***Maxim Sokhatsky*[1], *Pavlo Lyutko*[2], *Pavlo Maslyanko*[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.*

We adress the problem of missing a programming language with dependent types for Erlang virtual machines. Presence of dependent types in programming languages turns them to theorem provers and helps builing certified software for mission critical applications. While other platforms has dependently types languages: Scala (JVM), F* (CLR), Agda and Idris (GHC); Erlang virtual machines (LING and BEAM) lacks support of such languages. We creates a PL based on Dependent Type Theory and Categorical Semantics of Lambda Encodings (called EXE), with small core of Pure Type System (called OM) for Erlang virtual machines. OM is an implementation of Calculus of Constructions (CoC), the pure lambda calculus with dependent types enriched with infinite universes. EXE is an implementation of Calculus of Inductive Constructions (CiC) that compiles to OM. The core problem of building a dependently typed language is a semantics of inductive types encoding. We propose a system of pluggable encoding backends which are based on verified categorical models. Switcheable encoding will allow us to measure the performance of different encodings on real world applications. OM and EXE code can be compiled (by code extraction and type erasure) to bytecode of Erlang virtual machines BEAM and LING.

Motivation. 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.

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

$$\mathrm{E} := *_i \mid (\ \mathrm{E}\ ) \mid \mathrm{E}\ \mathrm{E} \mid \lambda\ (\ \mathrm{L}\ :\ \mathrm{E}\ )\ \to\ \mathrm{E} \mid \forall\ (\ \mathrm{L}\ :\ \mathrm{E}\ )\ \to\ \mathrm{E}$$

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

```
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.

Natural Numbers: $\mu\ X \to 1 + X$
List A: $\mu\ X \to 1 + A \times X$
Lambda calculus: $\mu\ X \to 1 + X \times X + X$
Stream: $\nu\ X \to A \times X$
Potentialy Infinite List A: $\nu\ X \to 1 + A \times X$
Finite Tree: $\mu\ X \to \mu\ Y \to 1 + X \times Y = \mu\ X = List\ X$

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 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: Type): Type :=
     (nil: list A)
     (cons: A → list A → list A)
```

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

Intermediate Lambda Assembler OM. We present here (in this document) only Church-Boem-Berrarducci simplified encoding.

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

Cons. : 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)

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

Erlang. Result language after type erasure.

```
list () -> fun (List) -> fun (Cons) -> fun (Nil) -> List end end end.
nil  () -> fun (Cons) -> fun (Nil) -> Nil end end.
cons () -> fun (Head) -> fun (Tail) -> fun (Cons) -> fun (Nil) ->
             ((Cons(Head))((Tail(Cons))(Nil))) end end end end.
```

References. **1.** P.Lyutko,M.Sokhatsky *Categorical Encoding of Inductive Types* 2016
2. P.Lyutko,M.Sokhatsky *Identity Type Encoding* 2016
3. M.Sokhatsky,P.Maslyanko *System Engineering and Verification of Execution Environment* 2016