



**synrc** research center s.r.o.  
ROHÁČOVA 141/18, PRAHA 3 13000, CZECH REPUBLIC

# **Categorical Foundations for Fibrational Theory / Lambek Encoding**

Technical Article

Maxim Sokhatsky, Synrc Research Center

Kyiv 2016

# Contents

<b>1</b>	<b>Types and Categories</b>	<b>3</b>
1.1	Category Theory, Programs and Functions . . . . .	3
1.2	Algebraic Types and Cartesian Categories . . . . .	4
1.3	Exponential, $\lambda$ -calculus and Cartesian Closed Categories . . . . .	5
1.4	$\lambda$ -language . . . . .	5
1.5	Functors, $\Lambda$ -calculus . . . . .	5
1.6	Algebras . . . . .	6
1.7	Initial Algebras . . . . .	6
1.8	Recursive Types . . . . .	6
1.9	Intuitionistic Type Theory . . . . .	7
1.10	Logic and Quantification . . . . .	7

# 1 Types and Categories

## 1.1 Category Theory, Programs and Functions

Category theory is widely used as an instrument for mathematicians for software analysis. Category theory could be treated as an abstract algebra of functions. Let's define an Category formally: **Category** consists of two lists: the one is morphisms (arrows) and the second is objects (domains and codomains of arrows) along with associative operation of composition and unit morphism that exists for all objects in category.

The formation axioms of objects and arrows are not given here and autopostulating yet. Formation axioms will be introduced during exponential definition. Objects  $A$  and  $B$  of an arrow  $f : A \rightarrow B$  are called **domain** and **codomain** respectively.

Intro axioms – associativity of composition and left/right unit arrow compositions show that categories are actually typed monoids, which consist of morphisms and operation of composition. There are many languages to show the semantic of categories such as commutative diagrams and string diagrams however here we define here in proof-theoretic manner:

$$\begin{array}{c}
 \frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash g : B \rightarrow C}{\Gamma \vdash g \circ f : A \rightarrow C} \qquad \frac{}{\Gamma \vdash id_A : A \rightarrow A} \\
 \\
 \frac{\Gamma \vdash f : B \rightarrow A \quad \Gamma \vdash g : C \rightarrow B \quad \Gamma \vdash h : D \rightarrow C}{\Gamma \vdash (f \circ g) \circ h = f \circ (g \circ h) : D \rightarrow A} \qquad \frac{\Gamma \vdash f : A \rightarrow B}{\Gamma \vdash f \circ id_A = f : A \rightarrow B} \\
 \\
 \frac{}{\Gamma \vdash id_B \circ f = f : A \rightarrow B}
 \end{array}$$

Composition shows an ability to connect the result space of the previous evaluation (codomain) and the arguments space of the next evaluation (domain). Composition is fundamental property of morphisms that allows us to chain evaluations.

1.  $A : *$
2.  $A : * , B : * \implies f : A \rightarrow B$
3.  $f : B \rightarrow C , g : A \rightarrow B \implies f \circ g : A \rightarrow C$
4.  $(f \circ g) \circ h = f \circ (g \circ h)$
5.  $A \implies id : A \rightarrow A$
6.  $f \circ id = f$
7.  $id \circ f = f$

## 1.2 Algebraic Types and Cartesian Categories

After composition operation of construction of new objects with morphisms we introduce operation of construction cartesian product of two objects  $A$  and  $B$  of a given category along with morphism product  $\langle f, g \rangle$  with a common domain, that is needed for full definition of cartesian product of  $A \times B$ .

This is an internal language of cartesian category, in which for all two selected objects there is an object of cartesian product (sum) of two objects along with its  $\perp$  terminal (or  $\top$  coterminial) type. Exe languages is always equiped with product and sum types.

Product has two eliminators  $\pi$  with an common domain, which are also called projections of an product. The sum has eliminators  $i$  with an common codomain. Note that eliminators  $\pi$  and  $i$  are isomorphic, that is  $\pi \circ \sigma = \sigma \circ \pi = id$ .

$$\begin{array}{c}
\frac{\Gamma x : A \times B}{\Gamma \vdash \pi_1 : A \times B \rightarrow A; \Gamma \vdash \pi_2 : A \times B \rightarrow B} \quad \frac{}{\Gamma \vdash \perp} \\
\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B} \quad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash a \mid b : A \otimes B} \\
\frac{}{\Gamma \vdash \top} \quad \frac{\Gamma x : A \otimes B}{\Gamma \vdash \beta_1 : A \rightarrow A \otimes B; \Gamma \vdash \beta_2 : B \rightarrow A \otimes B}
\end{array}$$

The  $\perp$  type in Haskell is used as **undefined** type (empty sum component presented in all types), that is why Hask category is not based on cartesian closed but CPO [?]. The  $\perp$  type has no values. The  $\top$  type is known as unit type or zero tuple () often used as an default argument for function with zero arguments. Also we include here an axiom of morphism product which is given during full definition of product using commutative diagram. This axiom is needed for applicative programming in categorical abstract machine. Also consider co-version of this axiom for  $[f, g] : B + C \rightarrow A$  morphism sums.

$$\begin{array}{c}
\frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash g : A \rightarrow C \quad \Gamma \vdash B \times C}{\Gamma \vdash \langle f, g \rangle : A \rightarrow B \times C} \\
\begin{array}{l}
\pi_1 \circ \langle f, g \rangle = f \\
\pi_2 \circ \langle f, g \rangle = g \\
\langle f \circ \pi_1, f \circ \pi_2 \rangle = f \\
\langle f, g \rangle \circ h = \langle f \circ h, g \circ h \rangle \\
\langle \pi_1, \pi_2 \rangle = id
\end{array}
\end{array}$$

### 1.3 Exponential, $\lambda$ -calculus and Cartesian Closed Categories

Being an internal language of cartesian closed category, lambda calculus except variables and constants provides two operations of abstraction and applications which defines complete evaluation language with higher order functions, recursion and corecursion, etc.

To explain functions from the categorical point of view we need to define categorical exponential  $f : A^B$ , which are analogue to functions  $f : A \rightarrow B$ . As we already defined the products and terminals we could define an exponentials with three axioms of function construction, one eliminator of application with apply a function to its argument and axiom of currying the function of two arguments to function of one argument.

$$\begin{array}{c}
\frac{\Gamma x : A \vdash M : B}{\Gamma \vdash \lambda x . M : A \rightarrow B} \\
\\
\frac{\Gamma f : A \rightarrow B \quad \Gamma a : A}{\Gamma \vdash \text{apply } f \ a : (A \rightarrow B) \times A \rightarrow B} \\
\\
\frac{\Gamma \vdash f : A \times B \rightarrow C}{\Gamma \vdash \text{curry } f : A \rightarrow (B \rightarrow C)}
\end{array}
\qquad
\begin{array}{l}
\text{apply} \circ \langle (\text{curry } f) \circ \pi_1, \pi_2 \rangle = f \\
\text{curry } \text{apply} \circ \langle g \circ \pi_1, \pi_2 \rangle = g \\
\text{apply} \circ \langle \text{curry } f, g \rangle = f \circ \langle \text{id}, g \rangle \\
(\text{curry } f) \circ g = \text{curry } (f \circ \langle g \circ \pi_1, \pi_2 \rangle) \\
\text{curry } \text{apply} = \text{id}
\end{array}$$

### 1.4 $\lambda$ -language

$$\begin{array}{l}
\text{Objects : } \perp \mid \rightarrow \mid \times \\
\text{Morphisms : } \text{id} \mid f \circ g \mid \langle f, g \rangle \mid \text{apply} \mid \lambda \mid \text{curry}
\end{array}$$

### 1.5 Functors, $\Lambda$ -calculus

Functor comes as a notion of morphisms in categories whose objects are categories. Functors preserve compositions of arrows and identities, otherwise it would be impossible to deal with categories. One level up is notion of morphism between categories whose objects are Functors, such morphisms are called natural transformations. Here we need only functor definition which is needed as general type declarations.

$$\begin{array}{c}
\frac{\Gamma \vdash f : A \rightarrow B}{\Gamma \vdash F f : (A \rightarrow B) \rightarrow (F A \rightarrow F B)} \\
\\
\frac{\Gamma \vdash \text{id}_A : A \rightarrow A}{\Gamma \vdash F \text{id}_A = \text{id}_{FA} : F A \rightarrow F A} \\
\\
\frac{\Gamma \vdash f : B \rightarrow C, g : A \rightarrow B}{\Gamma \vdash F f \circ F g = F(f \circ g) : F A \rightarrow F C}
\end{array}$$

We start thinking of functors on dealing with typed theories, because functors usually could be seen as higher order type con

## 1.6 Algebras

F-Algebras gives us a categorical understanding recursive types. Let  $F : C \rightarrow C$  be an endofunctor on category  $C$ . An F-algebra is a pair  $(C, \phi)$ , where  $C$  is an object and  $\phi : F C \rightarrow C$  an arrow in the category  $C$ . The object  $C$  is the carrier and the functor  $F$  is the signature of the algebra. Reversing arrows gives us F-Coalgebra.

$$\begin{array}{ccc}
 F C & \xrightarrow{\varphi} & C \\
 F f \downarrow & & \downarrow f \\
 F D & \xrightarrow{\psi} & D
 \end{array}
 \qquad
 \begin{array}{ccc}
 C & \xrightarrow{\varphi} & F C \\
 f \downarrow & & \downarrow F f \\
 D & \xrightarrow{\psi} & F D
 \end{array}$$

$$f \circ \varphi = \psi \circ F f \qquad \psi \circ f = F f \circ \varphi$$

## 1.7 Initial Algebras

A F-algebra  $(\mu F, in)$  is the initial F-algebra if for any F-algebra  $(C, \varphi)$  there exists a unique arrow  $\llbracket \varphi \rrbracket : \mu F \rightarrow C$  where  $f = \llbracket \varphi \rrbracket$  and is called catamorphism. Similar a F-coalgebra  $(\nu F, out)$  is the terminal F-coalgebra if for any F-coalgebra  $(C, \varphi)$  there exists unique arrow  $\llbracket \varphi \rrbracket : C \rightarrow \nu F$  where  $f = \llbracket \varphi \rrbracket$

$$\begin{array}{ccc}
 F \mu F & \xrightarrow{in} & \mu F \\
 F \llbracket \varphi \rrbracket \downarrow & & \downarrow \llbracket \varphi \rrbracket \\
 F C & \xrightarrow{\varphi} & C
 \end{array}
 \qquad
 \begin{array}{ccc}
 C & \xrightarrow{\phi} & F C \\
 \llbracket \varphi \rrbracket \downarrow & & \downarrow F \llbracket \varphi \rrbracket \\
 \nu F & \xrightarrow{out} & F \nu F
 \end{array}$$

$$f \circ in = \varphi \circ F f \equiv f = \llbracket \varphi \rrbracket \qquad out \circ f = F f \circ \varphi \equiv f = \llbracket \varphi \rrbracket$$

## 1.8 Recursive Types

As was shown by Wadler [?] we could deal with recursive equations having three axioms: one  $fix : (A \rightarrow A) \rightarrow A$  fixedpoint axiom, and axioms  $in : F T \rightarrow T$  and  $out : T \rightarrow F T$  of recursion direction. We need to define fixed point as axiom because we can't define recursive axioms. This axioms also needs functor axiom defined earlier.

$$\frac{\Gamma \vdash M : F (\mu F)}{\Gamma \vdash in_{\mu F} M : \mu F}
 \qquad
 \frac{\Gamma \vdash M : A \rightarrow A}{\Gamma \vdash fix M : A}$$

$$\frac{\Gamma \vdash M : \mu F}{\Gamma \vdash out_{\mu F} M : F (\mu F)}$$

## 1.9 Intuitionistic Type Theory

Using Build Yourself Type Theory approach sooner or later you should decide the palette of inductive structures. Such in Coq abstract algebra framework was built upon polymorphic records [?] rather than type classes engine like it is used in Agda and Idris. However Idris still lacks of polymorphic records and coinductive types. Lean also lacks of coinductive structures but has powerful non-recursive polymorphic records which are used in Lean HoTT library.

As was shown by Stephan Kaes [?], one of the strategies of type class engine implementation is using polymorphic structures which allows us to deal with persistent structures on a low theoretical level. Moreover this style of coding is completely compatible with Erlang records which are used to model KVS and N2O hierarchies.

## 1.10 Logic and Quantification

Dependent product is generalisation of exponential or function definition, when domain of morphism is dependent on codomain. Dependent sum is generalisation of cartesian product when one element depends on another. Dependent product is noted as  $\forall$  and dependent sum is noted as  $\exists$ :

$$\begin{array}{c}
\frac{\Gamma x : A \vdash B \quad \Gamma \vdash A}{\Gamma \vdash \Pi(x : A)B} \qquad \frac{\Gamma \vdash a : A \quad \Gamma x : A \vdash B \quad \Gamma b : B(x = a)}{\Gamma \vdash (a, b) : \Pi(x : A)B} \\
\\
\frac{\Gamma x : A \vdash B \quad \Gamma \vdash A}{\Gamma \vdash \Sigma(x : A)B} \qquad \frac{\Gamma \vdash a : A \quad \Gamma x : A \vdash B \quad \Gamma b : B(x = a)}{\Gamma \vdash (a, b) : \Sigma(x : A)B} \\
\\
\frac{\Gamma \vdash x : A \quad \Gamma \vdash x' : A}{\Gamma \vdash Id_A(x, x')}
\end{array}$$

$$\begin{array}{ll}
\text{reflexivity} & : Id_A(a, a) \\
\text{substitution} & : Id_A(a, a') \rightarrow B(x = a) \rightarrow B(x = a') \\
\text{symmetry} & : Id_A(a, b) \rightarrow Id_A(b, a) \\
\text{transitivity} & : Id_A(a, b) \rightarrow Id_A(b, c) \rightarrow Id_A(a, c) \\
\text{congruence} & : (f : A \rightarrow B) \rightarrow Id_A(x, x') \rightarrow Id_B(f(x), f(x'))
\end{array}$$

## References