Groupoid Infinity
Kyiv, Ukraine

Intermediate Language with Dependent Types and Strong Normalization for Erlang/OTP applications.

Maxim Sokhatsky

2016—2017

Contents

1 Abstract

In this paper presented the pure type system based intermediate language Om and its typechecker with bytecode extraction to Erlang. The typesystem contains infinite number of universes, so it is known to be consistent in dependent type theory. The typechecker also supports switchable predicative and impredicative hierarchies of universes. Supporting Erlang platform natively dictated the look and feel of this work. This system is expected to be usable as trusted core for certified applications which could be runned inside Erlang virtual machines LING and BEAM. The syntax is made compatible with Morte haskell library and supports its base library, however extens the indexed universes. We also show how to program in this environment and link with Erlang inductive and coinductive free structures. The very basic prelude library is shipped as a part of work. The brief notes given on the top-level language which compiles to pure type system core. In the results section we will show that lambda evaluation performance on BEAM virtual machine is acceptable to run the production software. The main audience of this work are Erlang users.

2 Pure Type System as Intermediate Language

The Om language is a dependently typed lambda calculus, an extension of Barendregt' and Coquand Calculus of Constructions with predicative hierarchy of indexed universes. There is no fixpoint axiom needed for the definition of infinity term dependance.

All terms respect ranking *Axioms* inside sequence of universes *Sorts* and complexity of the dependent term is equal maximum complexity of term and its dependency *Rules*. The type system is completely described by the following PTS notation (due to Barendregt):

$$\begin{cases} Sorts = Type.\{i\},\ i : Nat \\ Axioms = Type.\{i\} : Type.\{inc\ i\} \\ Rules = Type.\{i\} \leadsto Type.\{j\} : Type.\{max\ i\ j\} \end{cases}$$

The Om language is based on Henk [6] languages described first by Erik Meijer 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 lambda terms. It is based only on one type constructor Π, its intro λ and *apply* eliminator, infinity number of universes, and β-reduction. The design of Om language resemble Henk and Morte both design and implementation. This language indended to be small, concise, easy provable and able to produce verifiable peace of code that can be distributed over the networks, compiled at target with safe trusted linkage.

2.1 BNF

Om syntax is compatible with λC Coquand's Calculus of Constructions presented in Morte and Henk languages. However it has extension in a part of specifying universe index as a **Nat** number.

```
<> ::= #option
 I ::= #identifier
 U ::= * < #number >
 O ::= U
     | I | ( O ) | O O | O → O
     | λ ( I : O ) → O
     | ∀ ( I : O ) → O
```

Equivalent tree encoding for parsed terms is following:

```
data pts = star (n: nat)
         | var (n: name)
         | app (f a: pts)
         | lambda (x: name) (d c: pts)
         | pi (x: name) (d c: pts)
```

2.2 Universes

As Om has infinite number of universes it should include metateoretical Nat unductive type in its core. Om also doesnt have

$$U_0 : U_1 : U_2 : U_3 : ...$$

U_0 — propositions
U_1 — sets
U_2 — types
U_3 — kinds

$$\frac{}{Nat} \tag{I}$$

$$\frac{o : Nat}{Type_o} \tag{S}$$

2.3 Predicative Universes

All terms obey the A ranking inside the sequence of S universes, and the complexity R of the dependent term is equal to a maximum of the term's complexity and its dependency. Note that predicative universes are incompatible with Church lambda term encoding. You can switch predicative vs impredicative uninverses by typecheker parameter.

$$\frac{i : Nat, j : Nat, i < j}{Type_i : Type_j} \tag{A_1}$$

$$\frac{i : Nat, j : Nat}{Type_i \rightarrow Type_j : Type_{max(i,j)}} \qquad (R_1)$$

2.4 Impredicative Universes

Propositional contractible bottom space is the only available extension to predicative hierarchy that not leads to inconsistency. However there is another option to have infinite impredicative hierarchy.

$$\frac{i : Nat}{Type_i : Type_{i+1}} \qquad (A_2)$$

$$\frac{i : Nat, \quad j : Nat}{Type_i \rightarrow Type_j : Type_j} \qquad (R_2)$$

2.5 Contexts

The contexts models the dictionary with variables for typechecker. It can be typed as simple *List* (*Prod String Term*). The elimination rule is not given here as in implemetation the whole dictionary is destroyed after typechecking.

$$\frac{}{\Gamma : Context} \qquad \text{(S)}$$

$$\frac{\Gamma : Context}{Empty : \Gamma} \qquad \text{(S)}$$

$$\frac{A : Type, \quad x : A, \quad \Gamma : Context}{(x : A) \ \vdash \ \Gamma : Context} \qquad \text{(S)}$$

2.6 Single Axiom Language

This langauge is called one axiom language (or pure) as eliminator and introduction adjoint functors inferred from type formation rule. The only computation rule of Pi type is called beta-reduction.

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

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

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

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

This language could be embedded in itself and used as Logical Framework for the Pi type:

```
record Pi (A: Type) :=
       (intro:  (A → Type) → Type)
       (lambda: (B: A → Type) → pi A B → intro B)
       (app:    (B: A → Type) → intro B → pi A B)
       (applam: (B: A → Type) (f: pi A B) → (a: A) →
                Path (B a) ((app B (lambda B f)) a) (f a))
       (lamapp: (B: A → Type) (p: intro B) →
                Path (intro B) (lambda B (λ (a:A) → app B p a)) p)
```

We extend the $PTS\infty$ with remote AST node wich means remote file loading from trusted storage, anyway this will be checked by typechecker. We deny recursion over remote node. We also add index to var for simplified de Bruin indexes.

```
data om = star             (n: nat)
        | var    (n: name) (n: nat)
        | remote (n: name) (n: nat)
        | app                        (f a: pts)
        | lambda (x: name)           (d c: pts)
        | arrow                      (d c: pts)
        | pi     (x: name)           (d c: pts)
```

2.7 Hierarchy

H returns the target Universe of B term dependendance on A. Thereare two dependance rules known as predicative and impredicative which returns max universe or universe of last term respectively.

```
dep A B impredicative → B
dep A B predicative   → max A B

h Arg Out → dep Arg Out om:hierarchy(impredicative)
```

2.8 Universes

Star returns the number of Universe. If argument is not Universe it returns $\{error, \_\}$.

```
star (:star,N) → N
star _         → (:error, "*")
```

2.9 Functions

Func returns true if the argument is functional space. Otherwise returns $\{error, \_\}$.

```
func ((:forall,),(I,O)) → true
func T                  → (:error,(:forall,T))
```

2.10 Variables

Var returns true if the var N is defined in dictionary B. Otherwise returns $\{error, \_\}$.

```
var N B             → var N B (proplists:is_defined N B)
var N B true        → true
var N B false       → (:error,("free var",N,proplists:get_keys(B)))
```

2.11 Shift

Shift renames var N in N

```
sh (:var,(N,I)),N,P) when I>=P  → (var,(N,I+1))
sh ((:forall,(N,0)),(I,O)),N,P) → ((:forall,(N,0)),sh I N P,sh O N P+1)
sh ((:lambda,(N,0)),(I,O)),N,P) → ((:lambda,(N,0)),sh I N P,sh O N P+1)
sh (Q,(L,R),N,P)                → (Q,sh L N P,sh R N P)
sh (T,N,P)                      → T
```

2.12 Substitution

```
sub Term Name Value                    → sub Term Name Value 0
sub (:arrow,            (I,O)) N V L → (:arrow,        sub I N V L,sub O N V L);
sub ((:forall,(N,0)),(I,O)) N V L → ((:forall,(N,0)),sub I N V L,sub O N(sh V N 0)L+1)
sub ((:forall,(F,X)),(I,O)) N V L → ((:forall,(F,X)),sub I N V L,sub O N(sh V F 0)L)
sub ((:lambda,(N,0)),(I,O)) N V L → ((:lambda,(N,0)),sub I N V L,sub O N(sh V N 0)L+1)
sub ((:lambda,(F,X)),(I,O)) N V L → ((:lambda,(F,X)),sub I N V L,sub O N(sh V F 0)L)
sub (:app,              (F,A)) N V L → (:app,sub F N V L,sub A N V L)
sub (:var,              (N,L)) N V L → V
sub (:var,              (N,I)) N V L when I>L → (:var,(N,I-1))
sub T                          _ _ _ → T.
```

2.13 Type Checker

For sure in pure system we should be careful with **:remote** AST node. Remote AST nodes like **#List/Cons or #List/map** are a remote link to files. So using trick one should desire circular dependency over **:remote**. This is denied in the system. Same notes apply to normalization and definitional equality.

```
type (:star,N)              D → (:star,N+1)
type (:var,(N,I))           D → let true = var N D in keyget N D I
type (:remote,N)            D → cache type N D
type (:arrow,(I,O))         D → (:star,h(star(type I D)),star(type O D))
type ((:forall,(N,0)),(I,O)) D → (:star,h(star(type I D)),star(type O [(N,norm I)|D]))
type ((:lambda,(N,0)),(I,O)) D → let star (type I D)
                                      NI = norm I
                                    in ((:forall,(N,0)),(NI,type(O,[(N,NI)|D])))
type (:app,(F,A))           D → let T = type(F,D)
                                      true = func T
                                      ((:forall,(N,0)),(I,O)) = T
                                      Q = type A D
                                      true = eq I Q
                                    in norm (subst O N A)
```

2.14 Normalization

Normalization perform substitutions on applications to functions searching over all function spaces, performing recursive normalization over the lambda and pi nodes.

```
norm :none                     → :none
norm :any                      → :any
norm (:app,(F,A))              → case norm F of
                                  ((:lambda,(N,0)),(I,O)) → norm (subst O N A)
                                                     NF → (:app,(NF,norm A)) end
norm (:remote,N)               → cache (norm N [])
norm (:arrow,        (I,O))    → ((:forall,("_",0)),(norm I,norm O))
norm ((:forall,(N,0)),(I,O))   → ((:forall,(N,0)),   (norm I,norm O))
norm ((:lambda,(N,0)),(I,O))   → ((:lambda,(N,0)),   (norm I,norm O))
norm T                         → T
```

2.15 Definitional Equality

Definitional Equality simple ensures the erlang term equality.

```
eq ((:forall,("_",0)), X) (:arrow,Y)     → eq X Y
eq (:app,(F1,A1))         (:app,(F2,A2)) → let true = eq F1 F2 in eq A1 A2
eq (:star,N)              (:star,N)      → true
eq (:var,(N,I))           (:var,(N,I))   → true
eq (:remote,N)            (:remote,N)    → true
eq ((:farall,(N1,0)),(I1,O1))
   ((:forall,(N2,0)),(I2,O2)) →
   let true = eq I1 I2
    in eq O1 (subst (shift O2 N1 0) N2 (:var,(N1,0)) 0)
eq ((:lambda,(N1,0)),(I1,O1))
   ((:lambda,(N2,0)),(I2,O2)) →
   let true = eq I1 I2
    in eq O1 (subst (shift O2 N1 0) N2 (:var,(N1,0)) 0)
eq (A,B)                          → (:error,(:eq,A,B))
```

3 Language Usage

In this section we will lift PTS system to MLTT system by defining *Sigma* and *Equ* types to pure *Pi* type. We will use Boehm inductive dependent encoding.

3.1 Sigma Type

Here we will show you some examples of Om Language usage. The PTS system is extremely powerful even without Sigma type. But we can encode **Sigma** type similar how we encode **Prod** tuple pair in Boehm encoding. Let's formulate Sigma type as inductive type:

```
data Sigma (A: Type) (P: A -> Type): Type =
     (intro: (x:A) (y:P x) -> Sigma A P)

-- Sigma/@
   \ (A: *)
-> \ (P: A -> *)
-> \/ (Exists: *)
-> \/ (Intro: \/ (x: A) -> \/ (y: P x) -> Exists)
-> Exists

-- Sigma/Intro
   \ (A: *)
-> \ (P: A -> *)
-> \ (x: A)
-> \ (y: P x)
-> \ (Exists: *)
-> \ (Intro: \/ (z: A) -> P z -> Exists)
-> Intro x y

-- Sigma/fst
   \ (A: *)
-> \ (B: A -> *)
-> \ (S: #Sigma/@ A B)
-> S A ( \(x: A) -> \(_: B x) -> x)

-- Sigma/snd
   \ (A: *)
-> \ (B: A -> *)
-> \ (S: #Sigma/@ A B)
-> S (B (#Sigma/fst A B S))
     (\(_: A) -> \(y: B (#Sigma/fst A B S)) -> y)

-- Sigma/test
#Sigma/Intro #Nat/@ #List/@ #Nat/Zero (#List/Nil #Nat/@)

> om:fst(om:erase(om:norm(om:a("#Sigma/snd #Nat/@ #List/@ #Sigma/test")))).
{{λ,{'Cons',0}},
 {any,{{λ,{'Nil',0}},{any,{var,{'Nil',0}}}}}}
```

For using **Sigma** type for Logic purposes one should change the home Universe of the type to **Prop**. Here it is:

```
data Sigma (A: Prop) (P: A -> Prop): Prop =
     (intro: (x:A) (y:P x) -> Sigma A P)
```

3.2 Equality Type

Another example of expressivness is Equality type a la Martin-Löf.

```
data Equ (A: Type): A -> A -> Type :=
     (refl (a: A): Equ A a a)

-- Equ/@
   \ (A: *)
-> \ (x: A)
-> \ (y: A)
-> \/ (Equ: A -> A -> *)
-> \/ (Refl: \/ (z: A) -> Equ z z)
-> Equ x y

-- Equ/Refl
   \ (A: *)
-> \ (x: A)
-> \ (Equ: A -> A -> *)
-> \ (Refl: \/ (z: A) -> Equ z z)
-> Refl x
```

You cannot construct lambda that will check different values of A type, however you may want to use built-in definitionaly equality and normalization feature of typecheker to actually compare two values:

```
> om:print(
  om:type(
  om:a("(\\ (z: #Equ/@ #Nat/@ #Nat/One #Nat/One) -> #Prop/True)"++
       " (#Equ/Refl #Nat/@ (#Nat/Succ #Nat/Zero))"))).
   \/ (True: *0)
-> \/ (Intro: True)
-> True
ok

> om:print(
  om:type(
  om:a("(\\ (z: #Equ/@ #Nat/@ #Nat/One #Nat/One) -> #Prop/True)"++
       " (#Equ/Refl #Nat/@ #Nat/Zero)"))).
** exception error: no match of right hand side value
   {error,{"==",
          {app,{{var,{'Succ',0}},{var,{'Zero',0}}}},
          {var,{'Zero',0}}}}
```

3.3 Effect Type System

This work expects to compile to limited target platforms. For now Erlang, Haskell and LLVM are awaiting. Erlang version is expected to be useful both on LING and BEAM Erlang virtual machines. This language allows you to define trusted operations in System F and extract this routines to Erlang/OTP platform and plug as trusted resourses. As example we also provide infinite coinductive process creation and inductive shell that linked to Erlang/OTP IO functions directly.

IO protocol We can construct in pure type system the state machine based on (co)-free monads driven by IO protocols. Assume String is equal List of Nat (as it is in Erlang natively), and three external constructors: getLine, putLine and pure. We need to put correspondent implementations on host platform as parameters to perform the actual IO.

```
String: Type = List Nat
data IO: Type =
     (getLine: (String -> IO) -> IO)
     (putLine: String -> IO)
     (pure: () -> IO)
```

3.3.1 Infinity I/O Type

Infinity I/O Type Spec.

```
-- IOI/@
   \ (r : *)
-> \/ (x : *)
-> (\/ (s : *) -> s -> (s -> #IOI/F r s) -> x)
-> x

-- IOI/F
   \ (a : *)
-> \ (State : *)
-> \/ (IOF : *)
-> \/ (PutLine_ : #IOI/data -> State -> IOF)
-> \/ (GetLine_ : (#IOI/data -> State) -> IOF)
-> \/ (Pure_ : a -> IOF)
-> IOF

-- IOI/MkIO
   \ (r : *)
-> \ (s : *)
-> \ (seed : s)
-> \ (step : s -> #IOI/F r s)
-> \ (x : *)
-> \ (k : forall (s : *) -> s -> (s -> #IOI/F r s) -> x)
-> k s seed step

-- IOI/data
#List/@ #Nat/@
```

Infinite I/O Sample Program.

```
-- Morte/corecursive
( \ (r: *1)
 -> ( (((#IOI/MkIO r) (#Maybe/@ #IOI/data)) (#Maybe/Nothing #IOI/data))
    ( \ (m: (#Maybe/@ #IOI/data))
     -> (((((#Maybe/maybe #IOI/data) m) ((#IOI/F r) (#Maybe/@ #IOI/data)))
           ( \ (str: #IOI/data)
            -> ((((#IOI/putLine r) (#Maybe/@ #IOI/data)) str)
                (#Maybe/Nothing #IOI/data))))
          (((#IOI/getLine r) (#Maybe/@ #IOI/data))
           (#Maybe/Just #IOI/data))))))
```

Erlang Coinductive Bindings.

```
copure() ->
    fun (_) -> fun (IO) -> IO end end.

cogetLine() ->
    fun(IO) -> fun(_) ->
        L = ch:list(io:get_line("> ")),
        ch:ap(IO,[L]) end end.

coputLine() ->
    fun (S) -> fun(IO) ->
        X = ch:unlist(S),
        io:put_chars(": "++X),
        case X of "0\n" -> list([]);
                      _ -> corec() end end end.

corec() ->
    ap('Morte':corecursive(),
        [copure(),cogetLine(),coputLine(),copure(),list([])]).

> om_extract:extract("priv/normal/IOI").
ok
> Active: module loaded: {reloaded,'IOI'}

> om:corec().
> 1
: 1
> 0
: 0
#Fun<List.3.113171260>
```

3.3.2 I/O Type

I/O Type Spec.

```
-- IO/@
   \ (a : *)
-> \/ (IO : *)
-> \/ (GetLine_ : (#IO/data -> IO) -> IO)
-> \/ (PutLine_ : #IO/data -> IO -> IO)
-> \/ (Pure_ : a -> IO)
-> IO

-- IO/replicateM
   \ (n: #Nat/@)
-> \ (io: #IO/@ #Unit/@)
-> #Nat/fold n (#IO/@ #Unit/@)
               (#IO/[>>] io)
               (#IO/pure #Unit/@ #Unit/Make)
```

Guarded Recursion I/O Sample Program.

```
-- Morte/recursive
((#IO/replicateM #Nat/Five)
 ((((#IO/[>>=] #IO/data) #Unit/@) #IO/getLine) #IO/putLine))
```

Erlang Inductive Bindings.

```
pure() ->
    fun(IO) -> IO end.

getLine() ->
    fun(IO) -> fun(_) ->
        L = ch:list(io:get_line("> ")),
        ch:ap(IO,[L]) end end.

putLine() ->
    fun (S) -> fun(IO) ->
        io:put_chars(": "++ch:unlist(S)),
        ch:ap(IO,[S]) end end.

rec() ->
    ap('Morte':recursive(),
        [getLine(),putLine(),pure(),list([])]).
```

Here is example of Erlang/OTP shell running recursive example.

```
> om:rec().
> 1
: 1
> 2
: 2
> 3
: 3
> 4
: 4
> 5
: 5
#Fun<List.28.113171260>
```

4 Top Language with Inductive Types

Despite we can encode inductive types in PTS, the best usage of inductive types comes with recursors and fixpoint type that allow recursive typecheck, limiting from the other side the normalization proerties.

So called Calculus of Inductive constructions are used as a top language on top of PTS to reason about inductive types. Here we will show you a sketch of such inductive language model which indendent to be a language extension to PTS system.

Exe is a general purpose functional language with Π and Σ types, recursive algebraic types, higher order functions, corecursion, free monad for effects encoding. It compiles to a small MLTT core of dependent type system with inductive types and equality.

It also has an *Id* type (with its recursor) wich is useful for proving things, *Let* sytax sugar, case analisys over inductive types and also a *pi*-Calculus primitives: *spawn*, *send* and *receive* which are native mappings to Erlang/OTP.

4.1 BNF

```
<> ::= #option
[] ::= #list
 | ::= #sum
 1 ::= #unit
 I ::= #identifier

 U ::= Type < #nat >
 T ::= 1 | ( I : O ) T
 F ::= 1 | I : O = O , F
 B ::= 1 | [ | I [ I ] → O ]
 L ::= 1 | I T
 O ::= I | ( O ) |
       U | O → O
         | O O
         | fun ( I : O ) → O
         | fst O
         | snd O
         | id O O O
         | transport O O O O O
         | ( I : O ) * O
         | ( I : O ) → O
         | data   I T : O := T
         | record I T : O := T
         | let F in O
         | case O B
         | receive B
         | spawn O O
         | send O O
```

That is how we see the language informally.

However in real life we use grammar parser generators. The Erlang version of parser encoded with OTP library **yecc** which implements LALR-1 grammar generator. This version resembles the model and slightly based on cubical type checker by Mortberg et all.

```
mod    -> 'module' id 'where' imp dec       : {module,'$2','$3','$4','$5'}.
imp    -> skip imp                          : '$2'.
imp    -> '$empty'                          : [].
imp    -> 'import' id imp                   : [{import,'$2'}|'$3'].
tele   -> '$empty'                          : [].
tele   -> '(' exp ':' exp ')' tele          : {tele,uncons('$2'),'$4','$6'}.
exp    -> app                               : '$1'.
exp    -> exp arrow exp                     : {arrow,'$1','$3'}.
exp    -> '(' exp ')'                       : '$2'.
exp    -> lam '(' exp ':' exp ')' arrow exp : {lam,uncons('$3'),'$5','$8'}.
exp    -> '(' exp ':' exp ')' arrow exp     : {pi,uncons('$2'),'$4','$7'}.
exp    -> '(' id ':' exp ')' '*' exp        : {sigma,'$2','$4','$7'}.
exp    -> id                                : '$1'.
app    -> exp exp                           : {app,'$1','$2'}.
dec    -> '$empty'                          : [].
dec    -> def skip dec                      : ['$1'|'$3'].
def    -> 'data' id tele '=' sum            : {data,'$2','$3','$5'}.
def    -> id tele ':' exp '=' exp           : {def,'$1','$2','$4','$6'}.
sum    -> '$empty'                          : [].
sum    -> id tele                           : {ctor,'$1','$2'}.
sum    -> id tele '|' sum                   : [{ctor,'$1','$2'}|'$4'].
```

4.2 AST

The model in Cubical and Coq of the Exe language is available at infinity[1] repository of groupoid organization.

```
data tele (A: U)   = emp | tel (n: name) (b: A) (t: tele A)
data branch (A: U) =        br (n: name) (args: list name) (term: A)
data label (A: U)  =       lab (n: name) (t: tele A)
data ind
   = star                         (n: nat)
   | var     (n: name)            (i: nat)
   | app               (f a: ind)
   | lambda  (x: name) (d c: ind)
   | pi      (x: name) (d c: ind)
   | sigma   (n: name) (a b: ind)
   | arrow             (d c: ind)
   | pair              (a b: ind)
   | fst               (p:   ind)
   | snd               (p:   ind)
   | id                (a b: ind)
   | idpair            (a b: ind)
   | idelim            (a b c d e: ind)
   | data_   (n: name) (t: tele ind) (labels:   list (label ind))
   | case    (n: name) (t: ind)      (branches: list (branch ind))
   | ctor    (n: name)               (args:     list ind)
```

[1] https://github.com/groupoid/infinity/tree/master/priv

4.3 Inductive Types

There are 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 and trees are examples of such recursive structures (so we call induction recursive sums); and the second is greatest fixed point or recursion withour a base (as $F_A\ X = A \times X$) — such kind of recursion on infinite lists (codata, streams, coinductive types) we can call recursive products.

4.4 Polynomial Functors

Least fixed point trees are called well-founded trees and encode polynomial functors.

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: $\mu\ X \to \mu\ Y \to 1 + X \times Y = \mu\ X = \mathit{List}\ X$

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.

4.5 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 $\mu L_A = \mathit{List}(A)$. 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 encdoding the base library of List will have following form:

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

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

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

```
-- List/@
   \ (A : *)
-> \/ (List: *)
-> \/ (Cons: \/ (Head: A) -> \/ (Tail: List) -> List)
-> \/ (Nil: List)
-> List

-- List/Cons
   \ (A: *)
-> \ (Head: A)
-> \ (Tail:
      \/ (List: *)
   -> \/ (Cons:
         \/ (Head: A)
      -> \/ (Tail: List)
      -> List)
   -> \/ (Nil: List)
   -> List)
-> \ (List: *)
-> \ (Cons:
      \/ (Head: A)
   -> \/ (Tail: List)
   -> List)
-> \ (Nil: List)
-> Cons Head (Tail List Cons Nil)

-- List/Nil
   \ (A: *)
-> \ (List: *)
-> \ (Cons:
      \/ (Head: A)
   -> \/ (Tail: List)
   -> List)
-> \ (Nil: List)
-> Nil
```

```
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))
```

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

4.6 Normal Forms

Here is example of List/map generic function.

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)
```

4.7 Prelude Base Library

The base library is modeled in cubical type checker.

```
  data Nat: Type :=
       (Zero: Unit → Nat)
       (Succ: Nat → Nat)

  data List (A: Type) : Type :=
       (Nil: Unit → List A)
       (Cons: A → List A → List A)

record list: Type :=
       (len: List A → integer)
       ((++): List A → List A → List A)
       (map: (A,B: Type) (A → B) → (List A → List B))
       (filter: (A → bool) → (List A → List A))

record String: List Nat := List.Nil

  data IO: Type :=
       (getLine: (String → IO) → IO)
       (putLint: String → IO)
       (pure: () → IO)

record IO: Type :=
       (data: String)
       ([>>=]: ...)

record Morte: Type :=
       (recursive: IO.replicateM Nat.Five
                    (IO.[>>=] IO.data Unit IO.getLine IO.putLine))
```

4.8 Compiler Passes

The underlying OM typechecker and compiler is a target language for EXE general purpose language. The overal size of OM language with extractor to Erlang is 263 lines of code.

TOKEN	54 LOC	Handcoded Tokenizer
PARSER	81 LOC	Parser
NORMAL	60 LOC	Term normalization and typechecking
ERASE	36 LOC	Delete information about types
EXTRACT	32 LOC	Extract Erlang Code

We benchmarked the unrolling of inductive list type in Church encoding extracted with OM with native erlang **lists:foldl**.

```
Pack/Unpack 1 000 000 Inductive Nat:   776407 us
Pack/Unpack 1 000 000 ErlangOTP List:  148084 us
Pack/Unpack 1 000 000 Inductive List: 1036461 us
```

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 Famalies* 1997

[12] B.Jacobs *(Co)Algebras) and (Co)Induction* 1997

[13] V.Vene *Categorical programming with (co)inductive types* 2000

[14] H.Geuvers *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