Mathematical Components for Cubical

Maksym Sokhatskyi [1] and Pavlo Maslianko [1]

[1] National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnical Institute" April 11, 2018

Abstract

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

1 Intro

This library is dedicated to cubical-compatible type checkers 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), 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 with HoTT book.

Table 1: Types Taxonomy

NR+ND	**R+ND**	**NR+D**	**R+D**
unit	nat	path	vector
bool	list	proto	fin
either		iso	
maybe		equiv	

NR*ND	**R*ND**	**NR*D**	**R*D**
pure	stream	sigma	cat
functor		setoid	prop
applicative			set
monad			groupoid

2 MLTT

2.1 Pi

2.2 Sigma

2.3 Identity Type

```
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 =
inv      (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) =
trans  (A B: U) (p: Path U A B) (a : A): 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)
```

3 Runtime Types

3.1 Empty and Unit

```
data Empty =
data Unit = tt
```

3.2 Bool

```
data bool = false | true
neg_: bool -> bool
or_:  bool -> bool -> bool
and_: bool -> bool -> bool
bool_case (A: U) (f t: A): bool -> A
bool_eq: bool -> bool -> bool
```

3.3 Either and Tuple

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

3.4 Maybe and Nat

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

3.5 List

```
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
zipWith (A B C: U) (f: A -> B -> C): list A -> list B -> list C
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
uncons (A: U): list A -> maybe ((a: A) * (list A))
length (A: U): list A -> nat
list_eq (A: eq): list A.1 -> list A.1 -> bool
```

3.6 Stream

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

tail (A: U): stream A -> stream A = split cons x xs -> xs
head (A: U): stream A -> A = split cons x xs -> x
fib (a b: nat): stream nat = cons a (fib b (add a b))
seq (start: nat): stream nat = cons start (seq (succ start))
ones:  stream nat = cons one ones
zeros: stream nat = cons zero zeros
nats:  stream nat = seq zero
```

3.7 Vector and Fin

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

3.8 IO

3.9 IOI

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

$$
\begin{tikzcd}
F\,\mu F \arrow[r, "in"] \arrow[d, "F\,(\!|\varphi|\!)"'] & \mu F \arrow[d, "(\!|\varphi|\!)"] \\
FC \arrow[r, "\varphi"] & C
\end{tikzcd}
\qquad
\begin{tikzcd}
C \arrow[r, "\phi"] \arrow[d, "{[\![\varphi]\!]}"'] & F\,C \arrow[d, "{F\,[\![\varphi]\!]}"] \\
\nu F \arrow[r, "out"] & F\nu F
\end{tikzcd}
$$

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

4.1 Fixpoint and Free Structures

```
data freeF   (F:U->U)(A B:U)= ReturnF (a:A) | BindF(f:F B)
data cofreeF (F:U->U)(A B:U)= CoBindF (a:A) (f: F B)
data free    (F:U->U)(A:U)  = Free    (_:fix(freeF F A))
data cofree  (F:U->U)(A:U)  = CoFree  (_:fix(cofreeF F A))

unfree   (A: U) (F: U -> U): free   F A -> fix (freeF   F A)
  = split Free   a -> a

uncofree (A: U) (F: U -> U): cofree F A -> fix (cofreeF F A)
  = split CoFree a -> a
```

4.2 Catamorphism

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

4.3 Anamorphism

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

4.4 Inductive 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)
```

4.5 Coinductive Types

```
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)
```

5 Algebraic Structures

```
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)

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)

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)
```

6 Category Theory

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

6.1 Precategory

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

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

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
```

6.2 Terminal and Initial Objects

```
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)

initialObject (C: precategory): U
  = (x: carrier C)
  * isInitial C x

terminalObject (C: precategory): U
  = (y: carrier C)
  * isTerminal C y
```

6.3 Functor

```
catfunctor (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
```

7 Proto

```
case    (A B C: U)(b:A->C)(c:B->C):or A B->C = split{inl x->b(x);inr y->c(y)}
fst       (A B: U): tuple A B -> A = split pair a b -> a
snd       (A B: U): tuple A B -> B = split pair a b -> b
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)
and       (A B: U): U = (_:A) * B

efq         (A: U): Empty -> A = split {}
neg         (A: U): U = A -> Empty
dneg   (A:U) (a:A): neg (neg A) = \(h: neg A) -> h a
dec         (A: U): U = or A (neg A)
stable      (A: U): U = neg (neg A) -> A
```