Groupoid Infinity
Kostiantynivska 20/14, Kyiv, Ukraine 04071

Categorical Encoding of Inductive Types

Paul Lyutko, Maxim Sokhatsky

Groupoid Infinity
Kamianets-Podilsky 2016

Contents

1 Abstract

Groupoid Infinity creates a new programming language with dependent types called EXE with small provable dependent core called OM. OM is an implementation of Calculus of Constructions (CoC), the pure lambda calculus with dependent types. It can be compiled (code extraction) to bytecode of Erlang virtual machines BEAM and LING. EXE is an implementation of Calculus of Inductive Constructions (CiC) that lives on top of CoC OM model. You may think of EXE as AST transformation of higher language (with HITs) to OM. This document describes the foundations of categorical encoding of inductive types used in EXE/OM stack.

1.1 Why new Dependent Language?

No Fixpoint and Induction in Core. We came up with pure CoC core having predicative and impredicative universe hierarchies and macro extensions. Other MLTT cores have additional axioms like Fixpoint and Induction (and even more) – something we strive to escape because it leads to the complex core. No, we don't have Fixpoint, and yes, we implemented Induction principle in pure CoC.

Extensible Language Design. Encoding of inductive types is based on categorical semantic of compilation to CoC. All other syntax constructions are inductive definitions, plugged into the stream parser. AST of the CoC 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) is just macrosystem over CoC language, its syntax extensions.

Changeable 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 the encoding which dedicated to inductive types. Another well known are Scott (laziness), Parigot (laziness and constant-time iterators) and CPS (continuations) encodings.

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 honor 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 [5].

The most modern type theory, the Theory of Homotopy Types (HoTT) [**?**] does not yet has a computational interpretation, finding such interpretation is an open problem and an area of active research [6]. So we chose one of the variants of the Type Theory of Per Martin Lof [3] to implement EXE.

We will follow the tradition in the literature on type systems [1] 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 definitionaly 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 [1]), 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. 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

$$\frac{i : Nat}{Type_i} \qquad \text{(sorts)}$$

$$\frac{i : Nat}{Type_i : Type_{i+1}} \qquad \text{(axioms)}$$

$$\frac{i : Nat, \quad j : Nat}{Type_i \to Type_j : Type_{max(i,j)}} \qquad \text{(rules)}$$

2.3 Dependent Types

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

$$\frac{x : A \vdash B : Type}{\Pi\ (x : A) \to B : Type} \qquad \text{(Π-formation)}$$

$$\frac{x : A \vdash b : B}{\lambda\ (x : A) \to b : \Pi\ (x : A) \to B} \qquad \text{(λ-intro)}$$

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

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

```
record Π (A: Type): Type :=
       (Π: (A → Type) → Type)
       (fun: (B: A → Type) → ∀ (a: A) → B a → Π B)
       (app: (B: A → Type) → Π B → ∀ (a: A) → B a)
       (app-fun (B: A → Type) (f: ∀ (a: A) → B a): ∀ (a: A) → app (fun f) a ==> f a)
       (fun-app (B: A → Type) (p: Π B): fun (λ(a: A) → app p a) ==> p)
```

2.4 Identity Types

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

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

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

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

```
record Id (A: Type): Type :=
       (Id: A → A → Type)
       (refl (a: A): Id a a)
       (Predicate := ∀ (x,y: A) → Id x y → Type)
       (Forall (C: Predicate) := ∀ (x,y: A) → ∀ (p: Id x y) → C x y p)
       (Δ (C: Predicate) := ∀ (x: A) → C x x (refl x))
       (axiom-J (C: Predicate): Δ C → Forall C)
       (computation-rule (C: Predicate) (t: Δ C):
                           ∀ (x: A) → (J C t x x (refl x)) ==> (t x))
```

2.5 Inductive Types

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

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

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

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

3 Category Theory

3.1 Programs and Functions

Category theory is widely used as an instrument for mathematicians for software analisys. 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.

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

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

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

```
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)
       (Dom₁∘:  ∀ (x,y: Ob) (f: Hom x y) → (Hom.Equ x y (Comp x x y id f) f))
       (Cod₁∘:  ∀ (x,y: Ob) (f: Hom x y) → (Hom.Equ x y (Comp x y y f id) f))
       (Subst∘: ∀ (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)) )
       (Assoc∘: ∀ (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 $< f, g >$ 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 $\top$ coterminal) 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 $\pi \circ \sigma = \sigma \circ \pi = id$.

$$\frac{x : A \times B}{\pi_1 \; : A \times B \to A; \; \pi_2 \; : A \times B \to B}$$

$$\frac{}{\perp}$$

$$\frac{a : A \qquad b : B}{a \mid b : A \otimes B}$$

$$\frac{a : A \qquad b : B}{(a, b) : A \times B}$$

$$\frac{x : A \otimes B}{ß_1 : A \to A \otimes B; \; ß_2 : B \to A \otimes B}$$

$$\frac{}{\top}$$

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 [15]. 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 $[f, g] : B + C \to A$ morphism sums.

$$\frac{f : A \to B \qquad g : A \to C \qquad B \times C}{\langle f, g \rangle : A \to B \times C}$$

$$\pi_1 \circ \langle f, g \rangle = f$$
$$\pi_2 \circ \langle f, g \rangle = g$$
$$\langle f \circ \pi_1, f \circ \pi_2 \rangle = f$$
$$\langle f, g \rangle \circ h = \langle f \circ h, g \circ h \rangle$$
$$\langle \pi_1, \pi_2 \rangle = id$$

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 vew we need to define categorica exponential $f : A^B$, which are analogue to functions $f : A \to B$. 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.

$$\frac{x : A \vdash M : B}{\lambda\ x\ .\ M : A \to B}$$

$$\frac{f : A \to B \qquad a : A}{apply\ f\ a\ \ : (A \to B) \times A \to B}$$

$$\frac{f : A \times B \to C}{curry\ f : A \to (B \to C)}$$

$$\begin{gathered}
apply \circ \langle (curry\ f) \circ \pi_1, \pi_2 \rangle = f \\
curry\ apply \circ \langle g \circ \pi_1, \pi_2 \rangle) = g \\
apply \circ \langle curry\ f, g \rangle = f \circ \langle id, g \rangle \\
(curry\ f) \circ g = curry\ (f \circ \langle g \circ \pi_1, \pi_2 \rangle) \\
curry\ apply = id
\end{gathered}$$

λ-language as CAM machine

$$\begin{aligned}
\text{Objects} &: \top \mid\ \to\ \mid \times \\
\text{Morphisms} &: id \mid f \circ g \mid \langle f, g \rangle \mid apply \mid \lambda \mid curry
\end{aligned}$$

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.

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

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

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

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

```
record Functor (φ: Type → Type): Type :=
       (fmap: ∀ (α,β: Type) → (α → β) → φ β → φ β;
       (id→: ∀ (α: Type) (x: φ α) → fmap id x = x)
       (comp→: ∀ (α,β,γ: Type) (f: β → γ) (g: α → β) (x: φ α)
               → fmap (f ∘ g) x = (fmap f ∘ fmap g) x)
```

3.5 Algebras

F-Algebras gives us a categorical understanding recursive types. Let $F : C \to C$ be an endofunctor on category C. An F-algebra is a pair (C, ϕ), where C is an object and $\phi\ : F\ C \to C$ 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.

$$\begin{tikzcd} F\ C \arrow[r, "\varphi"] \arrow[d, "F\ f"'] & C \arrow[d, "f"] \\ F\ D \arrow[r, "\psi"] & D \end{tikzcd} \qquad \begin{tikzcd} C \arrow[r, "\varphi"] \arrow[d, "f"'] & F\ C \arrow[d, "F\ f"] \\ D \arrow[r, "\psi"] & F\ D \end{tikzcd}$$

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

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 $(\!|\varphi|\!) : \mu F \to C$ where $f = (\!|\varphi|\!)$ 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 $[\![\varphi]\!] : C \to \nu F$ where $f = [\![\varphi]\!]$

$$\begin{tikzcd} F\ \mu F \arrow[r, "in"] \arrow[d, "F\ (\!|\varphi|\!)"'] & \mu F \arrow[d, "(\!|\varphi|\!)"] \\ FC \arrow[r, "\varphi"] & C \end{tikzcd} \qquad \begin{tikzcd} C \arrow[r, "\phi"] \arrow[d, "{[\![\varphi]\!]}"'] & F\ C \arrow[d, "F\ {[\![\varphi]\!]}"] \\ \nu F \arrow[r, "out"] & F \nu F \end{tikzcd}$$

$$f \circ in = \varphi \circ F\ f \equiv f = (\!|\varphi|\!) \qquad out \circ f = F\ f \circ \varphi \equiv f = [\![\varphi]\!]$$

3.7 Recursive Types

As was shown by Wadler [10] we could deal with recusrive equations having three axioms: one $fix : (A \to A) \to A$ fixedpoint axiom, and axioms $in : F\ T \to T$ and $out : T \to F\ T$ 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.

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

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

4 Categorical Encoding

4.1 List Sample

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

$$\begin{gathered}\text{Natural Numbers: } \mu\ X \to 1 + X \\ \text{List A: } \mu\ X \to 1 + A \times X \\ \text{Lambda calculus: } \mu\ X \to 1 + X \times X + X \\ \text{Stream: } \nu\ X \to A \times X \\ \text{Potentialy Infinite List A: } \nu\ X \to 1 + A \times X \\ \text{Finite Tree: } \mu\ X \to \mu\ Y \to 1 + X \times Y = \mu\ X = \mathit{List}\ X\end{gathered}$$

Initial Object

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

F_A = 1 + A × X

Construct corresponding F-Algebra

```
record ListAlg: (A:*) → * :=
       (X: *)
       (nil: X)
       (cons: A → X → X)
```

Introduce List Morphisms

```
infix '=' := Setoid.Ob.Equ

record ListMor: (A: *) → (x1 x2: ListAlg A) → * :=
       (map: x1.X → x2.X)
       (okNil: map x1.nil = x2.nil)
       (okCons: ∀ (a: A) → ∀ (x: x1)
                 → map x1.cons a x = x2.cons a (map x))
```

Introduce connected points of List type

```
record ListPoint: (A: *) → * :=
       (point: ∀ (x: ListAlg A) -> x.X)
       (pointOk:  ∀ (x1 x2: ListAlg A)
               → ∀ (m: ListMor A x1 x2)
               → Setoid.Ob.Equ (map m point x1) (point x2))
```

Theorems Section

$$\begin{tikzcd} \mathit{lim}\ U \arrow[d, "\pi_i"'] & \Longrightarrow & F\ \mathit{lim}\ U \arrow[d, "F\ \pi_i"'] & \Longrightarrow \mathit{lim} \left\{ \begin{array}{c} F\ \mathit{lim}\ U \\ \downarrow F\ \pi_i \\ F\ X_i \end{array} \right\} \\ X_i & & F\ X_i & \end{tikzcd}$$

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.

$$\begin{cases} q_{P,D,G} : End\ P\ (G'(-), G'(-)) \to P\ (Lim\ G', Lim\ G') \\ P : Set^{op} \times Set \to Set \\ U : Dialg\ P \to Set \\ G : D \to Dialg\ P \\ G' = UG : D \to Set \\ U(Lim\ G) = Lim\ G' \end{cases}$$

4.2.1 Unit

Like for Bool or Nil constructors encoding.

$$\begin{cases} P_0(A, B) = B \\ q_0\ e : Lim\ G' \\ q_0\ e = e \end{cases}$$

4.2.2 Fixed Type

Like for Cons first parameter.

$$\begin{cases} P_1(A, B) = X \to P(A, B) \\ q_1\ e : X \to P(Lim\ G', Lim\ G') \\ q_1\ e\ x\ A = e\ A\ x \end{cases}$$

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**.

$$\begin{cases} P_2(A, B) = A \to P(A, B) \\ q_2\ e : Lim\ G' \to P(Lim\ G', Lim\ G') \\ q_2\ e\ I\ A = e\ A\ (I\ A) \end{cases}$$

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 $\mu L_A = \mathit{List}(A)$. The constructor functions $nil : 1 \to \mathit{List}(A)$ and $cons : A \times \mathit{List}(A) \to \mathit{List}(A)$ are defined by $nil = in \circ inl$ and $cons = in \circ inr$, so $in = [nil, cons]$. Given any two functions $c : 1 \to C$ and $h : A \times C \to C$, the catamorphism $f = (\![c, h]\!) : \mathit{List}(A) \to C$ 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 $[1 \to \mathit{List}(A), A \times \mathit{List}(A) \to \mathit{List}(A)]$ as catamorphism. Using this encoding the base library of List will have following form:

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

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)
```

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

```
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))
```

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

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 ($D : B \to Type$) is defined as ($p : Sigma\ B\ P \to B$) which projects dependent pair to the first field. In topology such approach is called fibration. To the other direction for a given morphism ($p : E \to B$) which we understands as fibration with projection p, we could get its dependent type as ($D : B \to Type$) by calculation in every point ($b : B$) its image of projection p by using equality on elements of B. In type theory besides depndent 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 ($s : B \to E$) that $s * p = I$. The example of this implementaion could be seen in

EXE[1] OM[2]

4.3.3 Induction

The input for induction is a predicate — dependent type encoded with ($p : E \to B$). 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 ($I \to E$) 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

EXE [3] OM [4]

[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

[1] Henk Barendregt *Lambda Calculus With Types* 2010

[2] Erik Meijer, Symon Peyton Jones *Henk: a typed intermediate language* 1984

[3] Per Martin-Löf *Intuitionistic Type Theory* 1984

[4] Pierre-Louis Curien *Category theory: a programming language-oriented introduction*

[5] Nordstrom, Bengt and Petersson, Kent and Smith, Jan M. *Programming in Martin-Löf's Type Theory: An Introduction*

[6] Kristina Sojakova *Higher Inductive Types as Homotopy-Initial Algebras* 2015

[7] Alvaro Pelayo, Michael A. Warren *Homotopy type theory* 2014

[8] Varmo Vene *Categorical programming with (co)inductive types* 2000

[9] Frank Pfenning *Inductively defined types in the Calculus of Constructions* 1989

[10] P.Wadler *Recursive types for free* 2014

[11] P.Dybjer *Inductive Famalies* 1997

[12] N.Gambino,M.Hyland *Wellfounded Trees and Dependent Polynomial Functors* 1995

[13] T.Coquand, G.Huet *The Calculus of Constructions.* 1988

[14] Bart Jacobs *Categorical Logic and Type Theory* 1999

[15] NA.Danielsson,J.Hughes,J.Gibbons *Fast and Loose Reasoning is Morally Correct* 2006

[16] Lambek, J. and Scott, P. *Introduction to Higher Order Categorial Logic* 1986