a sequence of consecutive formulas in a proof, such that for any consecutive pair A, B, either A is a subformula of B or B is a subformula of A. This is the so-called *subformula property*. Note that a major premise of an elimination rule of a logical connective is the premise in where its own logical connective is occurred (i.e. the first one). The others are the minor ones.

Definition 3.7. A *branch* of a proof tree P is a sequence of consecutive nodes (every one of them, except the first, is the parent of the previous) in the proof tree with elements $A_1, \ldots, A_k$, where A_1 is a leaf and A_k is a minor premise of an elimination rule or the root, and the other A_i's are not.

$$\dfrac{\overbrace{\underbrace{B_n \to A_{i+1}}_{A_i}}^{\text{major premise of } \to^E} \qquad \overbrace{B_n}^{\text{minor premise of } \to^E}}{A_{i+1}}\ \to^E$$

Example 3.3. The following proof has 3 branches. The red one starts with 2 → elimination rules, it goes though 2 *major premise* of an → elimination rule, and ends with 2 introduction rules, it stops at the root (*main branch*). The green one consists of one instance of the introduction rule, certainly it stops at the *minor premise* of an → elimination rule. The blue one is a one-element branch stops at *minor premise* of an → elimination rule.

$$\dfrac{\dfrac{\dfrac{A \to (D \to E) \to C \quad A}{(D \to E) \to C}\to^E \quad \dfrac{E}{D \to E}\to^I}{C}\to^E}{\dfrac{B \to C}{A \to B \to C}\to^I}\to^I$$

Lemma 3.3. Every node in a proof is in exactly one branch of the proof.

Proof. Induction on the structure of the proof. Leaf case: trivial. Node case. Introduction rule case

$$\dfrac{P : B}{\to^I \ A\,P : A \to B}$$

by the induction hypothesis, P is in only one branch and there is only one way to extend the branch with the root. Elimination rule case

$$\dfrac{P : A \to B \quad Q : B}{\to^E \ P\,Q : B}$$

by by induction hypothesis, both P and Q have their own branch to belong, and there is only one way to extend with the root, it is the branch ending in P. □

The *main branch* is the branch of the root. A proof is *normal,* if it does not contain redex.

Theorem 3.5 (Branch theorem)**.** Every branch of a normal proof is of the form:

$$(E_1, \ldots, E_n, M, I_1, \ldots, I_k)$$

(possibly no E's and I's) such that E) E_i's are main premises of a → elimination rule and the immediate subsequent of every formula is contained in the formula itself. M) M (*the minimum formula* or *hollow*) is a premise of an introduction rule or the end formula of the branch. I) All I_i's are premises of a → introduction rule and the immediate subsequent of every formula contains the formula itself.

Proof. Let M be the highest premise of an introduction rule in the proof, if such exists. Then, below that, there are no applications of an elimination rule because the proof is normal. Above M, the subformula property holds, since it is a branch. Also below that. If there is no application of introduction rule, then let M be the end formula. The subformula property also holds. □

Theorem 3.6 (Subformula Principle)**.** Let P be a normal proof such that $\Gamma \vdash P : A$. Then every formula occurring in P is a subformula of an element of $\Gamma \cup \{A\}$.

Proof. We associate rank to every branch in a normal proof. The main branch has rank 0. The branch ends in an n-rank branch has rank $n + 1$.

By induction on the rank we prove that, in every branch of rank n the property holds.

If $n = 0$, then it is the main branch and by the subformula property, the claim holds

Induction case. An $n + 1$-rank branch ends in a C like that:

$$\frac{C \to D \quad C}{D}$$

where $C \to D$ belongs to an n-rnak branch. Hence, by the induction hypothesis, all the formulas in that branch contained in an element of $\Gamma \cup \{C\}$, but C is contained in $C \to D$, which is the n-rank branch, hence, C is a subformula of $\Gamma \cup \{A\}$. □

Definition 3.8. An inference system is *(syntactically) consistent,* or *free form contradiction,* if there is a formula A such that $\nvdash A$.

In structural proof theory consistency proofs are based on the structural properties of proofs. An other type of consistency proofs, when a semantics is provided and one pointing out that every provable formulas are true, but there are false formulas. We – by understandable reasons – skip the semantic way.

Theorem 3.7. The implicational logic is consistent. In fact, $\nvdash A$, if A is an atomic formula.

Proof. Suppose the contrary: $\vdash P : A$. Then there is a normal proof N such that $\vdash N : A$. By the Subformula Principle, the only formula used as a type concerning N is A. If $N = p_n$, then there is no premise pointing p_n to, so it is impossible. If the last applied rule is introduction, then A is compound, which is also a contradiction. If the last rule applied is an elimination rule, then a node typed by $C \to A$ must occur above A, which is a contradiction, since $C \to A$ is not a subformula of A. □

4 Lambda calculus

4.1 Typed or Type-free Lambda Calculus?

The pure and type free Lambda Calculus (LC) can be defined formally in two separate ways. The first one is the so-called *lambda calculus with variables* (syntax here: S'), the second one is the variable free or de Bruijn style (syntax here: S). The variable based

style has the nuisance property that there are syntactically different however semantically identical terms (hence, the notion of α-equivalence is needed). The de Bruijn style is just another syntax which can be defined easier using the notion of nat and F-algebras.

Definition 4.1. The syntax of the Lambda Calculus in the variable based style is the following:

$$S' ::= x \mid PQ \mid \lambda x.P$$

$x \in \text{var} = \{x_1, x_2, \ldots, ...\}$. The variable free version is

$$S ::= n \mid PQ \mid \lambda P$$

$n : \text{nat}$. (All the compound terms are considered to be placed inside brackets.)
The elements of S and S' are called terms.

Remark 4.1. Clearly, S' is the initial F-algebra of the functor:

$$X \mapsto \text{Var} + X^2 + \text{Var} \times X$$

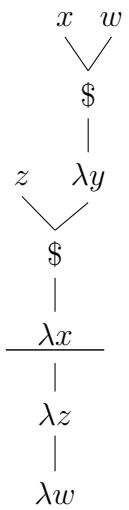
while in the variable free case, S is the initial algebra of

$$X \mapsto \text{nat} + X^2 + X$$

Here $\lambda x.t$ and λt are called *abstractions*, ts is called application. In AST's, application is denoted by \$

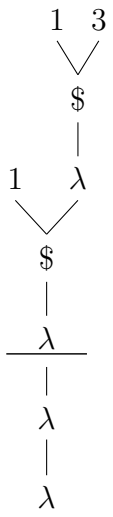
$$t \ \$ \ s \text{ or } \ \$ts.$$

Example 4.1. To make the "free variables" (not bounded by lambda's) explicit, we use de Bruijn's idea, as he closes the tree by lambda's. The AST of $\lambda x.(z\lambda y.(xw))$ is



which is the same as $\lambda(1\lambda(13))$, after fixing the order of "free variables"; if w is the most exterior and z is the one after w. (This

means that z is the first among the free variable of the term (*indexed by* 0) and w is the second among the free variable (*indexed by* 1) in the term.)



Before going inside into the type free calculus, let us point out some fact about the *typed,* that we already know.

Definition 4.2. Typeability relation is defined by the following rules recursively based on the structure of the terms. (Sometimes an additional type A is added to the term $\lambda x.P$ like $\lambda x : A.P$ or $\lambda\, A\, P$ in the syntax.) Variable based version:

$$\frac{}{\{x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n\} \vdash x_i : A_i} \qquad \frac{\Gamma \vdash P : A \to B \quad \Gamma \vdash Q : A}{\Gamma \vdash PQ : B} \qquad \frac{\Gamma \cup \{x : A\} \vdash P : B}{\Gamma \vdash \lambda x : A.P : A \to B}$$

de Bruijn version:

$$\frac{}{\Gamma_1 :: \underset{n^{\text{th}}}{A} :: \Gamma_2 \vdash n : A} \qquad \frac{\Gamma \vdash P : A \to B \quad \Gamma \vdash Q : A}{\Gamma \vdash PQ : B} \qquad \frac{A :: \Gamma \vdash P : B}{\Gamma \vdash \lambda A P : A \to B}$$

The above is the (Pure) Simple Type Lambda Calculus or (Pure) Simply Typed Lambda Calculus or (Pure) Simple Type Theory (STT for short). There is a close correspondence between STT the Natural Deduction Theory of the Implicational logic ($\mathbf{N}_{\to}$)

Theorem 4.1 (Curry–Howard Isomorphism)**.** The STT and $\mathbf{N}_{\to}$ are isomorphical, i.e. there is a translation $()^*$ from the terms of STT to the proof codes of $\mathbf{N}_{\to}$ such that

$$\Gamma \vdash_{\text{STT}} P : A \qquad \Longleftrightarrow \qquad \Gamma^* \vdash_{\mathbf{N}_{\to}} P^* : A.$$

Proof. Via de Bruijn notation:

$$n^* := p_n \qquad (\lambda\, A\, P)^* :=\to^I A\, P^* \qquad (PQ)^* :=\to^E P^*\, Q^*$$

The rest is trivial pattern matching. □

Remark 4.2. A corresponding **Curry–Howard paradigm** can be stated for any simple type system, i.e. type systems where the types are not parametrized by any type or terms. STT can be extended for example by the one-element type $\top$ or the pair type $\wedge$, as well as by nat or bool, and so on. However, the equality type $n = m$ or the "list of A's" type with parameter A become yet Dependent Type Theories.

Theorem 4.2 (Curry–Howard–Lambek Isomorphism)**.** There are translations $()^* : \mathrm{STT}_{\to,\wedge,\top} \to \mathbf{N}_{\to,\wedge,\top}$ and $()^{**} : \mathbf{N}_{\to,\wedge,\top} \to \mathrm{CCC}_{(.)^{(.)},\Pi,1}$ such that

$$\Gamma \vdash P : A \qquad \Longleftrightarrow \qquad \Gamma^* \vdash P^* : A \qquad \Longleftrightarrow \qquad P^{**} \in \mathrm{Hom}(\Gamma^{**}, A^{**})$$

and

$$\Gamma \vdash P \equiv_{\beta\eta} Q : A \qquad \Longleftrightarrow \qquad \Gamma^* \vdash P \equiv_{\beta\eta} Q^* : A \qquad \Longleftrightarrow \qquad P^{**} = Q^{**}.$$

Remark 4.3. Note, that this gives raise a connection between Heyting Algebras and intuitionistic logic, since BCCC is a Heyting algebra. Without the reduction rules the adequate order theoretical semantics of the intuitionistic logic Heyting algebra.

The core notion here is the substitution. It makes also the notion of free variable meaningful.

Definition 4.3 (Substitution and free variable)**.** Let M and N be to terms.

1. In S', $M[N/x]$ is defined by structural recursion:

   (a) $y[N/x] \coloneqq \begin{cases} N, & y = x \\ y, & x \neq y \end{cases}$

   (b) $PQ[N/x] \coloneqq P[N/x](Q[N/x])$

   (c) $\lambda y.P[N/x] \coloneqq \begin{cases} \lambda y.P, & y = x \\ (\lambda z.(P[z/y][N/x]), & x \neq y \quad (z \notin PNx) \end{cases}$

   The set of free variables $FV(M)$ is defined as: $FV(x) = \{x\}$, $FV(PQ) = FV(P) \cup FV(Q)$, $FV(\lambda x.P) = FV(P) \setminus \{x\}$.

2. In S, let s be a sequence of terms $N_0, \dots$. $M[s]$ is defined by structural recursion:

   (a) $n[N_0, \dots] \coloneqq N_n$

   (b) $PQ[s] \coloneqq P[s]Q[s]$

   (c) $\lambda P[s] \coloneqq \lambda(P[(0, s\uparrow^1)])$

   where $\uparrow^1$ elevates the "free variables" in s by 1 and s is shifted by 1 to the left as $(0, s\uparrow^1)$. Here "free variable" means that k in P is greater than the number of λ's minus 1, below k.

Example 4.2.

$$\begin{aligned}\lambda x.(z\lambda y.(xw))[(\lambda x.x)/w] &\overset{\text{de Bruijn}}{=} \lambda(1\lambda(13))[0,\lambda 0,2,...] = \\ &= \lambda(1\lambda(13)[0,1,(\lambda 0)\uparrow^1,3,...]) = \\ &= \lambda(1\lambda(13[0,1,2,(\lambda 0)\uparrow^2,4,...])) = \\ &= \lambda(1\lambda(1\lambda 0)) \overset{\text{un-de Bruijn}}{=} \lambda x.(z\lambda y.(x\lambda x.x))\end{aligned}$$

$$\begin{aligned}\lambda x.(z\lambda y.(xw))[(\lambda x.z)/z] &\overset{\text{de Buijn}}{=} \lambda(1\lambda(13))[\lambda 1,1,...] = \\ &= \lambda(1\lambda(13)[0,(\lambda 1)\uparrow^1,2,...]) = \\ &= \lambda(((\lambda 1)\uparrow^1)\lambda(13[0,1,(\lambda 1)\uparrow^2,3,...])) = \\ &= \lambda(((\lambda 1)\uparrow^1)\lambda(13)) = \\ &= \lambda(((\lambda 2)\lambda(13)) \overset{\text{un-de Bruijn}}{=} \lambda x.(\lambda x.z)\lambda y.(xw)\end{aligned}$$

$$\begin{aligned}\lambda x.(z\lambda y.(xw))[(\lambda z.x)/z] &\overset{\text{de Buijn}}{=} \lambda(1\lambda(13))[\lambda 1,1,...] = \\ &= \lambda(1\lambda(13)[0,(\lambda 1)\uparrow^1,2,...]) = \\ &= \lambda(((\lambda 1)\uparrow^1)\lambda(13[0,1,(\lambda 1)\uparrow^2,3,...])) = \\ &= \lambda(((\lambda 1)\uparrow^1)\lambda(13)) = \\ &= \lambda(((\lambda 2)\lambda(13)) \overset{\text{un-de Bruijn}}{=} \lambda u.(\lambda z.x)\lambda y.(uw)\end{aligned}$$

Remark 4.4. Note, that in S' the α reduction is needed. If $y \notin FV(P)$, then

$$\lambda x.P \to_\alpha \lambda y.P[x/y]$$

is a one-step α reduction. In de Bruijn notation, being no variable names, the notion is neglected.

4.2 Beta reduction

Definition 4.4. *the elementary β reduction*

$$(\lambda x.P)Q \to_\beta P[Q/x]$$

$$(\lambda P)Q \to_\beta P[Q]$$

There can be defined the one-step beta reduction in two ways. They differ in the two items. The first one is that M is a reduction of M. The second one is that one can do the reduction in each terms of an application.

Definition 4.5. *One-step β reduction:*

$$\frac{}{(\lambda x.P)Q \to_\beta P[Q/x]} \quad \frac{P \to_\beta P'}{PQ \to_\beta P'Q} \quad \frac{Q \to_\beta Q'}{PQ \to_\beta PQ'} \quad \frac{P \to_\beta P'}{\lambda x.P \to_\beta \lambda x.P'}$$

Parallel (one-step) β reduction:

$$\frac{}{x \Rightarrow_\beta x} \quad \frac{P \Rightarrow_\beta P' \quad Q \Rightarrow_\beta Q'}{(\lambda x.P)Q \Rightarrow_\beta P'[Q'/x]} \quad \frac{P \Rightarrow_\beta P' \quad Q \Rightarrow_\beta Q'}{PQ \Rightarrow_\beta PQ'} \quad \frac{P \Rightarrow_\beta P'}{\lambda x.P \Rightarrow_\beta \lambda x.P'}$$

Then the *multiple-step β reduction* ($\twoheadrightarrow_\beta$) is the reflexive transitive closure of $\to_\beta$. A term is normal, if it has no β-redex in it as subterm.

Example 4.3. The Church Numerals are

$$\lambda\lambda 1^n 0 \qquad \lambda y.\lambda x.y^n x$$

Here $1^n 0$ means that it is $1^0 0 \coloneqq 0$, $1^{n+1} \coloneqq 1((1^n)0)$. These are normal terms.

Successor function is

$$\text{succ} \coloneqq \lambda\lambda\lambda 1((21)0), \qquad \lambda n.\lambda y.\lambda x.y((ny)x)$$

Prove that succ $2 \twoheadrightarrow_\beta 3$

$$\begin{aligned} \text{succ } 2 &= (\lambda n.\lambda y.\lambda x.y((ny)x))\lambda y.\lambda x.y(yx) \\ &\to_\beta \lambda y.\lambda x.y((\lambda x.y(yx))x) \\ &\to_\beta \lambda y.\lambda x.y(y(yx)) = 3 \end{aligned}$$

Remark 4.5. As succ, all the primitive recursive operation can be implemented in lambda calculus, as well as minimalization. This is called *λ-definability* The type free lambda calculus is Turing-complete, in the sense that all λ-definable arithmetical function is general recursive function and vice versa. STT captures the primitive recursive functions.

4.3 Combinators

The *combinators*

$$\begin{aligned} \mathbf{I} &\coloneqq \lambda x.x \\ \mathbf{S} &\coloneqq \lambda x.\lambda y.\lambda z.xz(yz) \\ \mathbf{K} &\coloneqq \lambda x.\lambda y.x \end{aligned}$$

are typeable and they are the Hilbert axioms for implicational logic.

The combinator

$$\omega := \lambda x.xx$$

is not typable, hence is not an inhabitant of any type, in fact it is not a proof. However, it is an interesting combinator, since

$$\Omega := \omega\omega$$

cannot be normalized:

$$\Omega = (\lambda x.xx)(\lambda x.xx) \to_\beta xx[\lambda x.xx/x] = (\lambda x.xx)(\lambda x.xx) = \Omega$$

It is an infinite reduction sequence

$$\Omega \to_\beta \Omega \to_\beta \Omega \to_\beta \dots$$

unable to terminate in a normal form. Another notable example is the fix-point recursion combinator:

$$\mathbf{Y} := \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

By this, we can give an example for an infinite pairwise distinct reduction sequence:

$$\mathbf{Y} \to_\beta \mathbf{Y}' \to_\beta \mathbf{Y}'' \to_\beta \mathbf{Y}''' \to_\beta \mathbf{Y}'''' \to_\beta \dots$$

using $\mathbf{Y}$, every recursively defined function can be expressed as a lambda expression.

4.4 Church–Rosser property

Church–Rosser property for $\twoheadrightarrow_\beta$ is the following. For any term M, if $M \twoheadrightarrow_\beta M_1$ and $M \twoheadrightarrow_\beta M_2$ then there is a term M_3 such that $M_1 \twoheadrightarrow_\beta M_3$ and $M_2 \twoheadrightarrow_\beta M_3$:

$$\begin{array}{ccc} M & \twoheadrightarrow & M_2 \\ \downarrow & & \vdots \\ M_1 & \cdots\twoheadrightarrow & M_3 \end{array}$$

This is not true for one-step reduction, since one can choose an unlucky path:

$$\begin{array}{ccccc} M = \omega(\mathbf{II}) & & \longrightarrow & & M_2 = \omega\mathbf{I} \\ \downarrow & & & & \downarrow \\ M_2 = (\mathbf{II})(\mathbf{II}) & \longrightarrow & \mathbf{I}(\mathbf{II}) & \longrightarrow & M_3 = \mathbf{II} \end{array}$$

Theorem 4.3 (Church–Rosser)**.** Pure type free lambda calculus has the C–R property.

Since, it is more like a "strong" type theorem, like Strong Normalization, we do not prove it. Just to note that, M_3 will be the so-called complete development of M, defined as follows:

Definition 4.6. The complete development of M, defined as follows:

1. $x^c := x$
2. $(PQ)^c := P^c Q^c$ (if P does not start with λ)
3. $(\lambda x.P)^c := \lambda x.P^c$
4. $((\lambda x.P)Q)^c := P^c[Q^c/x]$

And then the C–R property has to proven for the new parallel reduction $\Rightarrow_\beta$, $x \Rightarrow_\beta x$ and $PQ \Rightarrow_\beta P'Q'$ is also included into the definition in case of $P \Rightarrow_\beta P'$ and $Q \Rightarrow_\beta Q'$.

4.5 Algorithmic proof search

The algorithmic decidability of natural deduction style system of propositional logic was shown by Gentzen before the notion of formal algorithm has been established. We are going to prove it in the typed lambda calculus and the basic idea is that we are going to search for normal proofs in the so-called *long normal form*. Then we will prove that the algorithm stops within finite steps.

Definition 4.7. Let $\Gamma \vdash M : A$. M is in *long normal form,* if there are variables $x_1, \ldots x_m, y$ and normal terms $N_1, \ldots N_n$ such that

$$M = \lambda x_1. \ldots \lambda x_m. y N_1 \ldots N_n$$

Remark 4.6. In de Bruijn notation, long normal form of the typed expression $\Gamma \vdash M : A$ is the following:

$$\Gamma \vdash \lambda A_1 \ldots \lambda A_m \; k \; N_1 \ldots N_n : A$$

Remark 4.7. Note that, a long normal form is a normal form, since it contains no β-redex.

Lemma 4.1. If N is in normal form and $\Gamma \vdash N : A$, then N is in a long normal form.

Proof. Case $N = 0$ is trivial: $\Gamma \vdash 0 : A$.

Suppose $N = \lambda B\, P$, by normality $B :: \Gamma \vdash P : C$ with $A = B \to C$, then for P the claim holds by the induction hypothesis, so $P = \lambda A_1 \ldots \lambda A_m \; k \; N_1 \ldots N_n$ is in long normal form and

$$B :: \Gamma \vdash \lambda A_1 \ldots \lambda A_m \; k \; N_1 \ldots N_n : C$$

hence

$$\Gamma \vdash \lambda A' \lambda A_1 \dots \lambda A_m \; k \; N_1 \dots N_n : A.$$

Suppose $N = PQ$, i.e. by normality $\Gamma \vdash P : B \to A$, $\Gamma \vdash Q : B$. Then by normality, the lambda part is missing (otherwise it would be a β redex)

$$P = k \; N_1 \dots N_n$$

hence

$$\Gamma \vdash (k \; N_1 \dots N_n)Q : A$$

where Q is in long normal form, and a fortiori is in normal form. □

Remark 4.8. The long normal form is the type theoretic counterpart of a main branch in implicational logic. For instance

$$\dfrac{\dfrac{\dfrac{p_2 : A \to (D \to E) \to C \quad p_1 : A}{\to^E p_2 p_1 : (D \to E) \to C}\to^E \quad \dfrac{p_4 : E}{\to^I D p_4 : D \to E}\to^I}{\to^E \to^E p_2 p_1 \to^I D p_4 : C}\to^E}{\to^I B \to^E \to^E p_2 p_1 \to^I D p_4 : B \to C}\to^I}{\to^I A \to^I B \to^E \to^E p_2 p_1 \to^I D p_4 : A \to B \to C}\to^I$$

In this sense, a long normal form is a main branch of the form:

$$\to^I A_1 \cdots \to^I A_m \to^E \cdots \to^E p_k N_1 \dots N_n$$

Remark 4.9. When we extend the language by $\wedge$ and $\wedge_i^E$ and $\wedge^I$, the long normal form and the branch theorem can be generalized. In this case, there will be no "the" main branch just "a" branch.

$$\dfrac{A \qquad \dfrac{\dfrac{\dfrac{B \wedge D \to C \quad B \wedge D}{C}\to^E \quad \dfrac{B \wedge D}{D}\wedge_2^E}{C \wedge D}\wedge^I}{B \to C \wedge D}\to^I}{A \wedge (B \to C \wedge D)}\wedge^I$$

Here a code of a main branch (starts at $B \wedge D$) is

$$\wedge^I N_1 \to^I B \wedge^I \to^E p_k N_2$$

or in a notation, where we introduce left and right conjunctions ($\wedge_l$, $\wedge_r$) in order to arrange the non selected branches into the second position:

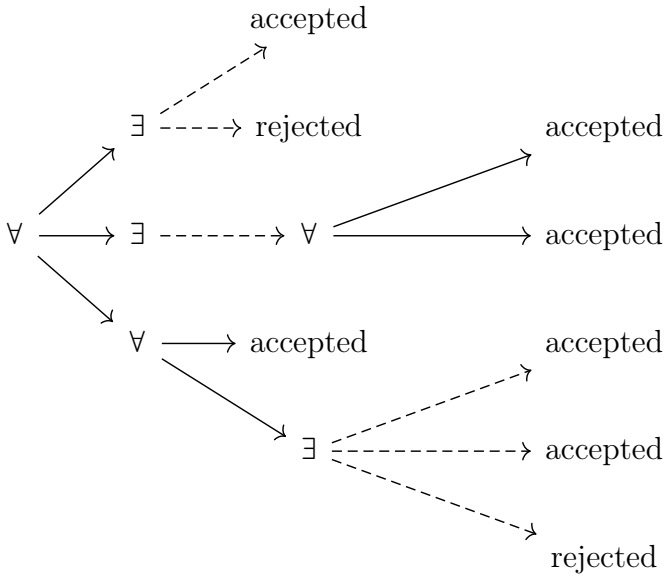
$$\wedge_r^I \to^I B \wedge_l^I \to^E p_k N_2 N_1$$

Then, the general form of a long normal form is:

$$*^I \cdots *^I \, (*^E \cdots *^E \, p_k N_1 \ldots N_n) M_1 \ldots M_s$$

where s is the number of $\wedge^I$.

An Alternating Turing Machine (ATM) is a generalization of a Non-Deterministic Turing Machine. In an ATM there are two kind of state: universal and existential. A universal state is said to be accepted when all the children of node in the computational tree are accepted and existential if at least one of its children in the computational tree is accepted. The TIME of the ATM is the height of the computational tree in terms of the input size. It follows that an APTIME ATM who has only existential nodes is an NP NTM, since it arrives into an accept state within polynomial time, if accepts.



Theorem 4.4. There is an APTIME algorithm to determine whether a type A is inhabited in the context Γ.

Proof. We know that, if a type is inhabited, then it has an inhabitant which is in normal form and this inhabitant is also in a long normal form. The algorithm goes as follows

1. If $A = A_1 \to A_2$, then extend Γ as $A_1 :: \Gamma$, then produce the output $\lambda A_1 ?$, store the question in a stack "$A_1 :: \Gamma \vdash ? : A_2$" and call $A_1 :: \Gamma \vdash ? : A_2$.

2. If A is atomic, then choose non-deterministically a type in Γ of the form

   $$A_1 \to A_2 \ldots A_{n+1}$$

   such that $A_{n+1} = A$. When there is no such a type, reject. If $n = 0$, then accept. If $n > 0$, then in the k^{th} position of Γ store the questions "$\Gamma \vdash ?_i : A_i$" in the stack, create the output $p_k ?_1 \ldots ?_n$ and call $\Gamma \vdash ?_i : A_i$.

If in any step the question is already in the stack, then reject avoiding loops.

Note that Γ can and may extends at least $|\mathrm{Sub}\Gamma \cup \{A\}|$ times along a single computational path, where $\mathrm{Sub}(B)$ denotes the subtypes of B. Similarly, the number of questions in a context is at most then $|\mathrm{Sub}\Gamma \cup \{A\}|$. Hence the height of the computational tree is at most $|\mathrm{Sub}\Gamma \cup \{A\}|^2$. □

5 Dependent types

One can extend STT by a simple way, with bool and nat, and by a dependent way by nat and dependent product and sum. First, let's see the extension with nat.

$$\frac{}{\mathrm{nat} : \mathrm{Type}} \quad \text{(nat formáció)}$$

$$\frac{}{0 : \mathrm{nat}} \quad \text{(nat intro 1)} \qquad \frac{\Gamma \vdash \mathrm{n} : \mathrm{nat}}{\Gamma \vdash \mathrm{S}\, n : \mathrm{nat}} \quad \text{(nat intro 2)}$$

így tehát nat-beli elemek 0, S 0, S S 0, ...

$$\frac{\Gamma, x : \mathrm{nat} \vdash P : \mathrm{Type} \quad \Gamma \vdash u : P[0/x] \quad \Gamma, y : \mathrm{nat}, z : P[y/x] \vdash v : P[\mathrm{S}\, y/x] \quad \Gamma \vdash t : \mathrm{nat}}{\Gamma \vdash \mathrm{rect}_{\mathrm{nat}}[x.P](u, y.z.v, t) : P[t/x]} \quad \text{(nat elim)}$$

$\mathrm{rect}_{\mathrm{nat}}[x.P](u, y.z.v, t)$ realizálható a nat-tal és konstruktoraival bővített STT-ben, primitív rekurzív módon a t felépítésére vonatkozó strukturális indukcióval. Ez magyarul azt jelenti, hogy a fenti feltételekkel nat minden (konstruált) t elemére meg tudunk adni egy olyan STT termet, ami lakója a $P[t/x]$ típusnak. Természetesen, mivel nat-nak végtelen sok eleme van, ezért ezt iterációval (pr. rekurzióval) adjuk meg.

primitive recursion $\mathrm{rect}_{\mathrm{nat}}[x.P](u, y.z.v, t)'$
match t : nat **with**
$0 \quad \Rightarrow \quad \mathrm{rect}_{\mathrm{nat}}[x.P](u, y.z.v, 0)' \quad := u$
$\mathrm{S}\, n \quad \Rightarrow \quad \mathrm{rect}_{\mathrm{nat}}[x.P](u, y.z.v, \mathrm{S}\, n)' \quad := v[n/y, \mathrm{rect}_{\mathrm{nat}}[x.P](u, y.z.v, n)'/z]$
end match
end primitive recursion

Nem kizárt, hogy P speciális alakja esetén lehetne másképp is realizálni rect-et. Ezt majd egyszerűbb adattípusoknál jobban látjuk. A fenti realizáció azonban kanonikus, mert minden P-re konkluzív. A komputációs szabályok azt a megállapodást rögzítik, hogy a fenti realizáció azonos rect-tel:

$$\frac{}{\Gamma \vdash \ \mathrm{rect}_{\mathrm{nat}}[x.P](u, y.z.v, 0) \equiv u : P[0/x]} \quad \text{nat } \beta \text{ 1}$$

$$\frac{}{\Gamma \vdash \ \mathrm{rect}_{\mathrm{nat}}[x.P](u, y.z.v, \mathrm{S}\, n) \equiv v[n/y, \mathrm{rect}_{\mathrm{nat}}[x.P](u, y.z.v, n)/z] : P[\mathrm{S}\, n/x]} \quad \text{nat } \beta \text{ 2}$$

Itt $[x.P]$-t motívumnak hívjuk, mert ebből lesz a céltipíus.

A fenti realizációs módon lehet nat-on értelmezett függvényeket és relációkat is definiálni. Ezeket a komputációs szabályok révén a tárgynyelvben rect-tel is meg tudjuk fogalmazni. Itt van például az nat-beli nullával való egyenlőség egy algoritmikus definíciója:

primitive recursion nat.eq.zero'(t) : nat
match t : nat **with**
$0 \quad \Rightarrow \quad$ nat.eq.zero'$(0) \quad := 1$
$\mathrm{S}\, n \quad \Rightarrow \quad$ nat.eq.zero'$(\mathrm{S}\, n)' \quad := 0$
end match
end primitive recursion

bár nat.eq.zero'(t) minden egyes eleme már az STT nyelvén készült el, a komputációs szabályokkal lehet egyetlen kifejezésben is hivatkozni rá.

$$\text{nat.eq.zero'}(0) \equiv 1 \qquad \text{nat.eq.zero'}(\mathrm{S}\, n) \equiv 0$$

tehát (és az $[x.\text{nat}]$ motívumot nem kiírva):

$$\text{nat.eq.zero}(t) := \mathrm{rect}_{\mathrm{nat}}(1, y.z.0, t)$$

és ellenőrizve a komputációs szabályokkal:

$$\mathrm{rect}_{\mathrm{nat}}(1, y.z.0, 0) \equiv 1 \qquad \mathrm{rect}_{\mathrm{nat}}(1, y.z.0, \mathrm{S}\, n) \equiv 0[n/y, \mathrm{rect}_{\mathrm{nat}}(1, y.z.0, n)/z]$$

Persze a 0 és 1 számok nem a legalkalmasabbak az igazságértékek reprezentációjára, egyfelől, mert a típuselméletben a "részhalmaz" nem egy értelmes dolog (csak szigma típusként, lásd később). Másfelől mert vészesen kötődik a kétértékű logikához, ami miatt inkább transparensen érdemes szétválasztani két logikát. Ezért lesz a Type_1-n belül két külön szort: a Prop és a Set (ezek elegendőek is, de ha kell ott van még Type_0).

Házi feladat: realizáljuk az összeadást és adjuk meg a rect-tel kifejezett alakját!

5.1 True

$$\frac{}{\mathrm{True} : \mathrm{Prop}} \text{ (True formáció)} \qquad \frac{}{\mathrm{tt} : \mathrm{True}} \text{ (True bevezetés)}$$

$$\frac{\Gamma, x : \mathrm{True} \vdash P : \mathrm{Type} \quad \Gamma \vdash u : P[\mathrm{tt}/x] \quad \Gamma \vdash t : \mathrm{True}}{\Gamma \vdash \mathrm{rect}_{\mathrm{True}}[x.P](u, t) : P[t/x]} \text{ (True elimináció)}$$

$$\frac{\Gamma, x : \mathrm{True} \vdash P : \mathrm{Prop} \quad \Gamma \vdash u : P[\mathrm{I}/x]}{\Gamma \vdash \mathrm{ind}_{\mathrm{True}} u I \equiv u : P[I/x]} \text{ (True komputáció)}$$

Konstans állítás esetén:

$$\frac{\Gamma \vdash P : \mathrm{Prop} \quad \Gamma \vdash u : P \quad \Gamma \vdash t : \mathrm{True}}{\Gamma \vdash \mathrm{ind}_{\mathrm{True}} u t : P} \text{ (True elimináció)}$$

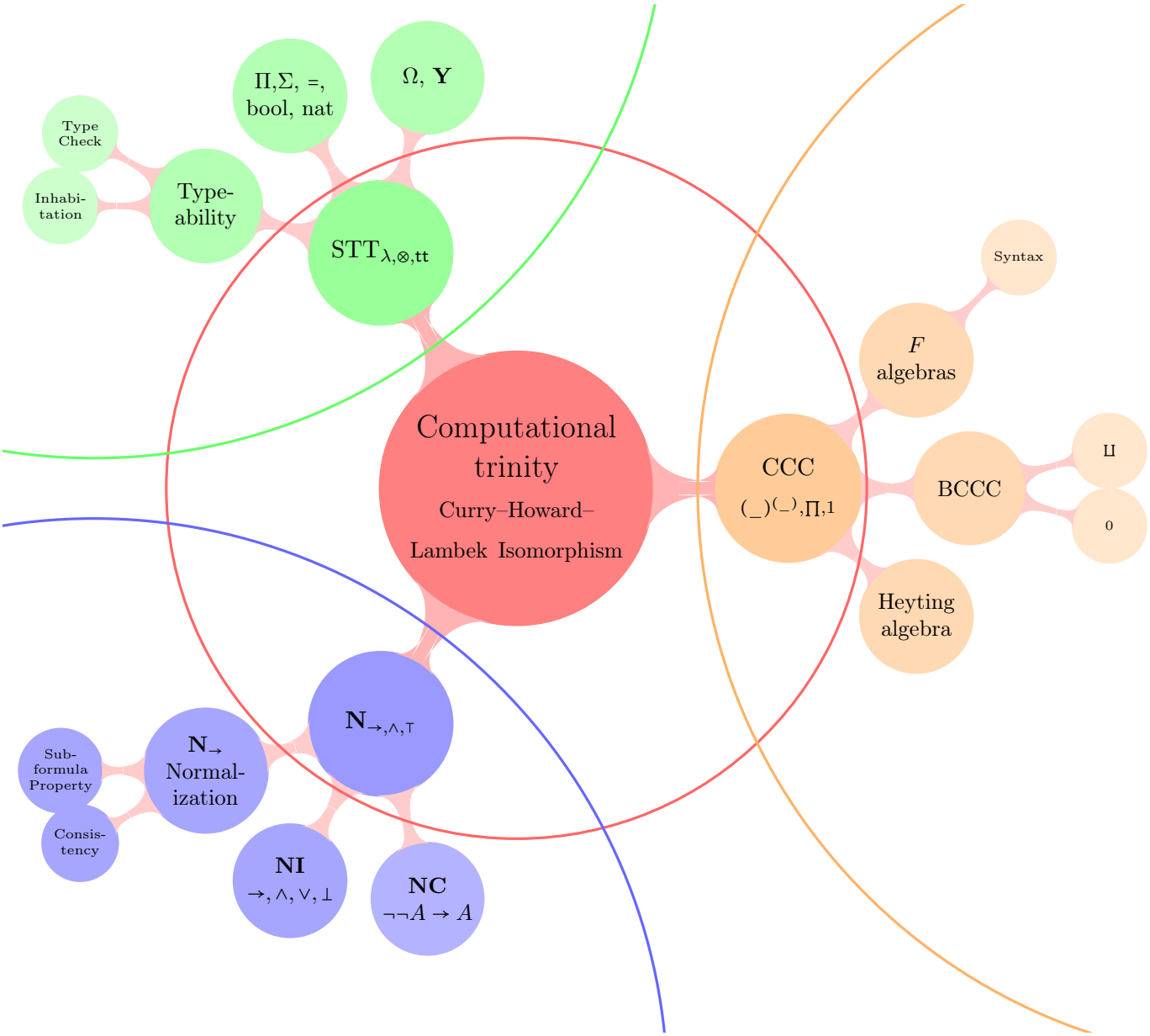
$$\frac{\Gamma \vdash P : \mathrm{Prop} \quad \Gamma \vdash u : P \quad \Gamma \vdash t : \mathrm{True}}{\Gamma \vdash \mathrm{ind}_{\mathrm{True}} u \mathrm{I} \equiv u : P} \text{ (True komputáció)}$$

:

6 Summary

The main idea behind the course is that there is a close connection between the *typed program languages* (Simply Typed Lambda Calculus, $\mathrm{STT}_{\lambda,\otimes,\mathtt{tt}}$) and *proofs of propositions* (Natural Deduction System of Implication, Conjunction and True, $\mathbf{N}_{\to,\wedge,\top}$). As it is shown, this connection is not just isomorphical, but identical, when the two system is presented in the de Bruijn style. In the one hand, this identity is a strictly claimed mathematical proposition, the **Curry–Howard Isomoprhism.** For a closer look, let see the function type or implication. In the propositional level, the type $A \to B$ is the intended type of computable functions from a A to B, and the intuitionistical proposition $A \to B$ is the claim that there is a proof scheme such that, once we have a proof of A, then included it in into the scheme, we also have a proof of B. Here proof schemes, can be considered as a *construction* in the *Brouwer–Heyting–Kolmogorov interpretation* of constructive (or intuitionistic) logic, i.e. computable functions. On the other hand, Curry–Howard Correspondence is a paradigm to establish connection between any kind of natural deduction system and typed programming languages. For instance, the dependent function type $\Pi_{x:A}B$ corresponds to the quantified proposition $(\forall x : A)B$.

While the adequate semantics of classical logic is the Boolean Algebra (or Boolean Lattices), the **adequate semantics** for the intuitionistic logic is *Heyting Algebra.* What's more, when the β and η convertibility is required (the definitional equivalence of proofs), then the correspondence raised to the level is Cartesian Closed Categories (CCC). The **Curry–Howard–Lambek Isomorphism** states the the STT with $\beta\eta$ rules, NI with $\beta\eta$ rules and CCC are pairwise isomorphic.


Computational trinity
Curry–Howard–
Lambek Isomorphism
STT$_{\lambda,\otimes,\mathsf{tt}}$
Π,Σ, =, bool, nat
Ω, **Y**
Type-ability
Type Check
Inhabi-tation
N$_{\to,\wedge,\top}$
N$_{\to}$ Normal-ization
Sub-formula Property
Consis-tency
NI
$\to, \wedge, \vee, \bot$
NC
$\neg\neg A \to A$
CCC
$(\_)^{(\_)}$,Π,1
F algebras
Syntax
BCCC
Ш
0
Heyting algebra

References

de Bruijn, N. G. (1994). Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Studies in logic and the foundations of mathematics*, 133:375–388.

Dummett, M. (1991). *The Logical Basis of Metaphysics*. Harvard University Press.

Fortune, S., Leivant, D., and O'Donnell, M. J. (1983). The expressiveness of simple and second-order type structures. *J. ACM*, 30:151–185.

Gentzen, G. (1969). *The Collected Papers of Gerhard Gentzen*. Amsterdam: North-Holland Pub. Co.

Lambek, J. and Scott, P. J. (1986). *Introduction to higher order categorical logic*. Cambridge University Press.

Milewski, B. (2019). *Category theory for programmers*. Bartosz Milewski.

Pfenning, F. and Paulin-Mohring., C. (1990). Iductively defined types in the calculus of constructions. In *Proceedings 5th Int. Conf. on Math. Foundations of Programming Semantics, MFPS'89, New Orleans, LA, USA, 29 Mar – 1 Apr 1989*, volume 442 of *Lecture Notes in Computer Science (LNCS)*, page 209–228.

Prawitz, D. (1965). *Natural Deduction: A Proof-Theoretical Study*. Dover.