Issue II: Inductive Types

Maksym Sokhatskyi [1]

[1] National Technical University of Ukraine
Igor Sikorsky Kyiv Polytechnical Institute

April 26, 2025

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 Inductive Types

1.1 W

Well-founded trees without mutual recursion represented as W-types.

Definition 1. (W-Formation). For $A : \mathcal{U}$ and $B : A \to \mathcal{U}$, type W is defined as $\mathrm{W}(A, B) : \mathcal{U}$ or

$$\mathrm{W}_{(x:A)} B(x) : \mathcal{U}.$$

```
def W' (A : U) (B : A →U) : U := W (x : A), B x
```

Definition 2. (W-Introduction). Elements of $\mathrm{W}_{(x:A)} B(x)$ are called well-founded trees and created with single sup constructor:

$$\sup : \mathrm{W}_{(x:A)} B(x).$$

```
def sup$'$ (A: U) (B: A → U) (x: A) (f: B x → W' A B)
  : W' A B
 := sup A B x f
```

Theorem 1. (Induction Principle $\mathrm{ind}_{\mathrm{W}}$). The induction principle states that for any types $A : \mathcal{U}$ and $B : A \to \mathcal{U}$ and type family C over $\mathrm{W}(A, B)$ and the function $g : G$, where

$$G = \prod_{x:A} \prod_{f:B(x)ßW(A,B)} \prod_{b:B(x)} C(f(b))ßC(\sup(x, f))$$

there is a dependent function:

$$\mathrm{ind}_{\mathrm{W}} : \prod_{C:\mathrm{W}(A,B)ß\mathcal{U}} \prod_{g:G} \prod_{a:A} \prod_{f:B(a)ß\mathrm{W}(A,B)} \prod_{b:B(a)} C(f(b)).$$

```
def W-ind (A : U) (B : A → U)
      (C : (W (x : A), B x) → U)
      (g : Π (x : A) (f : B x → (W (x : A), B x)),
             (Π (b : B x), C (f b)) → C (sup A B x f))
      (a : A) (f : B a → (W (x : A), B x)) (b : B a)
   : C (f b) := ind^W A B C g (f b)
```

Theorem 2. ($\mathrm{ind}_{\mathrm{W}}$ Computes). The induction principle $\mathrm{ind}^{\mathrm{W}}$ satisfies the equation:

$$\begin{aligned} \mathrm{ind}_{\mathrm{W}}\text{-}\beta : g(a, f, \lambda b.\mathrm{ind}^{\mathrm{W}}(g, f(b))) \\ =_{def} \mathrm{ind}_{\mathrm{W}}(g, \sup(a, f)). \end{aligned}$$

```
def ind^W-β (A : U) (B : A → U)
      (C : (W (x : A), B x) → U) (g : Π (x : A)
      (f : B x → (W (x : A), B x)), (Π (b : B x), C (f b)) → C (sup A B x f))
      (a : A) (f : B a → (W (x : A), B x))
    : PathP (⟨_⟩ C (sup A B a f))
       (ind^W A B C g (sup A B a f))
       (g a f (λ (b : B a), ind^W A B C g (f b)))
  := ⟨_⟩ g a f (λ (b : B a), ind^W A B C g (f b))
```

1.2 Empty

The Empty type represents False-type logical $\mathbf{0}$, type without inhabitants, void or $\perp$ (Bottom). As it has not inhabitants it lacks both constructors and eliminators, however, it has induction.

Definition 3. (Formation). Empty-type is defined as built-in $\mathbf{0}$-type:

$$\mathbf{0} : \mathcal{U}.$$

Theorem 3. (Induction Principle ind_0). $\mathbf{0}$-type is satisfying the induction principle:

$$\text{ind}_0 : \prod_{C\,:\,\mathbf{0}\,\to\,\mathcal{U}} \prod_{z\,:\,\mathbf{0}} C(z).$$

```
def Empty-ind (C: 0 → U) (z: 0) : C z := ind₀ (C z) z
```

Definition 4. (Negation or isEmpty). For any type A negation of A is defined as arrow from A to $\mathbf{0}$:

$$\neg A := A \to \mathbf{0}.$$

```
def isEmpty (A: U): U := A → 0
```

The witness of $\neg A$ is obtained by assuming A and deriving a contradiction. This techniques is called proof of negation and is applicable to any types in constrast to proof by contradiction which implies $\neg\neg A \to A$ (double negation elimination) and is applicable only to decidable types with $\neg A + A$ property.

1.3 Unit

Unit type is the simplest type equipped with full set of MLTT inference rules. It contains single inhabitant $\star$ (star).

1.4 Bool

1.5 Maybe

1.6 Either

1.7 Nat

1.8 List

2 Inductive Encodings

2.1 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.

2.2 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 [?] and [?].

2.3 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
```