



Groupoid Infinity

KOSTIANTYNIIVSKA 20/14, KYIV, UKRAINE 04071

Categorical Encoding of Inductive Types

Paul Lyutko, Maxim Sokhatsky

Groupoid Infinity

Kamianets-Podilsky 2016

Contents

1	Abstract	3
2	Type Theory	4
2.1	Intuitionistic Type Theory	4
2.2	Universes	5
2.3	Dependent Types	5
2.4	Identity Types	6
2.5	Inductive Types	6
3	Category Theory	7
3.1	Programs and Functions	7
3.2	Algebraic Types and Cartesian Categories	8
3.3	Exponential, λ -calculus and Cartesian Closed Categories	9
3.4	Functors, Λ -calculus	10
3.5	Algebras	11
3.6	Initial Algebras	11
3.7	Recursive Types	11
4	Categorical Encoding	12
4.1	List Sample	12
4.2	Basic Ornaments	13
4.2.1	Unit	13
4.2.2	Fixed Type	13
4.2.3	Recursive Parameters	13
4.3	Recursor and Induction	15
4.3.1	Recursor	15
4.3.2	Fibrations	15
4.3.3	Induction	15
4.4	Conclusion	16

1 Abstract

From PTS to HTS. We want to have flexible detachable layers on top of PTS core. Then Sigma for proving. Then well-founded trees or polynomial functors as known as data and record. Then higher path types, interval arithmetic, glue and comp for HIT. Each layers is driven by different math, the common in only the method – category theory.

Extensible Language Design. Encoding of inductive types is based on categorical semantic of compilation to PTS. All other syntax constructions are inductive definitions, plugged into the stream parser. AST of the PTS language is also defined in terms of inductive constructions and thus allowed in the macros. The language of polynomial functors (data and record) and core language of the process calculus (spawn, receive and send) are just macrosystem over Om language, its syntax extensions.

Changable Encodings. In pure CoC we have only arrows, so all inductive type encodings would be Church-encoding variations. Most extended nowadays is Church-Boehm-Berrarducci encoding, which dedicated to inductive types. Another well known are Scott (lazyness), Parigot (lazyness and constant-time iterators) and CPS (continuations) encodings. However most of them require variations of Fixpoint types.

Proved Categorical Semantic. There was modeled a math model (using higher-order categorical logic) of encoding, which calculates (co)limits in a category of (co)algebras built with given set of (de)constructors. We call such encoding in honour of Lambek lemma that leads us to the equality of (co)initial object and (co)limit in the categories of (co)algebras. Such encoding works with dependent types and its consistency is proved in Lean model.

2 Type Theory

2.1 Intuitionistic Type Theory

Type theories are formal calculi that can be seen as both mathematical formal logics, set theories, programming languages and formalized programming logics used to prove program properties and derive a correct program from a specification [?].

We will follow the tradition in the literature on type systems [?] and formulate our calculus with judgment rules in a well-known notation of Natural Deduction. There are two types of judgement rules: **a** is a object of type **A**; **a** and **b** are definitionally equal objects of type **A**. Rules for each type can be in turn classified into **formations**, **introductions**, **eliminators** and **computational rules**.

Formation rules are typing rules, introductions (type constructors) and eliminators are adjunctions that correspond to axioms, and computational rules correspond to operational semantics. From categorical point of view **typing rules** correspond to **functors**, **axioms** to **morphisms** and **operational semantics** to **equalities on morphisms**.

Our variant of intuitionistic type system has an **infinite** hierarchy of universes, the hierarchy is **non-cumulative**, the universe of propositions is **impredicative** while universes of values, types and higher types are **predicative**. The next paragraphs briefly describe the reasons for such design choices.

While there are variants of MLTT with finite number of universes (see for example the lambda cube and PTS formulations of Calculus of Constructions in [?]), most modern provers have an infinite hierarchy of universes as it's **more expressive**.

The first universe, the universe of propositions, has impredicative quantification while all other universes are predicative. This setup is known to simplify the type system while keeping it free of paradoxes and thus **sound as a formal logic**.

The first few universes have analogs in logic and axiomatic set theories. The bottom is contractible space. Universe 0 can be thought of as the universe of propostions. Universe 1 contains elements. Universe 1 contains sets. Universe 2 contains classes. The hierarchy can be cumulative or non-cumulative. In cumulative setting lower universes are included in higher ones — e.g. all elements are sets. In non-cumulative setting the universes don't intersect, so the case is analogous to axiomatic set theories **with urelements**.

2.2 Universes

$$\frac{i : \mathit{Nat}}{\mathit{Type}_i} \quad (\text{sorts})$$

$$\frac{i : \mathit{Nat}}{\mathit{Type}_i : \mathit{Type}_{i+1}} \quad (\text{axioms})$$

$$\frac{i : \mathit{Nat}, \quad j : \mathit{Nat}}{\mathit{Type}_i \rightarrow \mathit{Type}_j : \mathit{Type}_{\max(i,j)}} \quad (\text{rules})$$

2.3 Dependent Types

$$\frac{\pi_1 : A \quad u : A \vdash \pi_2 : B}{[\pi_1/u] \pi_2 : B} \quad (\text{subst})$$

$$\frac{x : A \vdash B : \mathit{Type}}{\Pi (x : A) \rightarrow B : \mathit{Type}} \quad (\Pi\text{-formation})$$

$$\frac{x : A \vdash b : B}{\lambda (x : A) \rightarrow b : \Pi (x : A) \rightarrow B} \quad (\lambda\text{-intro})$$

$$\frac{f : (\Pi (x : A) \rightarrow B) \quad a : A}{f a : B [a/x]} \quad (\mathit{App}\text{-elimination})$$

$$\frac{x : A \vdash b : B \quad a : A}{(\lambda (x : A) \rightarrow b) a = b [a/x] : B [a/x]} \quad (\beta\text{-computation})$$

```

record  $\Pi$  (A: Type): Type :=
  ( $\Pi$ : (A  $\rightarrow$  Type)  $\rightarrow$  Type)
  (fun: (B: A  $\rightarrow$  Type)  $\rightarrow$   $\forall$  (a: A)  $\rightarrow$  B a  $\rightarrow$   $\Pi$  B)
  (app: (B: A  $\rightarrow$  Type)  $\rightarrow$   $\Pi$  B  $\rightarrow$   $\forall$  (a: A)  $\rightarrow$  B a)
  (app-fun (B: A  $\rightarrow$  Type) (f:  $\forall$  (a: A)  $\rightarrow$  B a):  $\forall$  (a: A)  $\rightarrow$  app (fun f) a ==> f a)
  (fun-app (B: A  $\rightarrow$  Type) (p:  $\Pi$  B): fun ( $\lambda$ (a: A)  $\rightarrow$  app p a) ==> p)

```

2.4 Identity Types

$$\frac{a : A \quad b : A \quad A : Type}{Id(A, a, b) : Type} \quad (Id\text{-formation})$$

$$\frac{a : A}{refl(A, a) : Id(A, a, a)} \quad (Id\text{-intro})$$

$$\frac{p : Id(a, b) \quad x, y : A \quad u : Id(x, y) \vdash E : Type \quad x : A \vdash d : E [x/y, refl(x)/u]}{J(a, b, p, (x, y, u) d) : E [a/x, b/y, p/u]} \quad (J\text{-elimination})$$

$$\frac{a, x, y : A, \quad u : Id(x, y) \vdash E : Type \quad x : A \vdash d : E [x/y, refl(x)/u]}{J(a, a, refl(a), (x, y, u) d) = d [a/x] : E [a/y, refl(a)/u]} \quad (Id\text{-computation})$$

```

record Id (A: Type): Type :=
  (Id: A  $\rightarrow$  A  $\rightarrow$  Type)
  (refl (a: A): Id a a)
  (Predicate :=  $\forall$  (x, y: A)  $\rightarrow$  Id x y  $\rightarrow$  Type)
  (Forall (C: Predicate) :=  $\forall$  (x, y: A)  $\rightarrow$   $\forall$  (p: Id x y)  $\rightarrow$  C x y p)
  ( $\Delta$  (C: Predicate) :=  $\forall$  (x: A)  $\rightarrow$  C x x (refl x))
  (axiom-J (C: Predicate):  $\Delta$  C  $\rightarrow$  Forall C)
  (computation-rule (C: Predicate) (t:  $\Delta$  C):
     $\forall$  (x: A)  $\rightarrow$  (J C t x x (refl x)) ==> (t x))

```

2.5 Inductive Types

$$\frac{A : Type \quad x : A \quad B(x) : Type}{W(x : A) \rightarrow B(x) : Type} \quad (W\text{-formation})$$

$$\frac{a : A \quad t : B(a) \rightarrow W}{sup(a, t) : W} \quad (W\text{-intro})$$

$$\frac{w : W \vdash C(w) : Type \quad x : A, u : B(x) \rightarrow W, \quad v : \Pi(y : B(x)) \rightarrow C(u(y)) \vdash c(x, u, v) : C(sup(x, u))}{w : W \vdash wrec(w, c) : C(w)} \quad (W\text{-elimination})$$

$$\frac{w : W \vdash C(w) : Type \quad x : A, u : B(x) \rightarrow W, \quad v : \Pi(y : B(x)) \rightarrow C(u(y)) \vdash c(x, u, v) : C(sup(x, u))}{x : A, \quad u : B(x) \rightarrow W \vdash wrec(sup(x, u), c) = c(x, u, \lambda(y : B(x)) \rightarrow wrec(u(y), c)) : C(sup(x, u))} \quad (W\text{-computation})$$

3 Category Theory

3.1 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: morphisms (arrows) and objects (domains and codomains of arrows) along with associative operation of composition and unit morphism that exists for all objects in category.

$$\frac{f : A \rightarrow B \quad g : B \rightarrow C}{g \circ f : A \rightarrow C} (Comp) \qquad \frac{}{id_A : A \rightarrow A} (Id)$$

$$\frac{f : B \rightarrow A \quad g : C \rightarrow B \quad h : D \rightarrow C}{(f \circ g) \circ h = f \circ (g \circ h) : D \rightarrow A} (Assoc) \qquad \frac{f : A \rightarrow B}{f \circ id_A = f : A \rightarrow B} (Id_L)$$

$$\frac{f : A \rightarrow B}{id_B \circ f = f : A \rightarrow B} (Id_R)$$

```

record Setoid: Type :=
  (Carrier: Type)
  (Equ: Carrier → Carrier → Prop)
  (Refl: ∀ (e₀: Carrier) → Equ e₀ e₀)
  (Trans: ∀ (e₁,e₂,e₃: Carrier) → Equ e₁ e₂ → Equ e₂ e₃ → Equ e₁ e₃)
  (Sym:   ∀ (e₁ e₂: Carrier) → Equ e₁ e₂ → Equ e₂ e₁)

record Cat: Type :=
  (Ob: Type)
  (Hom:   ∀ (dom,cod: Ob) → Setoid)
  (Id:    ∀ (x: Ob) → Hom x x)
  (Comp:  ∀ (x,y,z: Ob) → Hom x y → Hom y z → Hom x z)
  (Dom₁₀: ∀ (x,y: Ob) (f: Hom x y) → (Hom.Equ x y (Comp x x y id f) f))
  (Cod₁₀: ∀ (x,y: Ob) (f: Hom x y) → (Hom.Equ x y (Comp x y y f id) f))
  (Subst₀: ∀ (x,y,z: Ob)
    (f₁, f₂: Hom x y) (Hom.Equ x y f₁ f₂)
    (g₁, g₂: Hom y z) (Hom.Equ y z g₁ g₂) →
    (Hom.Equ x z (Comp x y z f₁ g₁) (Comp x y z f₂ g₂)) )
  (Assoc₀: ∀ (x,y,z,w: Ob) (f: Hom x y) (g: Hom y z) (h: Hom z w)
    → (Hom.Equ x w (Comp x y w f (Comp y z w g h))
      (Comp x z w (Comp x y z f g) h)))

```

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

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

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.

$$\frac{f : A \rightarrow B \quad g : A \rightarrow C \quad B \times C}{\langle f, g \rangle : A \rightarrow B \times C}$$

$$\begin{aligned}
 \pi_1 \circ \langle f, g \rangle &= f \\
 \pi_2 \circ \langle f, g \rangle &= g \\
 \langle f \circ \pi_1, f \circ \pi_2 \rangle &= f \\
 \langle f, g \rangle \circ h &= \langle f \circ h, g \circ h \rangle \\
 \langle \pi_1, \pi_2 \rangle &= id
 \end{aligned}$$

3.3 Exponential, λ -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 exponential 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.

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

λ -language as CAM machine

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

3.4 Functors, Λ -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.

$$\frac{f : A \rightarrow B}{F f : (A \rightarrow B) \rightarrow (F a \rightarrow F b)}$$

$$\frac{id_A : A \rightarrow A}{F id_A = id_{FA} : F A \rightarrow F A}$$

$$\frac{f : B \rightarrow C, g : A \rightarrow B}{F f \circ F g = F(f \circ g) : F A \rightarrow F C}$$

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

```

record Functor ( $\varphi$ : Type  $\rightarrow$  Type): Type :=
  (fmap:  $\forall$  ( $\alpha, \beta$ : Type)  $\rightarrow$  ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $\varphi$   $\beta \rightarrow \varphi$   $\alpha$ );
  (id $\rightarrow$ :  $\forall$  ( $\alpha$ : Type) ( $x$ :  $\varphi$   $\alpha$ )  $\rightarrow$  fmap id  $x$  =  $x$ );
  (comp $\rightarrow$ :  $\forall$  ( $\alpha, \beta, \gamma$ : Type) ( $f$ :  $\beta \rightarrow \gamma$ ) ( $g$ :  $\alpha \rightarrow \beta$ ) ( $x$ :  $\varphi$   $\alpha$ )
     $\rightarrow$  fmap ( $f \circ g$ )  $x$  = (fmap  $f$   $\circ$  fmap  $g$ )  $x$ )

```

3.5 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, ϕ) , 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.

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

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

3.6 Initial Algebras

A F-algebra $(\mu F, in)$ is the initial F-algebra if for any F-algebra (C, φ) 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, φ) there exists unique arrow $\llbracket \varphi \rrbracket : C \rightarrow \nu F$ where $f = \llbracket \varphi \rrbracket$

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

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

3.7 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.

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

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

4 Categorical Encoding

4.1 List Sample

Lambek encoding is a categorical proof-representation of higher inductive types encoding. Let's start with simple Church/Boem/Berrarducci encoding for List as one of the basic inductive types:

$$\begin{aligned} \text{Natural Numbers: } & \mu X \rightarrow 1 + X \\ \text{List A: } & \mu X \rightarrow 1 + A \times X \\ \text{Lambda calculus: } & \mu X \rightarrow 1 + X \times X + X \\ \text{Stream: } & \nu X \rightarrow A \times X \\ \text{Potentially Infinite List A: } & \nu X \rightarrow 1 + A \times X \\ \text{Finite Tree: } & \mu X \rightarrow \mu Y \rightarrow 1 + X \times Y = \mu X = \text{List } X \end{aligned}$$

Initial Object

```
data List: (A:*) → * :=
  (nil: List A)
  (cons: A → List A → List A)
```

$$F_A = 1 + A \times X$$

Construct corresponding F-Algebra

```
record listAlg (A: Type) : Type :=
  (X: Type)
  (nil: Unit → X)
  (cons: A → X → X)
```

Introduce List Morphisms

```
record listMor (A: Type) (x1,x2: ListAlg A) : Type :=
  (map: x1.X → x2.X)
  (mapNil: Path x2.X (map (x1.nil unit)) (x2.nil unit))
  (mapCons: ∀ (a: A) → ∀ (x: x1) →
    Path x2.X (map (x1.cons a x)) (x2.cons a (map x)))
```

Introduce connected points of List type

```
record listPoint (A: Type) : Type :=
  (point: ∀ (x: ListAlg A) → x.X)
  (map: ∀ (x1,x2: listAlg A) → ∀ (m: ListMor A x1 x2) →
    Path x2.X (m.map (point x1)) (point x2))
```

Theorems Section

$$\begin{array}{ccc} \lim U & & F \lim U \\ \pi_i \downarrow & \Longrightarrow & F \pi_i \downarrow \\ X_i & & F X_i \end{array} \Longrightarrow \lim \left\{ \begin{array}{c} F \lim U \\ F \pi_i \downarrow \\ F X_i \end{array} \right\}$$

4.2 Basic Ornaments

Our encoding allows you to precise control the type of encoded parameter. There is only three cases and three equations: 1) for unit; 2) particular functorial type over a parameter type and 3) recursive embedding such as in Cons constructor.

q — is a limit in Dialg P category. The constructor body is calculated with q applied to forgetful functor U.

$$\left\{ \begin{array}{l} q_{P,D,G} : \text{End } P (G'(-), G'(-)) \rightarrow P (\text{Lim } G', \text{Lim } G') \\ P : \text{Set}^{op} \times \text{Set} \rightarrow \text{Set} \\ U : \text{Dialg } P \rightarrow \text{Set} \\ G : D \rightarrow \text{Dialg } P \\ G' = UG : D \rightarrow \text{Set} \\ U(\text{Lim } G) = \text{Lim } G' \end{array} \right.$$

4.2.1 Unit

Like for Bool or Nil constructors encoding.

$$\left\{ \begin{array}{l} P_0(A, B) = B \\ q_0 e : \text{Lim } G' \\ q_0 e = e \end{array} \right.$$

4.2.2 Fixed Type

Like for Cons first parameter.

$$\left\{ \begin{array}{l} P_1(A, B) = X \rightarrow P(A, B) \\ q_1 e : X \rightarrow P(\text{Lim } G', \text{Lim } G') \\ q_1 e x A = e A x \end{array} \right.$$

4.2.3 Recursive Parameters

Like for Cons second parameter. This case is a key in encoding recursive data types such as **Lists** and recursive record types such as **Streams**.

$$\left\{ \begin{array}{l} P_2(A, B) = A \rightarrow P(A, B) \\ q_2 e : \text{Lim } G' \rightarrow P(\text{Lim } G', \text{Lim } G') \\ q_2 e I A = e A (I A) \end{array} \right.$$

Data Type, Polymorphic Functions and Theorems The data type of lists over a given set A can be represented as the initial algebra $(\mu L_A, in)$ of the functor $L_A(X) = 1 + (A \times X)$. Denote $\mu L_A = List(A)$. The constructor functions $nil : 1 \rightarrow List(A)$ and $cons : A \times List(A) \rightarrow List(A)$ are defined by $nil = in \circ inl$ and $cons = in \circ inr$, so $in = [nil, cons]$. Given any two functions $c : 1 \rightarrow C$ and $h : A \times C \rightarrow C$, the catamorphism $f = \llbracket [c, h] \rrbracket : List(A) \rightarrow C$ is the unique solution of the equation system:

$$\begin{cases} f \circ nil = c \\ f \circ cons = h \circ (id \times f) \end{cases}$$

where $f = foldr(c, h)$. Having this the initial algebra is presented with functor $\mu(1+A \times X)$ and morphisms $sum [1 \rightarrow List(A), A \times List(A) \rightarrow List(A)]$ as catamorphism. Using this encoding the base library of List will have following form:

$$\begin{cases} foldr = \llbracket [f \circ nil, h] \rrbracket, f \circ cons = h \circ (id \times f) \\ len = \llbracket [zero, \lambda a n \rightarrow succ\ n] \rrbracket \\ (++) = \lambda xs\ ys \rightarrow \llbracket [\lambda(x) \rightarrow ys, cons] \rrbracket(xs) \\ map = \lambda f \rightarrow \llbracket [nil, cons \circ (f \times id)] \rrbracket \end{cases}$$

Lists in Exe language We encode List as usually we do in Π, Σ -provers.

```
data list: (A: *) → * :=
  (nil: list A)
  (cons: A → list A → list A)
```

$$\begin{cases} list = \lambda ctor \rightarrow \lambda cons \rightarrow \lambda nil \rightarrow ctor \\ cons = \lambda x \rightarrow \lambda xs \rightarrow \lambda list \rightarrow \lambda cons \rightarrow \lambda nil \rightarrow cons\ x\ (xs\ list\ cons\ nil) \\ nil = \lambda list \rightarrow \lambda cons \rightarrow \lambda nil \rightarrow nil \end{cases}$$

```
record lists: (A B: *) :=
  (len: list A → integer)
  ((++): list A → list A → list A)
  (map: (A → B) → (list A → list B))
  (filter: (A → bool) → (list A → list A))
```

$$\begin{cases} len = foldr (\lambda x\ n \rightarrow succ\ n)\ 0 \\ (++) = \lambda ys \rightarrow foldr\ cons\ ys \\ map = \lambda f \rightarrow foldr (\lambda x\ xs \rightarrow cons\ (f\ x)\ xs)\ nil \\ filter = \lambda p \rightarrow foldr (\lambda x\ xs \rightarrow if\ p\ x\ then\ cons\ x\ xs\ else\ xs)\ nil \\ foldl = \lambda f\ v\ xs = foldr (\lambda x\ g \rightarrow (\lambda \rightarrow g\ (f\ a\ x)))\ id\ xs\ v \end{cases}$$

4.3 Recursor and Induction

4.3.1 Recursor

There is a belief that recursor (non-dependent eliminator) in Type Theory is a weaker property than induction principle (dependent eliminator). At the same time from category theory we know that Universal Property defines the object uniquely. In the case of initial object in the category of algebras, the initiality could be defined by recursor. That means that all properties of algebra follow from its initiality, as a case it is possible to get the recursor from induction. There is a sensitive moment here, all categorical constructions are being formulated with defined equality on morphisms, in type theory the equality is built-in type that could have extended properties. Simplify we could say that we can get recursor from induction without equality, and with proper equality we could get induction from recursor.

4.3.2 Fibrations

Mechanism of getting induction principle from equality is based on the presentation of dependent types through fibrations. Hereby dependent type ($D : B \rightarrow Type$) is defined as ($p : Sigma B P \rightarrow B$) which projects dependent pair to the first field. In topology such approach is called fibration. To the other direction for a given morphism ($p : E \rightarrow B$) which we understands as fibration with projection p , we could get its dependent type as ($D : B \rightarrow Type$) by calculation in every point ($b : B$) its image of projection p by using equality on elements of B . In type theory besides dependent pair *Sigma* also used the dependent product *Pi*. In encoding of dependent types with fibrations there is a correspondance between elements of dependent and morphism-fibrations for projection p : such ($s : B \rightarrow E$) that $s * p = I$. The example of this implementaion could be seen in

EXE¹ OM²

4.3.3 Induction

The input for induction is a predicate — dependent type encoded with ($p : E \rightarrow B$). Induction needed additional information for predicate. The type of induction is defined by set of inductive constructors. Induction is just a statement that on E we have the structure of F -algebra of inductive type. Now we could apply recursor to E getting the map ($I \rightarrow E$) from initial object which in fact the section (fibre bundle) of fibration and thus defines the dependent function which is a proved value of induction principle. The example for *Bool* could be found in

EXE³ OM⁴

¹<https://github.com/groupoid/exe/blob/master/prelude/macro.new/Mini.macro>

²<https://github.com/groupoid/om/blob/master/priv/posets/sec2all>

³<https://github.com/groupoid/exe/blob/master/prelude/macro.new/Data.Bool.macro>

⁴<https://github.com/groupoid/om/blob/master/priv/posets/Data/Bool/induc>

4.4 Conclusion

Summarizing we encode types of source lambda calculus with objects of selected category, dependent types with fibrations, dependent function as fibrations, inductive types as limits of identity functors on category of F-algebras.

References

Category Theory

- [1] S.MacLane *Categories for the Working Mathematician* 1972
- [2] W.Lawvere *Conceptual Mathematics* 1997
- [3] P.Curien *Category theory: a programming language-oriented introduction* 2008

Pure Type Systems

- [4] P.Martin-Löf *Intuitionistic Type Theory* 1984
- [5] T.Coquand *The Calculus of Constructions.* 1988
- [6] E.Meijer *Henk: a typed intermediate language* 1997
- [7] H.Barendregt *Lambda Calculus With Types* 2010

Inductive Type Systems

- [8] F.Pfenning *Inductively defined types in the Calculus of Constructions* 1989
- [9] P.Wadler *Recursive types for free* 1990
- [10] N.Gambino *Wellfounded Trees and Dependent Polynomial Functors* 1995
- [11] P.Dybjer *Inductive Families* 1997
- [12] B.Jacobs *(Co)Algebras and (Co)Induction* 1997
- [13] V.Vene *Categorical programming with (co)inductive types* 2000
- [14] H.Geuevers *Dependent (Co)Inductive Types are Fibrational Dialgebras* 2015

Homotopy Type Systems

- [15] T.Streicher *A groupoid model refutes uniqueness of identity proofs* 1994
- [16] T.Streicher *The Groupoid Interpretation of Type Theory* 1996
- [17] B.Jacobs *Categorical Logic and Type Theory* 1999
- [18] S.Awodey *Homotopy Type Theory and Univalent Foundations* 2013
- [19] S.Huber *A Cubical Type Theory* 2015
- [20] A.Joyal *What is an elementary higher topos* 2014
- [21] A.Mortberg *Cubical Type Theory: a constructive univalence axiom* 2017