Impredicative Encoding of Inductive Types in Cubical Type Theory

Maksym Sokhatskyi [1] and Pavlo Maslianko [1]

[1] National Technical University of Ukraine
Igor Sikorsky Kyiv Polytechnical Institute
May 6, 2018

Abstract

Impredicative Encoding of Inductive Types in HoTT.

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

Contents

1 Intro

2 Martin-Lof Type Theory

2.1 Universes

$$\frac{i : Nat}{Type_i} \quad \text{(sorts)}$$

$$\frac{i : Nat}{Type_i : Type_{i+1}} \quad \text{(axioms)}$$

$$\frac{i : Nat, \quad j : Nat}{Type_i \to Type_j : Type_{max(i,j)}} \quad \text{(rules)}$$

2.2 Dependent Types

$$\frac{\pi_1 : A \quad u : A \vdash \pi_2 : B}{[\pi_1/u]\, \pi_2 : B} \quad \text{(subst)}$$

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

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

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

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

2.3 Identity Types

$$\frac{a : A \quad b : A \quad A : Type}{Id(A, a, b) : Type} \quad (Id\text{-formation})$$

$$\frac{a : A}{refl(A, a) : Id(A, a, a)} \quad (Id\text{-intro})$$

$$\frac{p : Id(a, b) \quad x, y : A \quad u : Id(x, y) \vdash E : Type \quad x : A \vdash d : E\ [x/y,\ refl(x)/u]}{J(a, b, p, (x, y, u)\ d) : E\ [a/x,\ b/y,\ p/u]} \quad (J\text{-elimination})$$

$$\frac{a, x, y : A, \quad u : Id(x, y) \vdash E : Type \quad x : A \vdash d : E\ [x/y,\ refl(x)/u]}{J(a, a, refl(a), (x, y, u)\ d) = d\ [a/x] : E\ [a/y,\ refl(a)/u]} \quad (Id\text{-computation})$$

2.4 n-Types

A type A is contractible, or a singleton, if there is $a : A$, called the center of contraction, such that $a = x$ for all $x : A$: A type A is proposition if any x,y: A are equals. A type is a Set if all equalities in A form a prop. It is defined as recursive definition.

$$isContr = \sum_{a:A} \prod_{x:A} a =_A x, \quad isProp(A) = \prod_{x,y:A} x =_A y, \quad isSet = \prod_{x,y:A} isProp\ (x =_A y),$$

$$isGroupoid = \prod_{x,y:A} isSet\ (x =_A y), \quad PROP = \sum_{X:U} isProp(X), \quad SET = \sum_{X:U} isSet(X), ...$$

The following types are inhabited: isSet PROP, isGroupoid SET. All these functions are defined in **path** module. As you can see from definition there is a recurrent pattern which we encode in cubical syntax as follows:

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

isContr (A: U): U = (x:A) * ((y: A) -> Equ 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))
PROP      : U = (X:U) * isProp X
SET       : U = (X:U) * isSet  X
GRPOUPOID : U = (X:U) * isGroupoid X
```

2.5 Path Eliminators

The very basic ground of type checker is heterogeneous equality *PathP* and contructive implementation of reflection rule as lambda over interval $[0, 1]$ that return constant value a on all domain.

```
Path (A:U)(a b:A):U = PathP (<i>A) a b
HeteroEqu (A B:U)(a:A)(b:B)(P:Equ U A B):U = Path P a b
refl (A:U)(a:A):Path A a a = <i> a
sym (A:U)(a b:A) (p: Path A a b): Path A b a = <i> p @ -i
transitivity (A: U)(a b c:A)(p: Path A a b) (q: Path A b c):
    Path A a c = comp (<i> Path A a (q @ i)) p []
```

$$trans : \prod_{p:A=_U B}^{A,B:U} \prod_{a:A} B, \quad singleton : \prod_{x:A}^{A:U} \sum_{y:A} x =_A y$$

$$subst : \prod_{a,b:A}^{A:U,B:A\to U} \prod_{p:a=_A b} \prod_{e:B(a)} B(b), \quad congruence : \prod_{f:A\to B}^{A,B:U} \prod_{a,b:A} \prod_{p:a=_A b} f(a) =_B f(b)$$

Transport tranfers the element of type to another by given path equality of the types. Substitution is like transport but for dependent functions values: by given dependent function and path equality of points in the function domain we can replace the value of dependent function in one point with value in the second point. Congruence states that for a given function and for any two points in the function domain, that are connected, we can state that function values in that points are equal.

```
singl (A:U) (a:A): U = (x: A) * Path A a x
trans (A B:U) (p: Path U A B) (a: A): B = comp p a []
congruence (A B: U) (f:A->B) (a b: A)
           (p: Path A a b): Path B (f a) (f b)
           = <i> f (p @ i)

subst (A:U) (P:A->U) (a b: A)
      (p: Path A a b) (e: P a): P b
      = trans (P a) (P b) (congruence A U P a b p) e
```

These function are defined in **proto_path** module, and all of them except singleton definition are underivable in MLTT.

3 Encoding

3.1 Church Encoding

You know Church encoding which also has its dependent alanolgue in CoC, however in Coq it is imposible to detive Inductive Principle as type system lacks fixpoint and functional extensionality. The example of working compiler of PTS languages are Om and Morte. Assume we have Church encoded NAT:

```
nat = (X:U) -> (X -> X) -> X -> X
```

where first parameter ($X->X$) is a *succ*, the second parameter X is *zero*, and the result of encoding is landed in X. Even if we encode the parameter

```
list (A: U) = (X:U) -> X -> (A -> X) -> X
```

and paremeter A let's say live in 42 universe and X live in 2 universe, then by the signature of encoding the term will be landed in X, thus 2 universe. In other words such dependency is called impredicative displaying that landed term is not a predicate over parameters. This means that Church encoding is incompatible with predicative type checkers with predicative of predicative-cumulative hierarchies.

3.2 Impredicative Encoding

In HoTT n-types is encoded as n-groupoids, thus we need to add a predicate in which n-type we would like to land the encoding:

```
NAT (A: U) = (X:U) -> isSet X -> X -> (A -> X) -> X
```

Here we added isSet predicate. With this motto we can implement propositional truncation by landing term in isProp or even HIT by langing in isGroupoid:

```
TRUN (A:U) type = (X: U) -> isProp X -> (A -> X) -> X
S1 = (X:U) -> isGroupoid X -> ((x:X) -> Path X x x) -> X
MONOPLE (A:U) = (X:U) -> isSet X -> (A -> X) -> X
NAT = (X:U) -> isSet X -> X -> (A -> X) -> X
```

The main publication on this topic could be found at [1] and [2].

4 The Unit Example

Here we have the implementation of Unit impredicative encoding in HoTT.

```
upPath      (X Y:U)(f:X->Y)(a:X->X): X -> Y = o X X Y f a
downPath    (X Y:U)(f:X->Y)(b:Y->Y): X -> Y = o X Y Y b f
naturality  (X Y:U)(f:X->Y)(a:X->X)(b:Y->Y): U
  = Path (X->Y)(upPath X Y f a)(downPath X Y f b)

unitEnc': U = (X: U) -> isSet X -> X -> X
isUnitEnc (one: unitEnc'): U
  = (X Y:U)(x:isSet X)(y:isSet Y)(f:X->Y) ->
    naturality X Y f (one X x)(one Y y)

unitEnc: U = (x: unitEnc') * isUnitEnc x
unitEncStar: unitEnc = (\(X:U)(_:isSet X) ->
  idfun X,\(X Y: U)(_:isSet X)(_:isSet Y)->refl(X->Y))
unitEncRec  (C: U) (s: isSet C) (c: C): unitEnc -> C
  = \(z: unitEnc) -> z.1 C s c
unitEncBeta (C: U) (s: isSet C) (c: C)
  : Path C (unitEncRec C s c unitEncStar) c = refl C c
unitEncEta (z: unitEnc): Path unitEnc unitEncStar z = undefined
unitEncInd (P: unitEnc -> U) (a: unitEnc): P unitEncStar -> P a
  = subst unitEnc P unitEncStar a (unitEncEta a)
unitEncCondition (n: unitEnc'): isProp (isUnitEnc n)
  =  \ (f g: isUnitEnc n) ->
     <h> \ (x y: U) -> \ (X: isSet x) -> \ (Y: isSet y)
  -> \ (F: x -> y) -> <i> \ (R: x) -> Y (F (n x X R)) (n y Y (F R))
       (<j> f x y X Y F @ j R) (<j> g x y X Y F @ j R) @ h @ i
```

References

1. Steve Awodey. Impredicative encodings in hott. 2017.

2. Sam Speight. Impredicative encoding of inductive types in hott. 2017.