Mathematical Components for Cubical Syntax

Maksym Sokhatskyi [1] and Pavlo Maslianko [1]

[1] National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnical Institute" April 19, 2018

Abstract

The introduction of path spaces and its eliminators in cubical type theory (CTT) as core type checker primitives calls for re-examination the proofs for mathematical components in base libraries of the major provers Coq, Agda, F*, Lean. They provide successful interpretation of calculus of inductive constructions with predicative hierarchies, however the homotopical foundation of CTT demonstrates computational semantics of univalence axiom and different kind of proofs by using path composition and kan filling operations.

We present a base library compatible with Cubical language [2] with respect to both run-time types and its mathematical models. This library is about to extract to run-time languages from cubical syntax. The basic core of the library contains abstractions for Pi, Sigma and Path types, prop, set, and groupoid hierarchy, fixpoint, control structures, recursion schemes, algebraic hierarchy, and category theory.

Despite minimalistic cubical syntax lacks type classes we show the elegant way of encoding type classes in cubical. Also, the minimalistic syntax gives us a lightweight extension to the pure type system (PTS) core and simple and more straightforward extraction to it for a subset of CTT programs.

This article demonstrates an approach of type refinement to create the library where run-time types and its mathematical models can be easily separated while remain fit each other. So we cover here only models needed for modeling run-time theories while touching homotopical and geometrical models left for further articles, as for quotient sets, circle, sphere, h-pushout, truncations, iso, univalence, and other HoTT types.

Keywords: Formal Methods, Type Theory, Programming Languages, Theoretical Computer Science, Applied Mathematics, Cubical Type Theory, Martin-Löf Type Theory

Contents

1 Intro **3**

2 MLTT as Modeling Language **4**
2.1 Pi and Sigma 4
2.2 Identity Type 4
2.3 ∞-Groupoid and h-Types 5
2.4 Operational Semantics 6

3 Runtime Types **7**
3.1 Proto 7
3.2 Empty and Unit 7
3.3 Either and Tuple 7
3.4 Bool 8
3.5 Maybe and Nat 8
3.6 List and Stream 8
3.7 Vector and Fin 9
3.8 IO and Infinity IO 9

4 Control Structures **11**
4.1 Signatures 11
4.2 Run-time types 12
4.3 Accessors 12
4.4 Theorems as Proof-carrying Code 12

5 F-Algebras and Recursion Schemes **14**
5.1 Fixpoint and Free Structures 14
5.2 Catamorphism and Futumorphism 14
5.3 Anamorphism and Histomorphism 15
5.4 Inductive and Coninductive Types 15

6 Algebra **16**
6.1 Monoid and Commutative Monoid 16
6.2 Group and Abelian Group 16
6.3 Ring and Abelian Ring 16

7 Category Theory **17**
7.1 Signature 17
7.2 Accessors 17
7.3 Theorems 17
7.4 Terminal and Initial Objects 18
7.5 Functors 18

8 Conclusion **19**

1 Intro

Research object. The homotopy type theory base libraries in Agda, Cubical, Lean, and Coq. While modern Lean and Agda has the cubical mode, Coq lacks the computational semantics of path primitives while has HoTT library. The real programming language is not enough to develop the software and theorems, the language should be shipped with base library. In this article, we unveil the practical implementation of the base library for cubical type checkers with respect to target run-time environments.

Research subject. We will analyze the base library through the needs of particular features, like basic theorems about MLTT (Pi, Sigma, Equ, HeteroEqu), run-time types and data structures (Empty, Unit, Maybe, Nat, List, Either, Tuple), control structires (Functor, CoFunctor, ContraFunctor, CoContraFunctor, Applicative, CoApplicative, Monad, CoMonad, Inductive, CoInductive), algebra tower (Monoid, CMonoid, Group, AbGroup, Ring, AbRing), and category theory (Category, Functor, Initial, Terminal, Adjoint). We use Martin-Löf Type Theory as a subject and its extension CTT.

Research results. Research result is presented as source code repository that can be used by cubicaltt language and contains the minimal base library used in this article. These primitives form a valuable part of base library, so this article could be considered as an brief introduction to several modules: **pi**, **sigma**, **path**, **prop**, **set**, **control**, **iso**, **equiv**, **cat**, **algebra**, **recursion**. But the library has even more modules, that exceed the scope of this article so you may refer to source code repository[1].

This library is dedicated to cubical-compatible type checkers [2] based on homotopy interval [0,1] and MLTT as a core. The base library is founded on top of 5 core modules: proto (composition, id, const), path (subst, trans, cong, refl, singl, sym), prop, set (isContr, isProp, isSet, isInfinityGroupoid), equiv (fiber, eqiuv) and iso (lemIso, isoPath). This machinery is enough to prove univalence axiom.

(i) The library has rich recursion scheme primitives in recursion module, while very basic nat, list, stream functionality. (ii) The very basic theorems are given in pi, iso_pi, sigma, iso_sigma, retract modules. (iii) The library has category theory theorems from HoTT book in cat, fun and category modules. (iv) The library also includes categorical encoding of dependent types presented in Cwf module.

This library is best to read guided by HoTT book.

[1] http://github.com/groupoid/infinity

Table 1: Types Taxonomy

NR+ND	R+ND	NR+D	R+D
unit	nat	path	vector
bool	list	proto	fin
either		iso	
maybe		equiv	
empty		pi	
NR*ND	**R*ND**	**NR*D**	**R*D**
pure	stream	sigma	cat
functor		setoid	prop
applicative			set
monad			groupoid

2 MLTT as Modeling Language

As MLTT language is used for modeling the base library should include its definition and theorems. The **mltt** module contains the theorems of operational semantics of dependent type theory and MLTT model.

2.1 Pi and Sigma

Pi and Sigma modules provide basic theorems about dependent products and sum. Here is tautology alias definitions for better syntax understanding. In run-time Sigma is being transformed into pair and lambda into functions. All type annotations and term dependence information are being erased.

```
Pi      (A:U) (P:A->U): U = (x:A) -> P x
lambda  (A:U) (B:A->U) (a: A) (b: B a): A -> B a = \ (x: A) -> b
app     (A:U) (B:A->U) (a: A) (f: A -> B a): B a = f a
Sigma   (A:U) (B:A->U): U = (x: A) * B x
pair    (A:U) (B:A->U) (a: A) (b: B a): Sigma A B = (a,b)
pr1     (A:U) (B:A->U) (x: Sigma A B): A = x.1
pr2     (A:U) (B:A->U) (x: Sigma A B): B (pr1 A B x) = x.2
```

2.2 Identity Type

Identity types or Prop types (when using PTS and built-in definitional equality for type checking normalized term forms with means of identity) are both considered to be erased in run-time. For modeling propositional equality later in 1984 was introduced Equ type. [4] However unlike Pi and Sigma the eliminator J of Equ type is not derivable in MLTT [3, 2, 5].

```
Path    (A: U) (a b: A): U
singl   (A: U) (a: A): U
refl    (A: U) (a: A): Path A a a
```

```
sym      (A: U) (a b: A) (p: Path A a b): Path A b a
eta      (A: U) (a: A): singl A a
contr    (A: U) (a b: A) (p: Path A a b): Path (singl A a) (eta A a) (b,p)
cong   (A B: U) (f: A->B) (a b: A) (p: Path A a b): Path B (f a) (f b)
subst    (A: U) (P: A->U) (a b: A) (p: Path A a b) (e: P a): P b
J        (A: U) (a: A) (C: (x: A) -> Path A a x -> U)
      (d: C a (refl A a)) (x: A) (p: Path A a x): C x p
  = subst (singl A a) T (eta A a) (x, p) (contr A a x p) d
          where T (z: singl A a): U = C (z.1) (z.2)
```

You can build setoid [1] definition using sym, refl and cong.

2.3 ∞-Groupoid and h-Types

The reasoning about higher equalties is made through explicit recursion. We encode the level of path dimention through embedded natural numbers in the definition.

```
data N = Z | S (n: N)
```

As foundation we provide recursive and corecursive versions of groupoid definitions. h-Types [5] (Prop, Set, Groupoid, etc) are defined through these primitives.

```
n_grpd (A: U) (n: N): U = (a b: A) -> ((rec A a b) n) where
  rec (A: U) (a b: A): (k: N) -> U = split
    Z -> Path A a b
    S n -> n_grpd (Path A a b) n

inf_grpd (A: U): U
  = (carrier: A)
  * (eq: (a b: A) -> Path A a b)
  * ((a b: A) -> inf_grpd (Path A a b))
```

As you can see, h-Types properties are just eliminated recursors.

```
isContr      (A: U): U = (x: A) * ((y: A) -> Path A x y)
isProp       (A: U): U = n_grpd A Z
isSet        (A: U): U = n_grpd A (S Z)
isGroupoid   (A: U): U = n_grpd A (S (S Z))
isGrp2       (A: U): U = n_grpd A (S (S (S Z)))
isGrp3       (A: U): U = n_grpd A (S (S (S (S Z))))
...
isInfinityGroupoid (A: U): U = inf_grpd A
```

And finally the definitions as refined h-Types through its properties.

```
PROP          : U = (X:U) * isProp(X)
SET           : U = (X:U) * isSet(X)
GROUPOID      : U = (X:U) * isGroupoid(X)
INF_GROUPOID  : U = (X:U) * isInfinityGroupoid(X)
```

2.4 Operational Semantics

By using such definition of MLTT we can commit the basic properties of dependent theory, computational rules. The proofs are trivial with **refl** function and could be found in **mltt** module. Here is full list of inference rules properties.

$$((x : A) \to f(x))(a) = f(a) \tag{1}$$

$$f = ((x : A) \to f(x)) \tag{2}$$

$$pr_1(a, b) = a \tag{3}$$

$$pr_2(a, b) = b \tag{4}$$

$$(pr_1 p, pr_2 p) = p \tag{5}$$

```
MLTT (A:U): U
  = (Pi_Former:   (A->U)->U)
  * (Pi_Intro:    (B:A->U) (a:A)->B a->(A->B a))
  * (Pi_Elim:     (B:A->U) (a:A)->(A->B a)->B a)
  * (Pi_Comp1:    (B:A->U) (a:A) (f:A->B a) -> Equ (B a)
                  (Pi_Elim B a (Pi_Intro B a (f a)))(f a))
  * (Pi_Comp2:    (B: A->U) (a:A) (f:A->B a) ->
                  Equ (A->B a) f (\(x:A)->Pi_Elim B a f))
  * (Sig_Former: (A->U)->U)
  * (Sig_Intro:  (B:A->U) (a:A)->(b:B a)->Sigma A B)
  * (Sig_Elim1:  (B:A->U)->(_: Sigma A B)->A)
  * (Sig_Elim2:  (B:A->U)->(x: Sigma A B)->B (pr1 A B x))
  * (Sig_Comp1:  (B:A->U) (a:A) (b: B a)->Equ A a
                  (Sigma_Elim1 B (Sigma_Intro B a b)))
  * (Sig_Comp2:  (B:A->U) (a:A) (b:B a)->Equ (B a) b
                  (Sigma_Elim2 B (a,b)))
  * (Id_Former:  A->A->U)
  * (Id_Intro:   (a:A) -> Equ A a a)
  * (Id_Elim:    (a x: A) (C: predicate A a)
                  (d:C a(Id_Intro a))(p:Equ A a x)->C x p)
  * (Id_Comp:    (x y:A)(C: D A)(p: Equ A x y)
                              (b: C x x (reflect A x))
                        (X: Equ U (C x x (reflect A x))
                                  (C x y p)) ->
                    HeteroEqu X b (J A x C b y p)) * Unit
```

3 Runtime Types

The purpose of run-time types is to build solid ground for writing type checkers, evaluating results and printing models to console, writing formally verified infinitely runned free comonadic processes. In one framework we can combine the powerful semantics of CTT and homotopical primitives to prove properties of run-time types in a more simplier and elegant way.

3.1 Proto

Proto module contains very basic common functions of id, composition and non-inductive cartesian non-dependent product.

```
idfun       (A:U) (a:A): A = a
constfun (A B:U) (a:B): A->B = \(_:A)->a
o       (A B C:U) (f: B->C) (g: A->B): A->C = \(x:A)->f(g(x))
both      (A B:U): U = (_:A) * B
```

3.2 Empty and Unit

The very basic types are empty type without elements and unit type with single element. Interpreting types as propositions leads us to efq and neg eliminators of Empty type useful for proving decidability, stablity and hedberg theorem.

```
data Empty =
data Unit = tt

efq         (A: U): Empty -> A = split {}
neg         (A: U): U = A -> Empty
dec         (A: U): U = either A (neg A)
stable      (A: U): U = neg (neg A) -> A
hedberg     (A: U) (h: (a x:A) -> stable (Path A a x)): isSet(A)
```

3.3 Either and Tuple

Either and Tuple are dual inductive data types. They are basic control structures and widely through all base library. The tuple semantically same as both but is ready to pattern machthing.

```
data either (A B: U) = inl (a: A) | inr (b: B)
data tuple (A B: U) = pair (a: A) (b: B)

case (A B C: U) (b: A -> C) (c: B -> C): either A B -> C
fst (A B: U): tuple A B -> A
snd (A B: U): tuple A B -> B
```

3.4 Bool

The basic hedberg theorem could be used to prove that bool is set. Bool is application data type it is rarely used inside run-time library however it is used is real world applications. Here we provide a proof that bool is set by providing the proof that for any booleans path between them is decidable.

```
data bool = false | true

bool_case (A: U) (f t: A): bool -> A = split { false->f; true->t }
bool_dec: (b1 b2: bool) -> dec (Path bool b1 b2)
bool_isSet: isSet bool = hedberg bool bool_dec
```

3.5 Maybe and Nat

Maybe and Nat are very useful for monadic protocol handling and basic number processing. Nat data types is isomorphic to infinite GMP positive integers of Erlang virtual machine. Base library also include formal proof that fix maybe equals nat. Maybe and Nat types are used in list library for handling special cases for empty string.

```
data maybe (A: U) = nothing | just (a: A)
data nat = zero | succ (n: nat)
```

Also we provide constructive proof that $Fix(Maybe) = Nat$.

3.6 List and Stream

List implemented as very simple data type but no simplier that needed to implement type checker. List data type is used to model categorical semantics of dependent type theory as Categories with Famalies by Dybjer in **cwf** module. At the rest list has pretty standard implementation.

```
data list (A: U) = nil | cons (a: A) (as: list A)

null (A: U): list A -> bool
head (A: U): list A -> maybe A
tail (A: U): list A -> maybe (list A)
nth (A: U): nat -> list A -> maybe A
append (A: U): list A -> list A -> list A
reverse (A: U): list A -> list A = rev nil where
map (A B: U) (f: A -> B): list A -> list B = split
zip (A B: U): list A -> list B -> list (tuple A B)
foldr (A B: U) (f: A -> B -> B) (Z: B): list A -> B
foldl (A B: U) (f: B -> A -> B) (Z: B): list A -> B
switch (A: U) (a b: Unit -> list A): bool -> list A
filter (A: U) (p: A -> bool): list A -> list A
length (A: U): list A -> nat
list_eq (A: eq): list A.1 -> list A.1 -> bool
```

Stream

Stream module provide model for infinity streams, IO, and other corecursive models, provide basic theorems about streams bisimulation (equality).

```
data stream (A: U) = cons (x: A) (xs: stream A)

tail (A: U):       stream A -> stream A
head (A: U):       stream A -> A
fib (a b: nat):    stream nat
seq (start: nat): stream nat

data Bisimilar (A: U) (xs ys: stream A) =
     bisimilar (h: Path A (head A xs) (head A ys))
          (t: Bisimilar A (tail A xs) (tail A ys))

bisimilarityIsEquality (A: U) (xs ys: stream A):
     Path U (Bisimilar A xs ys) (Path (stream A) xs ys)
```

3.7 Vector and Fin

Vector and Fin are basic dependent inductive types. Fin types is used in several models, e.g. for naming type constructors in the de Brujin fashion.

```
data vector (A:U)(n:nat) = vnil  | vcons (x:A)(xs:vector A (pred n))
data fin          (n:nat) = fzero | fsucc (pre: fin (pred n))
```

Cubical Syntax lacks of full GADT support, so we need to ban using type constructors and provide its typeable versions:

```
fz (n: nat): Fin (succ n)          = fzero
fs (n: nat): Fin n -> Fin (succ n) = \(x: Fin n) -> fsucc x
```

3.8 IO and Infinity IO

We extracted the pure function for the following IO free structure that encode the two functions for reading and printing string values in Church encoding. The demonstarted program could be seen in Om project [2].

```
data IO (A: U)
  = getLine (fun: String -> IO A)
  | putLine (io: IO A) (str: String)
  | pure (finish: A)

main: U = replicateM 100 (>>= String () getLine putLine)
```

[2] http://github.com/groupoid/om

IO Infinity

The infinity IO sample is also provided by Om project [3] ans demonstrate the cofree comonadic encoding of process running.

```
data IOI.F (A State: U)
   = getLine: (fun: String -> State)
   | putLine: (str: String) (state: State)
   | pure: (finish: A)

data IOI (A State: U) =
     intro: (init: State)
            (action: State -> IOI.F A State)
```

We put Erlang Coinductive Bindings as pure function to comonad instance:

```
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([])]).
```

Extract the Erlang program from Abstract Syntax:

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

And run the program in run-time environment.

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

[3] http://github.com/groupoid/om

4 Control Structures

SKI combinatoric lambda caclulus was developed by Ukranian Logician Moisei Isaievich Sheinfinkel. It could be used as basis for computational foundation just like lambda caclulus. Later Connor McBride evolved the SKI into Applicative type-class. It is known that SKI is just an Church encoded Applicative. Now Applicative type-class is a mandatory tool in any base library.

$$S = apply : (x \to b \to c) \to (x \to b) \to (x \to c) \tag{6}$$

$$K = pure : a \to (x \to a) \tag{7}$$

$$I = SKK \tag{8}$$

We show here the developer's logic behind the scenes of implementation. First we need to cut all basic type classes like Pure, Applicative, Functor, CoFunctor, ContraFunctor, CoContraFunctor and their flip eliminators. Another powerful control structure from came from Category Theory are Monad and CoMonad that is used for modeling IO.

4.1 Signatures

```
pure                  A  -> F A
extract             F A  ->   A
extend      (F A -> B) -> F A -> F B
apply      F (A -> B) -> F A -> F B
fmap          (A -> B) -> F A -> F B
unmap   (F A -> F B) ->   (A -> B)
contra        (B -> A) -> F A -> F B
uncontra (F A -> F B) ->   (B -> A)
cofmap        (B -> A) -> F B -> F A
uncofmap (F B -> F A) ->   (B -> A)
cocontra      (A -> B) -> F B -> F A
uncocontra (F B -> F A) ->   (A -> B)
join         F (F A) -> F A
dup             F A  -> F (F A)
bind       F A ->(A  -> F B)-> F B
```

The we build up a signatures of type clases build as sigma encoded contexts or telescopes. The signatures are made external for code compactification. Then we select $F : U \to U$ functor as common head for sigma types. All quantifiers for members and laws are carried within projections. Projections contains full signature except F. These types could be considered for run-time. Also you may read Oleg Kiselev Type-Classes in ML. [4] Here is we encode type-classes with Sigma and instances with nested tuples.

[4] http://okmij.org/ftp/Computation/typeclass.html

4.2 Run-time types

Coupling signatures with type-class parameters gives us a notion of run-time types. All type-value projections along with parameters are being erased during code extraction.

```
pure_:        U = (F:U->U)*pure_sig F
functor_:     U = (F:U->U)*fmap_sig F
applicative_: U = (F:U->U)*(_:pure_sig F)*(_:fmap_sig F)*(apply_sig F)
monad_:       U = (_:applicative_)*(bind_sig F.1)
```

4.3 Accessors

Accessors are common technique in Type Refinement approach for beautifying the code. As cubical syntax provide only .1 and .2 notion for projections it is good to have named field accessors for all projections.

```
fmap (a: functor_):      fmap_sig  a.1 = a.2
pure (a: applicative_):  pure_sig  a.1 = a.2.1
amap (a: applicative_):  fmap_sig  a.1 = a.2.2.1
ap   (a: applicative_):  apply_sig a.1 = a.2.2.2
```

4.4 Theorems as Proof-carrying Code

The formal definition of control structures comes in Type Refinement approach when we separate the algebraic structure and its properties.

Functor

Such we defined the **functor** as $\Sigma_{F:U->U} fmap_F$. And the Functor has two laws:

$$o(fmap, id) = id \tag{9}$$

$$fmap(o(g,h)) = o(fmap(g), fmap(h)) \tag{10}$$

```
isFunctor (F: functor_): U
    = (id: (A: U) -> (x: F.1 A) ->
           Path (F.1 A) x ((fmap F) A A (idfun A) x))
    * (compose: (A B C:U) (f:B->C) (g:A->B) (x:F.1 A) ->
      Path (F.1 C) (F.2 A C (o A B C f g) x)
       ((o (F.1 A) (F.1 B) (F.1 C)
           (F.2 B C f) (F.2 A B g)) x)) * Unit
```

Applicative

Being applicative property contains applicative laws:

$$ap(pure(id), x) = id(x) \tag{11}$$

$$ap(pure(f), pure(x)) = pure(f(x)) \tag{12}$$

$$ap(ap(ap(pure(o), u), v), w) = ap(u, ap(v, w)) \tag{13}$$

$$ap(u, pure(y)) = ap(pure(f.f(y)), u) \tag{14}$$

```
isApplicative (F: applicative_): U
    = (id:  (A:U) -> (x: F.1 A) ->
       Path (F.1 A) x (ap F A A (pure F (id A) (idfun A)) x))
    * (hom: (A B:U)(f:A->B)(x: A) ->
       Path (F.1 B) (pure F B (f x))
                    (ap F A B (pure F (A->B) f) (pure F A x)))
    * (cmp: (A B C:U)(v: F.1(A->B))(u:F.1(B->C))(w:F.1 A) ->
       Path (F.1 C) (ap F B C u (ap F A B v w))
                    (ap F A C (ap F (A->B) (A->C)
                                  (ap F (B->C) ((A->B)->(A->C))
                    (pure F (ot A B C) (o A B C)) u) v) w))
    * (xchg: (A B:U)(x:A)(u:F.1(A->B))(f:A->B) ->
       Path (F.1 B) (ap F A B u ((pure F) A x))
                    (ap F (A->B) B (pure F ((A->B)->B)
                                           (\(f:A->B)->f(x))) u)) * Unit
```

Then we can define the FUNCTOR as Sigma of fuctor signature and functor properties:

```
FUNCTOR:     U = (f: functor_) * isFunctor f
APPLICATIVE: U = (f: applicative_)
                 * (_: isFunctor (f.1,f.2.2.1))
                 * isApplicative f
MONAD:       U = (f: monad_)
                 * (_: isFunctor (f.1,f.2.2.1))
                 * (_: isApplicative (f.1,f.2.1,f.2.2.1,f.2.2.2.1))
                 * isMonad f
```

We do not show here the full Control library (only the Functor and Applicative instances) due to sizes of the terms.

5 F-Algebras and Recursion Schemes

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

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

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

5.1 Fixpoint and Free Structures

Free and Cofree represent terminated or non-terminated sequence of functorial bindings defined as $\mu x.a + fx$ and $\nu x.a * fx$ where $\mu < fix < \nu$. Also we defined

```
data fix     (F:U->U)= Fix (point: F (fix F))
out_         (F:U->U): fix F -> F (fix F) = split Fix f -> f
in_          (F:U->U): F (fix F) -> fix F = \(x: F (fix F)) -> Fix x
data mu      (F:U->U) (A B:U) = Return (a: A) | Bind (f: F B)
data nu      (F:U->U) (A B:U) = CoBind (a: A)        (f: F B)
data free    (F:U->U) (A:U)   = Free    (_: fix (mu F A))
data cofree  (F:U->U) (A:U)   = CoFree  (_: fix (nu F A))
unfree       (F:U->U) (A:U):  free   F A -> fix (mu F A)
uncofree     (F:U->U) (A:U):  cofree F A -> fix (nu F A)
```

5.2 Catamorphism and Futumorphism

```
cata (A: U) (F: functor_) (alg: F.1 A -> A) (f: fix F.1): A
   = alg (F.2 (fix F.1) A (cata A F alg) (out_ F.1 f))

futu (A: U) (F: functor_)
     (f: A -> F.1 (free F.1 A)) (a: A): fix F.1
  = Fix (F.2 (free F.1 A) (fix F.1) (\(z: free F.1 A) -> w z) (f a)) where
  w: free F.1 A -> fix F.1 = split
    Free x -> unpack_fix x where
  unpack_free: mu F.1 A (fix (mu F.1 A)) -> fix F.1 = split
    Return x -> futu A F f x
    Bind g -> Fix (F.2 (fix (mu F.1 A)) (fix F.1)
                       (\(x: fix (mu F.1 A)) -> w (Free x)) g)
  unpack_fix: fix (mu F.1 A) -> fix F.1 = split
    Fix x -> unpack_free x
```

5.3 Anamorphism and Histomorphism

```
ana  (A: U) (F: functor_) (coalg: A -> F.1 A) (a: A): fix F.1
   = Fix (F.2 A (fix F.1) (ana A F coalg) (coalg a))

histo (A:U) (F: functor_)
      (f: F.1 (cofree F.1 A) -> A) (z: fix F.1): A
  = extract A ((cata (cofree F.1 A) F (\(x: F.1 (cofree F.1 A)) ->
    CoFree (Fix (CoBind (f x) ((F.2 (cofree F.1 A)
    (fix (nu F.1 A)) (uncofree F.1 A) x)))))) z) where
  extract (A: U): cofree F.1 A -> A = split
    CoFree f -> unpack_fix f where
  unpack_fix: fix (nu F.1 A) -> A = split
    Fix f -> unpack_cofree f where
  unpack_cofree: nu F.1 A (fix (nu F.1 A)) -> A = split
    CoBind a -> a
```

5.4 Inductive and Coninductive Types

```
ind (A: U) (F: U -> U): U
  = (in_: F (fix F) -> fix F)
  * (in_rev: fix F -> F (fix F))
  * (fold_: (F A -> A) -> fix F -> A)
  * Unit

inductive (F: U -> U) (A: U) (X: functor F): ind A F
  = (in_ F, out_ F, cata A F X, tt)

coind (A: U) (F: U -> U): U
  = (out_: fix F -> F (fix F))
  * (out_rev: F (fix F) -> fix F)
  * (unfold_: (A -> F A) -> A -> fix F)
  * Unit

coinductive (F: U -> U) (A: U) (X: functor F): coind A F
  = (out_ F, in_ F, ana A F X, tt)
```

6 Algebra

6.1 Monoid and Commutative Monoid

```
isAssociative (M: U) (op: M -> M -> M) : U

hasIdentity (M : U) (op : M -> M -> M) (id : M) : U
  = (_ : hasLeftIdentity M op id)
  * (hasRightIdentity M op id)

isMonoid (M: SET): U
  = (op: M.1 -> M.1 -> M.1)
  * (_: isAssociative M.1 op)
  * (id: M.1)
  * (hasIdentity M.1 op id)

isCMonoid (M: SET): U
  = (m: isMonoid M)
  * (isCommutative M.1 m.1)
```

6.2 Group and Abelian Group

```
isGroup (G: SET): U
  = (m: isMonoid G)
  * (inv: G.1 -> G.1)
  * (hasInverse G.1 m.1 m.2.2.1 inv)

isAbGroup (G: SET): U
  = (g: isGroup G)
  * (isCommutative G.1 g.1.1)
```

6.3 Ring and Abelian Ring

```
isRing (R: SET): U
  = (mul: isMonoid R)
  * (add: isAbGroup R)
  * (isDistributive R.1 add.1.1.1 mul.1)

isAbRing (R: SET): U
  = (mul: isCMonoid R)
  * (add: isAbGroup R)
  * (isDistributive R.1 add.1.1.1 mul.1.1)
```

7 Category Theory

More formal, precategory A consists of the following: (i) A type A_0, whose elements are called objects. We write $a : A$ for $a : A_0$. (ii) For each $a, b : A$, a set $hom_A(a, b)$, whose elements are called arrows or morphisms. (iii) For each $a : A$, a morphism $1_a : hom_A(a, a)$, called the identity morphism. (iv) For each $a, b, c : A$, a function $hom_A(b, c) \to hom_A(a, b) \to hom_A(a, c)$ called composition, and denoted $g \circ f$. (v) For each $a, b : A$ and $f : hom_A(a, b)$, we have $f = 1_b \circ f$ and $f = f \circ 1_a$. (vi) For each $a, b, c, d : A$ and $f : hom_A(a, b)$, $g : hom_A(b, c)$, $h : homA(c, d)$, we have $h \circ (g \circ f) = (h \circ g) \circ f$.

7.1 Signature

We divide the carrier, morphisms as algebraic structure of category definition (in essense CT is an abstract algebra of functions) and laws (theorems) or categorical properies, defined as equalities on morphisms:

```
cat: U = (A: U) * (A -> A -> U)
```

7.2 Accessors

Due to the nature of minimalistic syntax for each algebraic structure should be defined the accessors for its projections to avoid long projection composition, e.g. 2.2.2.2.2.1.

```
carrier (C: precategory): U = C.1.1
hom     (C: precategory) (a b: carrier C): U = C.1.2 a b
path    (C: precategory) (x: carrier C): hom C x x = C.2.1 x
compose (C: precategory) (x y z: carrier C) (f: hom C x y)
        (g: hom C y z): hom C x z = C.2.2.1 x y z f g
```

7.3 Theorems

We define theorems section as predicate in a form of Sigma which is usual a carrier for theorems.

```
isPrecategory (C: cat): U
  = (id:       (x: C.1) -> C.2 x x)
  * (c:        (x y z:C.1)->C.2 x y->C.2 y z->C.2 x z)
  * (homSet:   (x y: C.1) -> isSet (C.2 x y))
  * (left:     (x y: C.1) -> (f: C.2 x y) ->
               Path (C.2 x y) (c x x y (id x) f) f)
  * (right:    (x y: C.1) -> (f: C.2 x y) ->
               Path (C.2 x y) (c x y y f (id y)) f)
  * (compose:  (x y z w: C.1) -> (f: C.2 x y) ->
               (g: C.2 y z) -> (h: C.2 z w) ->
               Path (C.2 x w) (c x z w (c x y z f g) h)
               (c x y w f (c y z w g h))) * Unit
```

```
category: U = (C: cat) * isPrecategory C
```

7.4 Terminal and Initial Objects

Initiality and Terminality encoded directly from its definition.

```
isInitial  (C: precategory) (x: carrier C): U
        = (y: carrier C) -> isContr (hom C x y)
isTerminal (C: precategory) (y: carrier C): U
        = (x: carrier C) -> isContr (hom C x y)

initial  (C: precategory): U = (x:carrier C) * isInitial C x
terminal (C: precategory): U = (y:carrier C) * isTerminal C y
```

7.5 Functors

```
functor (A B: precategory): U
  = (ob:   carrier A -> carrier B)
  * (mor:  (x y: carrier A) ->
           hom A x y -> hom B (ob x) (ob y))
  * (id:   (x: carrier A) ->
           Path (hom B (ob x) (ob x))
                (mor x x (path A x)) (path B (ob x)))
  * (cmp:  (x y z: carrier A) -> (f: hom A x y) -> (g: hom A y z) ->
           Path (hom B (ob x) (ob z)) (mor x z (compose A x y z f g))
           (compose B (ob x) (ob y) (ob z) (mor x y f) (mor y z g))) * Unit
```

8 Conclusion

Table 2: Groupoid Infinity Base Library

Module	LOC	Module	LOC	Module	LOC	Module	LOC
algrbra	360	binnat	147	bool_theory	118	cat	33
cat_theory	1094	circle	66	collection	22	control	71
csystem	508	cwf	82	dagand	54	discor	43
equiv	60	girard	51	hedberg	28	helix	242
int	38	interval	31	iso	68	iso_pi	39
iso_prop	8	iso_sigma	57	list	63	list_theory	26
maybe	19	maybe_nat	29	maybe_theory	47	mltt	55
nat	65	nat_theory	123	ordinal	67	path	39
pi	15	prop	48	proto	20	proto_equiv	16
proto_path	16	pushuout	6	quotient	31	recursion	98
retract	10	set	96	sigma	36	stream	10
stream_theory	125	trunc	32	univ	136	vector	7

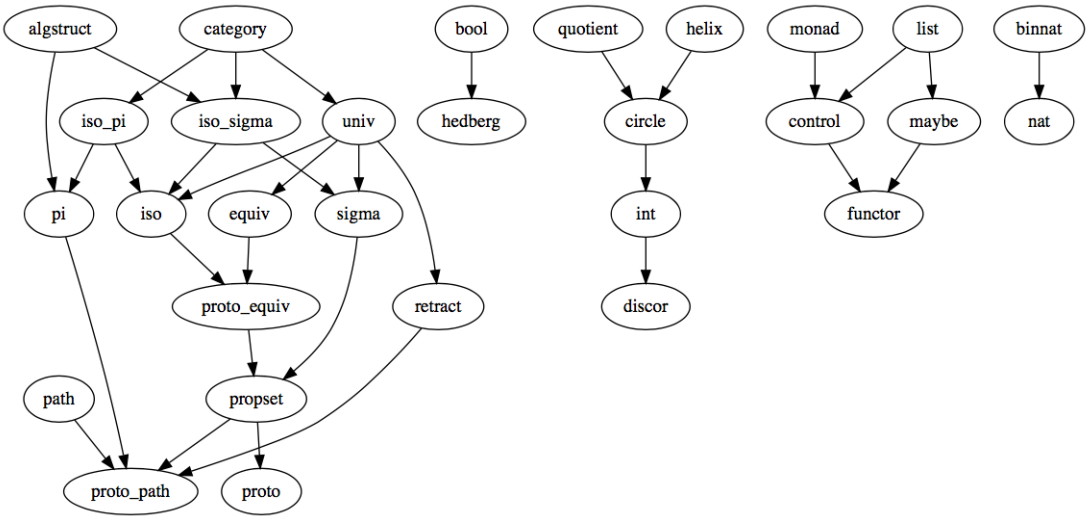

Figure 1: Base library and its dependencies used in article

References

[1] Errett Bishop. *Foundations of constructive analysis*. McGraw-Hill series in higher mathematics. McGraw-Hill, 1967.

[2] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. In *Cubical Type Theory: a constructive interpretation of the univalence axiom*, volume abs/1611.02108, 2017.

[3] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *In Venice Festschrift*, pages 83–111. Oxford University Press, 1996.

[4] P. Martin-Löf and G. Sambin. *Intuitionistic type theory*. Studies in proof theory. Bibliopolis, 1984.

[5] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.