Mathematical Components for Cubical Syntax

Maksym Sokhatskyi [1] and Pavlo Maslianko [1]

[1] National Technical University of Ukraine
Igor Sikorsky Kyiv Polytechnical Institute
April 28, 2018

Abstract

The introduction of path spaces and its eliminators in cubical type theory (CTT) as core type checker primitives call for re-examination the proofs of 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 the 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 [4] 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 fits 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

Usually, from the mathematical point of view, there are no differences between different syntactic proofs of the same theorem. However, from the programming point of view, we can think of code reuse and precisely defined math libraries that reduce the overall code size and provide simplicity for operating complex structures (most heavy one is the categorical library). So in this work, we will focus on proper decoupling and more programming friendly base library still usable for mathematicians.

Research object. The homotopy type theory base libraries in Agda [1], Cubical[2], Lean[3], and Coq[4]. While modern Lean and Agda has the cubical mode, Coq lacks the computational semantics of path primitives while has HoTT 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 structures and applicative programming (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 with [0,1] interval — 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 topics: **MLTT**, **Runtime Types**, **Control Structures**, **Recursive Schemes**, **Abstract Algebra**, **Category Theory**. But the library has even more modules, that exceed the scope of this article so you may refer to the source code repository [5]. Also, we created a web page that can easily be read with phone [6].

We redefined the Cubical Syntax in LALR grammar and implemented syntax extension for Erlang language [7]. If possible terms could be exracted to Om syntax [8].

All terms in this article will be given in this syntax.

```
def := data id tele = sum + id tele : exp = exp +
       id tele : exp where def

exp := cotele*exp + cotele→exp + exp→exp + (exp) + app + id +
       (exp,exp) + \ cotele→exp + split cobrs + exp.1 + exp.2
```

[1] https://github.com/HoTT/HoTT-Agda
[2] https://github.com/mortberg/cubicaltt
[3] https://github.com/gebner/hott3
[4] https://github.com/HoTT/HoTT
[5] https://github.com/groupoid/infinity
[6] https://groupoid.space/mltt/types
[7] https://github.com/groupoid/infinity/blob/master/src/cub_parser.yrl
[8] https://github.com/groupoid/om/blob/master/src/om_parse.erl

```
    0 := #empty          imp    := [ import id ]
  brs := 0 + cobrs       tele   := 0 + cotele
  app := exp exp         cotele := ( exp : exp ) tele
   id := [ #nat ]        sum    := 0 + id tele + id tele | sum
  ids := [ id ]          br     := ids → exp
codec := def dec         mod    := module id where imp def
  dec := 0 + codec       cobrs  := | br brs
```

The BNF notation consists of 1) telescopes (contexts or sigma chains); 2) inductive data definitions (sum chains); 3) split eliminator; 4) branches of split eliminators; 5) pure dependent type theory syntax. It also has where, import, module constructions.

Types Taxonomy

Types taxonomy shows us the core types categorized by several axes: 1) dependent (D) and non-dependent (ND) terms; 2) recursive trees (R) or non-recursive (NR) data types; 3) inductive (+) or coinductive (*) types; 4) higher inductive (H) type.

Table 1: Types Taxonomy

NR+ND	**R+ND**	**NR+D**	**R+D**	**H NR**
unit	nat	exists	vector	I
bool	list		fin	circle
either			W	suspension
maybe				pushout
empty				
NR*ND	**R*ND**	**NR*D**	**R*D**	**H R**
setoid	stream	sigma	cat	trunc
functor		equiv	n-groupoid	quotient
applicative		iso	∞-groupoid	
monad		fiber		

This library is dedicated to type checkers with cubical syntax, based on interval [0,1] and MLTT as a core. Please refer to original paper of cubicaltt [4] which also given in the footnote [9]. The base library is founded on top of cubical modules each fall into one of the following categories:

(i) MLTT Types: **pi**, **sigma**, **path**; (ii) Set Theory: **prop**, **set**, **ordinal**, **hedberg**; (iii) Run-Time Inductive Types: **proto**, **maybe**, **bool**, **nat**, **list**, **stream**, **vector**; (iv) Abstract Algebra in **algebra** module; (v) Control Structures in **control** module; (vi) Recursion Schemes in **recursion** module; (vii) Category Theory: **cat**, **fun**, **sip**, **cones**, **adj**; (viii) Univalent Foundations: **equiv**, **retract**, **iso**, **iso_pi**, **iso_sigma**, **univ**; (ix) Higher Inductive Types: **interval**, **circle**, **pushout**, **suspension**, **quotient**, **trunc**. (x) Process Calculus in **process** module; (xi) Categories with Families: **cwf**, **csystem**; (xii) Type Checker will be in **infinity** module.

This library is best to read guided by HoTT book [13]. We tried to make it definitionally compatible with the contemporary math foundations.

[9] http://www.cse.chalmers.se/~coquand/cubicaltt.pdf

2 Martin-Löf Type Theory

As Martin-Löf Type Theory is used as the modeling language, 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, while **cwf** and **csystem** modules contain its categorical semantics.

2.1 Pi and Sigma

Pi and Sigma modules provide basic theorems about dependent products and sum. Here are 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. Basic Pi and Sigma definitions are given in **pi** and **sigma** modules.

```
Pi      (A:U) (B:A->U): U = (x:A) -> B(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
```

Dependent functional extensionality.

```
piExt (A: U) (B: A -> U) (f g: (x:A) -> B x)
      (p: (x:A) -> Path (B x) (f x) (g x))
    : Path ((y:A) -> B y) f g = <i> \(a: A) -> (p a) @ i
```

Induction principle for Sigma types.

```
sigRec (A:U) (B:A -> U) (C:U)
       (g: (x:A) -> B(x) -> C) (p: Sigma A B)
     : C = g p.1 p.2
sigInd (A:U) (B:A -> U) (C: Sigma A B -> U)
       (p: Sigma A B) (g:(a:A)(b:B(a)) -> C(a,b))
     : C p = g p.1 p.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 at run-time. For modeling propositional equality later in 1984 was introduced Equ type. [10] However, unlike Pi and Sigma, the eliminator J of Equ type is not derivable in MLTT [8, 4, 13]. Here we give the constructive J definition in CTT. Basic laws of Identity Type are given in **proto_path** module.

```
Path    (A: U) (a b: A): U
singl   (A: U) (a: A): U = (x: A) * Path A a x
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)
```

We can build setoid [3] definition using sym, refl, and cong which represent symmetry, reflexivity, and congruence. Singleton is defined as Sigma with Path theorem. Contractability of singletons, substitution and transport are used for proving the J eliminator [13].

2.3 ∞-Groupoid and h-Types

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

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

As the foundation, we provide recursive and corecursive versions of groupoid definitions. h-Types [13] (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)
```

The groupoids are defined in **path** module.

2.4 Example: 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.

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

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

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

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

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

Here is the full list of inference rules properties. We grouped them as in MLTT paper [10]. The system lack Nat and List types as they can be church encoded.

```
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
```

Trivial check that our host type system preserves the dependent theory laws. See computational rules in **mltt** module.

```
instance (A: U): MLTT A
  = ( Pi A, lambda A, app A, comp1 A, comp2 A,
      Sigma A, pair A, pr1 A, pr2 A, comp3 A, comp4 A, comp5 A,
      Equ A, reflect A, J A, comp6 A, tt )
```

3 Runtime Types

The purpose of run-time types is to build the solid ground for writing type checkers, evaluating results and printing models to console, writing formally verified infinitely run 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 simple and elegant way.

3.1 Proto

The **proto** module contains very basic common functions of id, composition and synonyms for intro and eliminators of Pi type, in the dependent and non-dependent fashion. Th **proto** contains almost all short cuts to internal language constructions, while **pi** and **sigma** modules contains only dependent versions.

```
idfun     (A:U) (a:A): A = a
const   (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))
```

3.2 Empty and Unit

The very basic types are empty type without elements and unit type with the single element. Interpreting types as propositions lead us to efq and neg eliminators of the empty type useful for proving decidability, stability and analog of Hedberg theorem [6].

```
data Empty =
emptyRec  (C: U): empty -> C = split {}
emptyInd  (C: empty->U): (z:empty) -> C z = split {}

data Unit = tt
unitRec    (C: U) (x: C): unit -> C = split tt -> x
unitInd    (C: unit -> U) (x: C tt): (z:unit) -> C z = split tt -> x

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 matching.

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

```
tupleRec  (A B C: U) (c: (x:A) (y:B) -> C)
        : (x: tuple A B) -> C
        = split pair a b -> c a b
tupleInd  (A B:U) (C: tuple A B -> U)
          (c: (x:A) (y:B) -> C(pair x y))
        : (x: tuple A B) -> C x
        = split pair a b -> c a b

eitherRec (A B C: U) (b: A -> C) (c: B -> C)
        : either A B -> C
        = split { inl x -> b(x) ; inr y -> c(y) }
eitherInd (A B: U) (C: either A B -> U)
          (x: (a: A) -> C (inl a))
          (y: (b: B) -> C (inr b))
        : (x: either A B) -> C x
        = split { inl i -> x i ; inr j -> y j }
```

3.4 Bool

The basic **hedberg** theorem module could be used to prove that bool is set. The **bool** module 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

a1: U = bool -> bool
a2: U = bool -> bool -> bool

negation: a1 = split { false -> true; true -> false}
or:  a2 = split { false->idfun bool;true->lambda bool(bool)true }
and: a2 = split { false->lambda bool(bool)false;true->idfun bool }

boolRec (A: U) (f t: A) : bool -> A = split { false -> f ; true -> t }
boolInd (A: bool -> U) (f: A false) (t: A true)
      : (n:bool) -> A n = split { false  -> f ; true -> t }

boolDec : (b1 b2: bool) -> dec (Path bool b1 b2)
boolSet : isSet bool = hedberg bool boolDec
```

3.5 Maybe and Nat

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

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

natRec (C:U) (z: C) (s: nat->C->C)
     : (n:nat) -> C
     = split { zero -> z ; succ n -> s n (natRec C z s n) }
natInd (C:nat->U) (z: C zero) (s: (n:nat) -> C(n) -> C(succ n))
     : (n:nat) -> C(n)
     = split { zero -> z ; succ n -> s n (natInd C z s n) }

maybeRec (A P: U) (n: P) (j: A->P)
       : maybe A->P = split { nothing -> n; just a -> j a}
maybeInd (A: U) (P: maybe A -> U) (n: P nothing) (j: (a: A) -> P (just a))
       : (a: maybe A) -> P a = split { nothing -> n ; just x -> j x }
```

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

3.6 List and Stream

The **list** module implemented as very simple data type but no simpler than needed to implement the type checker. List data type is used to model categorical semantics of dependent type theory as Categories with Families by Dybjer in **cwf** module. At the rest, the list has a 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
map      (A B: U): (A -> B) -> list A -> list B
zip      (A B: U): list A -> list B -> list (tuple A B)
foldr    (A B: U): (A -> B -> B) -> B -> list A -> B
switch     (A: U): (Unit -> list A) -> bool -> list A
filter     (A: U): (A -> bool) -> list A -> list A
length     (A: U): list A -> nat
```

The **stream** module provide the 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)

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 are used in several models, e.g. for naming type constructors in the de Bruijn 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 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 Example: 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 demonstrated program could be seen in Om project [10].

```
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)
```

The infinity IO sample is also provided by Om project [11] and demonstrate the cofree comonadic encoding of the running process.

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

[10] https://github.com/groupoid/om/blob/master/priv/normal/Morte/recursive
[11] https://github.com/groupoid/om/blob/master/priv/normal/Morte/corecursive

4 Control Structures

SKI combinatoric lambda calculus was developed by Ukranian Logician Moisei Isaievich Sheinfinkel. It could be used as the basis for the computational foundation just like lambda calculus. Later Connor McBride [11] evolved the SKI into Applicative type-class. It is known that SKI is just a Church encoded Applicative. Now Applicative type-class is a mandatory tool in any base library. All control structures are given in **control** module.

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

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

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

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, Contra-Functor, 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

Here is given all signatures supported in control base library, mainly functors and monads.

```
pure_sig                  A  -> F A
extract_sig             F A  ->   A
extend_sig      (F A -> B) -> F A -> F B
appl_sig        F (A -> B) -> F A -> F B
fmap_sig          (A -> B) -> F A -> F B
unmap_sig     (F A -> F B) ->  (A -> B)
contra_sig        (B -> A) -> F A -> F B
uncontra_sig  (F A -> F B) ->  (B -> A)
cofmap_sig        (B -> A) -> F B -> F A
uncofmap_sig  (F B -> F A) ->  (B -> A)
cocontra_sig      (A -> B) -> F B -> F A
uncocontra_sig (F B -> F A) ->  (A -> B)
join_sig        F (F A) -> F A
dup_sig           F A  -> F (F A)
bind_sig        F A ->(A  -> F B)-> F B
```

We build up signatures of type classes build as sigma encoded contexts or telescopes. The signatures are made external to 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 contain full signature except F. These types could be considered for run-time. Also, you may read Oleg Kiselev Type-Classes in ML. [12] Here is we encode type-classes with Sigma and instances with nested tuples.

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

4.2 Run-time types

Coupling signatures with type-class parameters give 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)
                          * (   appl_sig F)

monad: U = (_: applicative)
         * (   bind_sig F.1)
```

Accessors

Accessors are the 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): appl_sig a.1 = a.2.2.2
```

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 functor 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
```

4.3 Example: List is a Functor

We do not show here the full Control library (only the Functor and Applicative instances) due to sizes of the terms. Here is sample list functor instance defined in **list_theory** module.

```
LIST: functor = (list,map)
laws: FUNCTOR = (LIST,list_id,list_compose,tt)
```

5 Recursion Schemes

An F-algebras provide us a generalized notion of folds over a parametrized inductive structures [12]. The formalism of recursion schemes was developed by Varmo Vene and Nicola Gambino [5, 14] and categorically by Bart Jacobs, Henning Basold and Herman Geuvers [9, 2]. Recursion Schemes, Fixpoint, Free and CoFree protocol data types are defined in **recursion** module. It contains: Ana, Cata, Para, Hylo, Zygo, PrePro, PostPro, Apo, Gapo, Futu, Histo, Chrono, Mutu, Meta, MCata, Inductive and CoInductive definitions.

The core notion for recursion schemes are F-algebras. $(\mu F, in)$ 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

The Fixpoint recursive data type has two eliminators: in and out. The μ and ν data types define the A+F(B) and A*F(B) non-recursive inductive and coinductive type respectively.

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 unpacking routines for the fix and for (co-)free structures.

```
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

Catamorphism denotes the homomorphism from the initial algebra to some other algebras. A generalization of folds.

```
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))
```

Histomorphism is a recursion based on a recursive cofree structure with unpacking control. A generalization of cata.

```
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
```

5.3 Anamorphism and Histomorphism

Anamorphism denotes the homomorphism to terminal coalgebra of an endofunctor from some other coalgebras. A generalization of unfolds.

```
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))
```

Futumorphism is a recursion based on a recursive free structure with unpacking control. A generalization of anamorphisms.

```
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
```

5.4 Example: Inductive and Coninductive Types

Inductive types are given as direct encoding of the commutative square as in [7]. For inductive and coinductive data types the first two fields of sigma are flipped.

```
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)
```

```
    * (cofree_: (F (cofree F A) -> A) -> fix F -> A)
    * Unit

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)
    * (free_: (A -> F (free F A)) -> A -> fix F)
    * Unit
```

The two instances with corresponding recursors are then given by recursion schemes.

```
inductive (A: U) (F: functor): ind A F.1
    = (in_ F.1,out_ F.1,cata A F,futu A F,tt)
coinductive (A: U) (F: functor): coind A F.1
    = (out_ F.1,in_ F.1,ana A F,histo A F,tt)
```

Here is the simple example how two instances (inductive and coinductive) of the list could be defined.

```
ind_list   (T:U): ind   list T = inductive   (list,map) T
coind_list (T:U): coind list T = coinductive (list,map) T
```

6 Abstract Algebra

Algebraic structures in **algebra** are the natural math companion to more developer oriented **control** module, which is also sort of algebraic structures. Here you can find everything except Fields and Modules yet. Only up to Abelian Groups and Rings. However, this structures is enough to define categories of Abelian groups and Grothendieck adjunctions.

6.1 Monoid and Commutative Monoid

The monoid is a simple algebra with one binary operation: M -¿ M -¿ M. By definition, the monoid is associative and has left and right identities. The commutative monoid is one predicate stronger, expects the operation to be commutative.

```
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

A group is a Monoid with inverse morphisms.

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

An abelian group is a commutative group with inverse morphisms.

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

6.3 Ring and Abelian Ring

A ring has two operations, one (*) is a Monoid and other (+) is an Abelian Group. A ring is supposed to be distributive across (*) and (+) operations.

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

Abelian Ring is a Ring where (*) operation is a commutative monoid.

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

7 Category Theory

The general notion of category is given as in [1] and [13]. The usual notion of category is called precategory while category is saturated precategories. The Category Theory consists of five modules: 1) **cat** for precategory, category, and (co-)terminal objects; 2) **fun** for categorical functor, its instances (id, compose), coslice category construction, universal arrows, and equivalence theorems; 3) **sip** for structure definition and structure identity principle theorem; 4) **cones** for cones, pullback definitions, and theorems; 5) **adj** for natural transformations and adjunctions. As a stress example for categorical library we took the C-Systems model of Vladimir Voevodsky implemented in **csystem** module on top of our categorical library. C-Systems are a categorical model of dependent type theory, similar to Dybjer's Categories with Families (**cwf** module).

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

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

7.1 Precategories

We define theorems section as predicate in a form of Sigma which is usual a carrier for theorems. 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 $h \circ (g \circ f) = (h \circ g) \circ f$.

```
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

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

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
```

7.2 Rezk Completion

"Every fully faithful and essentially surjective functor is an equivalence of categories" in classical set-based category theory is equivalent to the axiom of choice.

(i) For strict categories, it is equivalent to the axiom of choice. (ii) For precategories, it is underivable. (iii) For saturated categories, it is derivable without the axiom of choice.

Anything we say about objects of a category is invariant under isomorphism.

```
iso (C: precategory) (A B: carrier C): U
  = (f: hom C A B)
  * (g: hom C B A)
  * (_: Path (hom C A A) (compose C A B A f g) (path C A))
  * (    Path (hom C B B) (compose C B A B g f) (path C B))

isCategory (C: precategory): U
  = (A: carrier C) -> isContr ((B: carrier C) * iso C A B)

category: U = (C: precategory) * isCategory C
```

Such category saturation or completion is called Rezk competition [1]. Source code in Coq of Rezk completion for comparison could be taken here [13].

7.3 Terminals and Cones

Initiality and Terminality encoded directly from its definitions.

```
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

hasCospanCone (C: precategory) (D: cospan C) (W: carrier C): U
  = (f : hom C W D.2.1.1)
  * (g : hom C W D.2.2.1)
  * Path (hom C W D.1) (compose C W D.2.1.1 D.1 f D.2.1.2)
                       (compose C W D.2.2.1 D.1 g D.2.2.2)

cospanCone (C: precategory) (D: cospan C): U
  = (W: carrier C)
  * hasCospanCone C D W
```

Cones and Pullback definitions, properties and theorems are in **cones** module.

7.4 Functors

Categorical functor definition and instances are in **fun** module.

```
catfunctor (A B: precategory): U
  = (ob:   carrier A -> carrier B)
```

[13] https://github.com/benediktahrens/rezk_completion/blob/master/precategories.v

```
  * (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
```

7.5 Natural Transformations and Andjunctions

Natural Transformations and Adjunctions are defined in **adj** module.

```
isNaturalTrans (C D: precategory)
               (F G: catfunctor C D)
               (eta: (x: carrier C) -> hom D (F.1 x) (G.1 x)): U
  = (x y: carrier C) (h: hom C x y) ->
    Path (hom D (F.1 x) (G.1 y))
         (compose D (F.1 x) (F.1 y) (G.1 y) (F.2.1 x y h) (eta y))
         (compose D (F.1 x) (G.1 x) (G.1 y) (eta x) (G.2.1 x y h))

ntrans (C D: precategory) (F G: catfunctor C D): U
  = (eta: (x: carrier C) -> hom D (F.1 x) (G.1 x))
  * (isNaturalTrans C D F G eta)

adjoint (C D: precategory) (F: catfunctor D C) (G: catfunctor C D) : U
  = (unit: ntrans D D (idFunctor D) (compFunctor D C D F G))
  * (counit: ntrans C C (compFunctor C D C G F) (idFunctor C))
  * areAdjoint C D F G unit counit
```

7.6 Examples: Grothendieck Group and C-Systems

Here we give two complex examples that cover the usage of all categorical library (five modules) and abstract **algebra** module.

Example 1: Grothendieck Adjoint

The forgetful functor U from abelian groups to commutative monoids has a left adjoint G. This is called group completion. A standard presentation of the group completion is the Grothendieck group of a commutative monoid. As such group completion plays a central role in K-theory [16].

The top level example of using universal **algebra** module and categorical modules is given as Grothendieck Adjoint in categories of Abelian groups **Ab** and commutative monoids **CMon** of two functors: forgetful from, **Ab** to **CMon**; and Grothendieck, from **CMon** to **Ab**.

```
Ab:   precategory = ((abgroup,abgrouphom),(id,c,HomSet,L,R,C)) ...
CMon: precategory = ((cmonoid,cmonoidhom),(id,c,HomSet,L,R,C)) ...
```

```
grothendieck: catfunctor CMon Ab = ...
forgetful:    catfunctor Ab CMon = ...

grothendieck_adjoint: adjoint Ab CMon grothendieck forgetful
  = transport (<i> adjoint Ab CMon (gro_ifeq @ i) forgetgul)
              (univarr_adjoint CMon Ab forgetful gro_univarr)
```

Example 2: C-Systems

In the **csystem** module you can find a proof that Universe Category is enough to define the C-System. The code is based on Voevodsky's paper about C-systems defined by a universe in a category [15].

```
CSystem: U
   = (ob: nat -> U)
   * (hom: Sigma nat ob -> Sigma nat ob -> U)
   * (isC: isPrecategory (Sigma nat ob, hom))
   * (c0: isC0System ob hom isC)
   * isCSystem (ob, hom, isC, c0)

uc : U
   = (C: precategory)
   * (_: isCategory C)
   * (pt: terminal C)
   * (V: carrier C)
   * (VT : carrier C)
   * (p: hom C VT V)
   * ((f: homTo C V) -> hasPullback C (V, f, VT, p))

ucToCSystem: uc -> CSystem = ...
```

8 Conclusion

We finished the story about Groupoid Infinity base library. Despite we were limited with paper size and could not cover homotopical types and other run-time types, we tried to put into overview the very strong core of the base library by providing different sections of modules: 1) MLTT Core Types; 2) Run-time Types; 3) Control Structures; 4) Recursion Schemes; 5) Abstract Algebra; 6) Category Theory.

In other articles there will be coverage of the following sections of modules: 1) Process Calculus; 2) Univalent Foundations; 3) Categories with Families; 4) Higher Inductive Types. The main advantage of our mathematical components design was motivated and driven by simplicity and clarity, having complete and readable predicative definitions. The minimized Cubical Syntax refined for our purposes covers most of the run-time library which is a major part of the work.

The library consists of highly decoupled modules. E.g. we showed that we can combine Abstract Algebra objects with Category Theory and created new Categories.

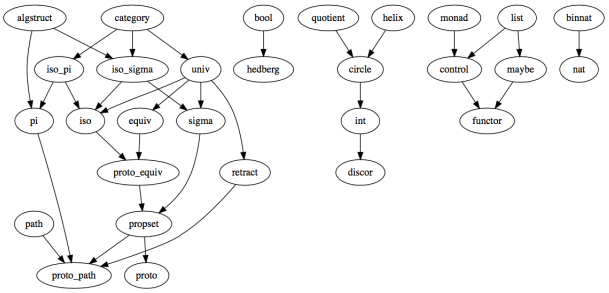

Figure 1: Base library and its dependencies

Table 2: Base Library defined in article

Module	LOC	Module	LOC	Module	LOC
algerbra	360	**fun**	212	**cat**	59
adj	72	**csystem**	508	**hedberg**	28
list	63	**maybe**	19	**mltt**	55
nat	65	**path**	39	**pi**	15
proto	20	**proto_path**	16	**recursion**	98
sip	166	**sigma**	36	**stream**	10
cones	109	**control**	71	**bool**	11

The models are given almost in pure MLTT with the explicit notion of all types used for term definition. In that way, this kind of specification avoids a class of mistakes connected with implicit specifications hidden inside type checker. The library doesn't stop evolving with this report and will evolve with more complex powerful theories remaining comfortable for run-time code extraction to Erlang[14].

References

[1] Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the rezk completion. In Maria del Mar González, Paul C. Yang, Nicola Gambino, and Joachim Kock, editors, *Extended Abstracts Fall 2013*, pages 75–76, Cham, 2015. Springer International Publishing.

[2] Henning Basold and Herman Geuvers. Type theory based on dependent inductive and coinductive types. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 327–336, New York, NY, USA, 2016. ACM.

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

[14] https://erlang.org

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

[5] Nicola Gambino and Martin Hyland. Wellfounded trees and dependent polynomial functors. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, pages 210–225, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[6] Michael Hedberg. A coherence theorem for martin-löf's type theory. *Journal of Functional Programming*, 8(4):413–436, 1998.

[7] Ralf Hinze and Nicolas Wu. Histo- and dynamorphisms revisited. In *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming, WGP 2013, Boston, Massachusetts, USA, September 28, 2013*, pages 1–12, 2013.

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

[9] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.

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

[11] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.

[12] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, Louisiana, USA, March 29 - April 1, 1989, Proceedings*, pages 209–228, 1989.

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

[14] V. Vene. *Categorical Programming with Inductive and Coinductive Types*. Dissertationes mathematicae Universitatis Tartuensis. Tartu University Press, 2000.

[15] Vladimir Voevodsky. A c-system defined by a universe in a category. 2014.

[16] C.A. Weibel. *The K-book: An Introduction to Algebraic K-theory*. Graduate studies in mathematics. American Mathematical Society, 2013.