synrc research center s.r.o.
Roháčova 141/18, Praha 3 13000, Czech Republic

Normal Form Representation of Dependent Inductive Types in terms of Pure Type System

Technical Article

Maxim Sokhatsky, Synrc Research Center

Kyiv 2016

Contents

1 Introduction

1.1 Verifiable Functional Compiler

This article describes the frontend language **Exe** along with its categorical semantic that compiles to a pure type system minimal core **Om** — the non-recursive subset of dependent type theory from which you can extract: LLVM code, untyped lambda code with erased type information, or even *System* F_ω programs. As a top-level task we expect to produce lean verifiable functional compiler pipeline, but as an milestone task we just produce concise working prototype in Erlang language.

1.2 General Purpose Functional Language Exe

General purpose function language with functors, lambdas on types, recursive algebraic types, higher order functions and embedded process calculus with corecursion, free monad for effects encoding. This language will be called Exe and dedicated to be high level general purpose functional programming language frontend to small core of dependent type system without recursion called Om. This language indended to be useful enough to encode KVS, N2O and BPE applications.

1.3 Intermediate Language Om

An intermediate Om language is based on Henk [**?**] languages described first by Erik Meyer and Simon Peyton Jones in 1997. Leter on in 2015 Morte impementation of Henk design appeared in Haskell, using Boem-Berrarducci encoding of non-recursive lamda terms. It is based only on π, λ and *apply* constructions, one axiom and four deduction rules. The design of Om language resemble Henk and Morte both design and implementation. This language indended to be small, conside, easy provable and clean and produce verifiable peace of code that can be distributed over the networks and compiled at target with safe linkage.

1.4 Target Erlang VM and LLVM platforms

This works expect to compile to limited target platforms. For now Erlang, Haskell and LLVM is awaiting. Erlang version is expected to be useful both on LING and BEAM Erlang virtual machines.

2 General Purpose Language

2.1 Category Theory, 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 two lists: the one is morphisms (arrows) and the second is objects (domains and codomains of arrows) along with assoociative operation of composition and unit morphism that exists for all objects in category.

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

Intro axioms – associativity of composition and left/right unit arrow compisitions 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:

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash g : B \to C}{\Gamma \vdash g \circ f : A \to C}$$

$$\frac{}{\Gamma \vdash id_A : A \to A}$$

$$\frac{\Gamma \vdash f : B \to A \qquad \Gamma \vdash g : C \to B \qquad \Gamma \vdash h : D \to C}{\Gamma \vdash (f \circ g) \circ h = f \circ (g \circ h) : D \to A}$$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash f \circ id_A = f : A \to B}$$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash id_B \circ f = f : A \to B}$$

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

2.2 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 $\bot$ 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 $\pi \circ \sigma = \sigma \circ \pi = id$.

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

$$\frac{}{\Gamma \vdash\ \bot}$$

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

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

$$\frac{}{\Gamma \vdash\ \top}$$

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

The $\bot$ 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 $\bot$ 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.

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

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

2.3 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.

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

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

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

$$apply \circ \langle (curry\ f) \circ \pi_1, \pi_2 \rangle = f$$
$$curry\ apply \circ \langle g \circ \pi_1, \pi_2 \rangle) = g$$
$$apply \circ \langle curry\ f, g \rangle = f \circ \langle id, g \rangle$$
$$(curry\ f) \circ g = curry\ (f \circ \langle g \circ \pi_1, \pi_2 \rangle)$$
$$curry\ apply = id$$

2.4 λ-language

$$\text{Objects} : \bot \mid\ \to\ \mid \times$$
$$\text{Morphisms} : id \mid f \circ g \mid \langle f, g \rangle \mid apply \mid \lambda \mid curry$$

2.5 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.

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

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

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

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

2.6 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.

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

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

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

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

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

2.8 Recursive Types

As was shown by Wadler [**?**] 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.

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

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

2.9 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: $\mu\ X \to \mu\ Y \to 1 + X \times Y = \mu\ X = List\ X$

As we know there are several ways to appear for variable in recursive algebraic type. Least fixpoint are known as an recursive expressions that have a base of recursion Both recursive and corecursive datatypes could be encoded using Boem-Berarducci encoding as an non-recursive definitions of folds that include in indentity signature all the constructor components of (co)inductive type.

2.10 Peano Numbers

Pointer Unary System is a category C with terminal object and a carrier X having morphism $[zero : 1_C \to X, succ : X \to X]$. The initial object of C is called Natural Number Objects and models Peano axiom set.

2.11 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:

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

2.12 Inductive Encoding and List Module

```
inductive 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))
```

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

2.13 Intuitionistic Type Theory

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

As was show by Spephan Kaes [**?**], one of strategy of type clasess engine implementataion is using polymorphic structures which allows us do deal with persistent structures on a low theoretical level. Moreover this style of coding is completly compatible with Erlang records which are used to model KVS and N2O hierarchies.

2.14 Logic and Quantification

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

$$\frac{\Gamma\ x : A \vdash B \qquad \Gamma\ \vdash A}{\Gamma\ \vdash \Pi(x : A)B} \qquad \frac{\Gamma\ \vdash a : A \qquad \Gamma\ x : A \vdash B \qquad \Gamma\ b : B(x = a)}{\Gamma\ \vdash (a, b) : \Pi(x : A)B}$$

$$\frac{\Gamma\ x : A \vdash B \qquad \Gamma\ \vdash A}{\Gamma\ \vdash \Sigma(x : A)B} \qquad \frac{\Gamma\ \vdash a : A \qquad \Gamma\ x : A \vdash B \qquad \Gamma\ b : B(x = a)}{\Gamma\ \vdash (a, b) : \Sigma(x : A)B}$$

$$\frac{\Gamma\ \vdash x : A \qquad \Gamma\ \vdash x' : A}{\Gamma\ \vdash Id_A(x, x')}$$

reflexivity	:	$Id_A(a, a)$
substitution	:	$Id_A(a, a') \to B(x = a) \to B(x = a')$
symmetry	:	$Id_A(a, b) \to Id_A(b, a)$
transitivity	:	$Id_A(a, b) \to Id_A(b, c) \to Id_A(a, c)$
congruence	:	$(f : A \to B) \to Id_A(x, x') \to Id_B(f(x), f(x'))$

3 Intermediate Language

3.1 Om Language Definition

Om resemble both the Henk theory of pure type system and λC calculus of constructions and Morte Core Specification

```
EXPR :=                       EXPR              EXPR
      | "\"   "(" LABEL ":" EXPR ")" "arrow" EXPR
      | "\/"  "(" LABEL ":" EXPR ")" "arrow" EXPR
      |                       EXPR     "arrow" EXPR
      |           LABEL
      | "*"
      | "[]"
      |       "("            EXPR ")"
```

During forward pass we stack applications (except typevars), then on reaching close paren ")" we perform backward pass and stack arrows, until neaarest unstacked open paren "(" appeared (then we just return control to the forward pass).

We need to preserve applies to typevars as they should be processes lately on rewind pass, so we have just typevars bypassing rule. On the rewind pass we stack lambdas by matching arrow/apply signatures where typevar(x) is an introduction of variable "x" to the Gamma context.

$$apply : (A \to B) \times A \to B$$
$$lambda : arrow\ (app\ (typevar\ x)\ ,\ A)\ B$$

3.2 Abstract Syntax Tree

$$\begin{array}{rcl} E & := & K \\ & | & x \\ & | & EE \\ & | & \lambda(x : E) \to E \\ & | & \Pi(x : E) \to E \end{array}$$

3.3 Exe Prelude

```
  enum unit: * :=
       (make: () → unit)

  enum bool: * :=
       (true: () → bool)
       (false: () → bool)

  enum option: (A:*) → * :=
       (none: () → option A)
       (some: A → option A)

  enum product: (A:*) → (B:*) → * :=
       (make: (a:A) → (b:B) → prod A B)

  enum sum: (A:*) → (B:*) → * :=
       (make: (a:A) → (b:B) -> sum A B)

  enum list: (A:*) → * :=
       (nil: () → list A)
       (cons: A → list A → list A)

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))

  enum eq: (A:*) → A → A → * :=
       (refl: (x:A) → eq A x x)

  enum exists: (A:*) → (A → *) → * :=
       (exists-intro: (P: A → *) → (x:A) → P x → exists A P)

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)
```

Om Normal Depended Forms

Lists/Map

```
λ (a: ⋆) → λ (b: ⋆) → λ (f: a → b) → λ (xs: ∀ (List: ⋆) →
∀ (Cons: ∀ (head: a) → ∀ (tail: List) → List) → ∀ (Nil: List)
→ List) → xs (∀ (List: ⋆) → ∀ (Cons: ∀ (head: b) → ∀ (tail: List)
→ List) → ∀ (Nil: List) → List) (λ (head: a) → λ (tail: ∀ (List:
⋆) → ∀ (Cons: ∀ (head: b) → ∀ (tail: List) → List) → ∀ (Nil:
List) → List) → λ (List: ⋆) → λ (Cons: ∀ (head: b) → ∀ (tail:
List) → List) → λ (Nil: List) → Cons (f head) (tail List Cons Nil))
(λ (List: ⋆) → λ (Cons: ∀ (head: b) → ∀ (tail: List) → List)
→ λ (Nil: List) → Nil)
```

Vector

```
λ (n : ∀ (Nat : ⋆) → ∀ (Succ : Nat → Nat) → ∀ (Zero : Nat) →
Nat) → λ (a : ⋆) → ∀ (Vector : (∀ (Nat : ⋆) → ∀ (Succ : Nat →
Nat) → ∀ (Zero : Nat) → Nat) → ⋆ → ⋆) → ∀ (Cons : ∀ (m : ∀ (Nat
: ⋆) → ∀ (Succ : Nat → Nat) → ∀ (Zero : Nat) → Nat) → ∀ (b :
⋆) → b → Vector m b → Vector (λ (Nat : ⋆) → λ (Succ : Nat → Nat)
→ λ (Zero : Nat) → Succ (m Nat Succ Zero)) b) → ∀ (Nil : ∀ (b :
⋆) → Vector (λ (Nat : ⋆) → λ (Succ : Nat → Nat) → λ (Zero : Nat)
→ Zero) b) → Vector n a
```

References

[1] Henk Barendregt *The Lambda Calculus. Its syntax and semantics* 1981

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

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

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

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

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

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

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