On commutativity, total orders, and sorting

Wind Wong [1] Vikraman Choudhury [2] Simon J. Gay [1]

[1]University of Glasgow

[2]Università di Bologna and OLAS Team, INRIA

February 14, 2024

Motivation

- The goal is to study free monoids and free commutative monoids.
- We created a framework to formalize different algebraic structures, free algebras and their universal properties.
- Univalent type theory gives us higher inductive types, which allows us to reason with commutativity and equations of algebras. (No setoid hell!)
- Using the framework, we study the relationship between sorting and total orders.

Homotopy Type Theory

Homotopy Type Theory extends intensional MLTT and allows us to reason with equivalences more powerfully.

- Function extensionality ($\forall x.\, f(x) = g(x) \to f = g$)
- Quotient types (via higher inductive types)
- Mere propositions
- Equalities between types (via univalence)

Homotopy Type Theory

Function extensionality ($\forall x.\, f(x) = g(x) \to f = g$)

- MLTT by itself does not have function extensionality
- It has to be added as an axiom (we lose canonicity!)
- funExt can be derived as a theorem in HoTT

Homotopy Type Theory

Quotient types

- In MLTT we can only emulate quotient types with setoids
- We need to prove functions are setoid homomorphisms when defining a function
- A lot of proof obligation
- HoTT lets us define quotient types directly with HITs (no more setoid hell!)

Homotopy Type Theory

Mere propositions

- In MLTT we don't have a distinction between sets and propositions (both are types)
- We might end up needing a stronger theorem to prove a proposition
- E.g. existential proofs are done with Σ-types, requiring us to construct the element
- HoTT allows us to have types that are "mere propositions"
- E.g. existential proofs can be done with propositionally truncated Σ-types (mere existence)
- We can use mere existential proofs to prove other propositions, even if we don't have the specific element

Homotopy Type Theory

Equalities between types

- In MLTT we don't have equalities between types
- HoTT gives us equalities between types by the univalence axiom
- E.g. given $A, B : \mathcal{U}, P : \mathcal{U} \to \mathcal{U}, A = B$, we can get $P(B)$ from $P(A)$ by substitution

Definition

Given types A and B, A is equivalent to B ($A \simeq B$) if there exists an equivalence $A \to B$. A function f is said to be an equivalence if $\left(\sum_{g:B\to A}(f \circ g \sim \mathrm{id}_B)\right) \times \left(\sum_{g:B\to A}(g \circ f \sim \mathrm{id}_A)\right)$.

Univalence axiom

$(A = B) \simeq (A \simeq B)$

Higher Inductive Types

$$\mathrm{isContr}(A) :\equiv \sum_{(a:A)} \prod_{(x:A)} (a = x). \qquad (\text{e.g. } \mathbf{1})$$

$$\mathrm{isProp}(A) :\equiv \prod_{(x,y:A)} (x = y). \qquad (\text{e.g. } \mathbf{1}, \mathbf{0})$$

$$\mathrm{isSet}(A) :\equiv \prod_{(x,y:A)} \mathrm{isProp}(x = y). \qquad (\text{e.g. } \mathbf{1}, \mathbf{0}, \mathbb{N}, \mathrm{hProp})$$

$$\mathrm{isGroupoid}(A) :\equiv \prod_{(x,y:A)} \mathrm{isSet}(x = y). \qquad (\text{e.g. } \mathrm{hSet})$$

$$\mathrm{is2Groupoid}(A) :\equiv \prod_{(x,y:A)} \mathrm{isGroupoid}(x = y). \qquad (\text{e.g. } \mathrm{hGroupoid})$$

$$\vdots$$

Higher Inductive Types

```
data List (A : Type) : Type where
  [] : List A
  _::_ : A → List A → List A

-- Swap list as HIT
data FMSet (A : Type) : Type where
  []    : FMSet A
  _::_  : (x : A) → (xs : FMSet A) → FMSet A
  comm  : ∀ x y xs → x :: y :: xs ≡ y :: x :: xs
  trunc : isSet (FMSet A)
```

Higher Inductive Types

```
-- Swap list as HIT
data FMSet (A : Type) : Type where
  []    : FMSet A
  _::_  : (x : A) → (xs : FMSet A) → FMSet A
  comm  : ∀ x y xs → x :: y :: xs ≡ y :: x :: xs
  trunc : isSet (FMSet A)
  -- alternatively
  -- trunc : (x y : FMSet A) → (p q : x ≡ y) → p ≡ q

_++_ : ∀ (xs ys : FMSet A) → FMSet A
[] ++ ys = ys
(x :: xs) ++ ys = x :: xs ++ ys
comm x y xs i ++ ys =
  -- proof x :: y :: (xs ++ ys) ≡ y :: x :: (xs ++ ys)
trunc xs zs p q i j ++ ys =
  -- proof cong (_++ ys) p ≡ cong (_++ ys) q
```

Higher Inductive Types

```
-- Set quotient
data _/_ (A : Type) (R : A → A → Type) : Type where
  [_] : (a : A) → A / R
  eq/ : (a b : A) → (r : R a b) → [ a ] ≡ [ b ]
  squash/ : (x y : A / R) → (p q : x ≡ y) → p ≡ q

data Perm {A : Type} : List A → List A → Type where
  perm-refl : ∀ {xs} → Perm xs xs
  perm-swap : ∀ {x y xs ys zs}
    → Perm (xs ++ x ∷ y ∷ ys) zs
    → Perm (xs ++ y ∷ x ∷ ys) zs

-- Swap list as quotient
FMSet : Type → Type
FMSet A = List A / Perm
```

Cubical Type Theory

- The project is done in Cubical Agda, an implementation of Cubical Type Theory
- Cubical Type Theory is a variant of HoTT that preserves computational content for proofs
- Univalence is not postulated and can be computationally derived
- Axioms are designed to preserve canonicity, using univalence won't destroy computational content of a proof