***Pavlo Maslyanko*[1], *Maxim Sokhatsky*[2], *Pavlo Liutko*[3]**
[1]*Faculty of Applied Mathematics at NTUU "KPI", Kyiv, Ukraine;* [2]*Synrc Research Center, s.r.o, Praha, Czech Republic;* [3]*Groupoid Infinity, Inc.*

(Co-)Inductive Dependent Types Normal Form Representation in Pure Type System, the Calculus of Constructions

While modern ITT provers expose core axioms of inductive constructions at checker-level, we promote pure type system setting for inductive constructions, and we show the technique of Berrarducci-based encoding with an example of Setoid base object along with its theorems. By preserving cathegorical semantics we unveil the universal method of modeling the inductive construction as initial object in fibrational F,G-dialgebras (or F-algebra for non-dependent case) with an example of compilation to Erlang untyped lambda calculus.

Lambda Assembler. Intermediate language resemble both the Henk [**?**] theory of PTS CoC language and can be tooled with its Morte [**?**] implementation:

```
E := * | □ | ( E ) | E E
           | λ ( L : E ) → E
           | ∀ ( L : E ) → E
```

Exe Language. We extend the core PTS language with inductive **data** and coinductive **record** definitions. This constructions is used to model any type in any universe. Top level language supports only (co-)inductive definitions that compiles directly to CoC lambda assembler.

```
I := #identifier
O := ∅ | ( O ) |
     □ | ∀ ( I : O ) → O |
     * | λ ( I : O ) → O |
     I | O → O | O O
L := ∅ | L I
A := ∅ | A ( L : O ) | A O
F := ∅ | F ( I : O ) | ()
P := I O , P | I O
E := ∅ | E data I : A := F
       | E record I : A [ extend P ] := F
```

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.

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

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

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 $f = [\![\varphi]\!]$

$$\begin{array}{ccc} F\ \mu F & \xrightarrow{in} & \mu F \\ {\scriptstyle F\ (\!|\varphi|\!)}\downarrow & & \downarrow{\scriptstyle (\!|\varphi|\!)} \\ FC & \xrightarrow{\varphi} & C \end{array} \qquad \begin{array}{ccc} C & \xrightarrow{\phi} & F\ C \\ {\scriptstyle [\![\varphi]\!]}\downarrow & & \downarrow{\scriptstyle F\ [\![\varphi]\!]} \\ \nu F & \xrightarrow{out} & F\nu F \end{array}$$

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

Inductive Types. There is two types of recursion: one is least fixed point (as $F_A\ X = 1 + A \times X$ or $F_A\ X = A + X \times X$), in other words the recursion with a base (terminated with a bounded value), lists are trees are examples of such recursive structures (so we call induction recursive sums); and the second is greatest fixed point or recursion withour base (as $F_A\ X = A \times X$) — such kind of recursion on infinite lists (codata, streams, coinductive types) we can call recursive products.

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

Lists. 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 $nil : 1 \to List(A)$ and $cons : A \times List(A) \to List(A)$ 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 $f = (\!|[c, h]|\!) : List(A) \to C$ 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 $[1 \to List(A), A \times List(A) \to List(A)]$ as catamorphism. Using this encdoding 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))
```

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

Exe Prelude. We have coinductive Setoid as a base onbject to reason about.

```
record Ob: * :=
       (elem: *)
       (eq: Equ elem)
       (point: elem)

record Hom (X Y: Ob): * :=
       (elem: X.elem → Y.elem)
       (eq: ∀ (x1, x2: X.elem) →
            X.eq x1 x2 → Y.eq (elem x1) (elem x2))
       (point: Y.eq (elem X.point) Y.point)

  data True: Prop :=
       (intro: () → true)

  data False: Prop := ()

  data Eq: (A:*) → A → A → Prop :=
       (refl: (x:A) → Eq A x x)

record Setoid: * :=
       (ob: *)
       (hom: ∀ (X Y: ob) → Hom X Y)
```

You can model Monad control construction by using records.

```
record pure: (P: * → *) → (A: *) → * :=
       (return: P A)

record functor: (F: * → *) → (A, B: *) → * :=
       (fmap: (A → B) → F A → F B)

record applicative: (F: * → *) → (A, B: *) → *
extend pure F A, functor F A B :=
       (ap: F (A → B) → F A → F B)

record monad: (F: * → *) → (A, B: *) → *
extend pure F A, functor F A B :=
       (join: F (F A) → F B)
```

Lambda Assembler. One List/Type constructor and two List constructors Cons and Nil.

```
  λ (a: *)
→ ∀ (List: *)
→ ∀ (Cons:
    ∀ (head: a)
  → ∀ (tail: List)
  → List)
→ ∀ (Nil: List)
→ List

  λ (a: *)
→ λ (head: a)
→ λ (tail:
    ∀ (List: *)
  → ∀ (Cons:
      ∀ (head: a)
    → ∀ (tail: List)
    → List)
  → ∀ (Nil: List)
  → List)
→ λ (List: *)
→ λ (Cons:
    ∀ (head: a)
  → ∀ (tail: List)
  → List)
→ λ (Nil: List)
→ Cons head (tail List Cons Nil)

  λ (a: *)
→ λ (List: *)
→ λ (Cons:
    ∀ (head: a)
  → ∀ (tail: List)
    → List)
→ λ (Nil: List)
→ Nil
```

Erlang. Result Language.

```
ap(Fun,Args) -> lists:foldl(fun(X,Acc) -> Acc(X) end,Fun,Args).

list () -> fun (List) -> fun (Cons) -> fun (Nil) -> List end end end.
nil  () -> fun (Cons) -> fun (Nil) -> Nil end end.
cons () -> fun (A) -> fun (List) -> fun (Cons) -> fun (Nil) ->
           ap(Cons,[A,ap(List,[Cons,Nil])]) end end end end.
```