Second-Year Annual Report

Li Nuo

July 8, 2013

Contents

Abstract

Quotient is a common notion in many fields, for instance, set theory and topology. In theoretical computer science, specially in Type Theory, given a type and an equivalence relation we would also expect a quotient type. Before we considered the definable quotients which are definable without using quotient types. In this report, we move to other extensions or variation of Type Theory in which the quotient types can be interpreted. We present an implementation of category with families of setoids for setoid model introduced by Altenkirch in [1] and an implementation of weak ω-groupoids for Homotopy Type Theory. The code is written in Agda.

1 Introduction

In the first year, Our work is mainly exploiting the definable quotients within the usual setting of Intensional Type Theory. After that, we conduct research on extending the Intensional Type Theory so that we can interpret quotient types as a type formalisation like product type.

We first work on Thorsten's Setoid model, trying to build categories with families to accommodate the types theory described in his [1] paper so that we could define quotient types following Martin Hofmann's Paper[10]. In this report we present the necessary part for the setoid model.

We also work in Homotopy Type Theory when participating the Univalent Foundation project in the Institute of Advanced Study in Princeton. Homotopy Type Theory is new branch between theoretical computer science and mathematics and it is a variation of Martin-Löf type theory. We have Vladimir Voevodsky's univalent axiom which identifies isomorphic structures. I will not explain this topic in detail here, but a well-written text book on Homotopy Type Theory which is written by many brilliant mathematicians and computer scientists is available now [12]. It is possible to define the quotient types in Homotopy Type Theory but to implement the Homotopy Type Theory in Intensional Type Theory, it is still a difficult problem. We work on defining semi-simplicial sets and weak ω-groupoidsto solve it. In this report we present an syntactic implementation of weak ω-groupoidsfollowing Brunerie's approach. We did some contributions like adapting using heterogeneous equality and syntactic construction of reflexivity.

2 Setoid Model

Quotient types are one of the extensional concepts in Type Theory [8]. To introduce an extensional propositional equality in Intensional Type Theory, Altenkirch [1] proposes a intensional setoid model with a proof-irrelevant universe of propositions **Prop** . It is called a setoid model since types are interpreted by setoids which contain a set and an equivalence relation for each of them. The solution to introduce the extensional equality is an object type theory defined

inside the setoid model which works as the metatheory. He also proved that the extended type theory generated from the metatheory is decidable and adequate, functional extensionality is inhabited and it permits large elimination (defining a dependent type by recursion). Within this type theory, introduction of quotient types is straightforward.

This model is different to a setoid model as E-category, for instance the one introduced by Hofmann [9] . E-category is a category equipped with an equivalence relation for homsets. To distinguish them, we call this category **E-setoids**. **E-setoids** is not a locally cartesian closed category (LCCC) which means that all morphisms are types and they are cartesian closed. Not all morphisms in our category of setoids give rise to types and it is not an LCCC. An LCCC is a model for category with families but not every category with families has to be an LCCC. However it is still a model for Type Theory just like the groupoid model which is a generalisation of it.

To develop this model of type theory in Agda, we have implemented the categories with families of setoids.

2.1 An implementation of Category with Families in Agda

Following the work in [1], we first define a proof-irrelevant universe of propositions. We name it as **hProp** since **Prop** is a reserved word which can't be used and **hProp** is a notion from Homotopy Type Theory which we will introduce later.

2.2 hProp

A proof-irrelvant universe only contains sets with at most one inhabitant.

```
record hProp : Set₁ where
  constructor hp
  field
    prf : Set
    Uni : {p q : prf} → p ≡ q

open hProp public renaming (prf to <_>)
```

We can extract the proof of any propostion $A : hProp$ by using <> and there is always a proof that all inhabitants of it are the same, in other words, if there is any proof of it, the proof is unique. This is not exactly the same as the *Prop* universe in Altenkirch's approach which is judgemental. It is just a judgement whether a set behaves like a *Proposition*. The *hProp* we define above is propositional since we can extract the proof of uniqueness.

We would like to have some basic propositions $\top$ and $\bot$. To distinguish them with the ones for non-proof irrelevant propositions which are already available

in Agda library, we add a prime to all similar symbols.

```
⊤' : hProp
⊤' = hp ⊤ refl

⊥' : hProp
⊥' = hp ⊥ (λ {p} → ⊥-elim p)
```

We also want the universal and existential quantifier for $hProp$, namely it is closed under Π-types and Σ-types. The universal quantifier of $hProp$ can be axiomitised but we decide to explicitly state that we require the functional extensionality to use this module. The reason is that functional extensionality is actually equivalent to the closure under Π-types.

```
∀' : (A : Set)(P : A → hProp) → hProp
∀' A P = hp ((x : A) → < P x >) (ext (λ x → Uni (P x)))
```

```
Σ' : (P : hProp)(Q : < P > → hProp) → hProp
Σ' P Q = hp (Σ < P > (λ x → < Q x >))
  (λ {p} {q} →
  sig-eq (Uni P) (Uni (Q (proj₁ q))))
```

Implication and conjuction which are independent ones of them follow simply.

```
_⇒_ : (P Q : hProp) → hProp
P ⇒ Q =   ∀' < P > (λ _ → Q)

_∧_ : (P Q : hProp) → hProp
P ∧ Q = Σ' P (λ _ → Q)
```

As long as we have implication and conjuction, more operators on proposition can be defined, for instances negation and logical equivalence.

```
¬ : hProp → hProp
```

```
¬ P = P ⇒ ⊥'

_↔_ : (P Q : hProp) → hProp
P ↔ Q = (P ⇒ Q) ∧ (Q ⇒ P)
```

2.3 Category

To define category of setoids we should define category first.

```
record Category : Set where
  constructor CatC
  field
    obj          : Set

    hom          : obj → obj → Set

    id           : ∀ a
                 → hom a a

    [_⇒_]_∘_     : ∀ a {β} γ
                 → hom β γ
                 → hom a β
                 → hom a γ

    isCategory : IsCategory obj hom id [_⇒_]_∘_
```

isCategory contains all the laws for this structure to be a category, for instance the associativity laws for composition.

2.4 Category of setoids

Then we could define setoids using **hProp**. An equivalence relation has three properties reflexivity, symmetry and transitivity.

```
record ishEquivalence {A : Set}(_≈h_ : A → A → hProp) : Set₁ where
  constructor _,_,_
  field
    refl    : {x : A} → < x ≈h x >
    sym     : {x y : A} → < x ≈h y > → < y ≈h x >
    trans   : {x y z : A} → < x ≈h y > → < y ≈h z > → < x ≈h z >
```

Here we use **hSetoid** as the name because **Setoid** is already used for non-proof-irrelvant setoids in the library. For each setoid, we have a carrier type and an equivalence relation.

```
record hSetoid : Set₁ where
  constructor _,_,_
  infix 4 _≈h_ _≈_
  field
    Carrier : Set
    _≈h_    : Carrier → Carrier → hProp
    isEquiv : ishEquivalence _≈h_
  open   ishEquivalence isEquiv public

  _≈_ : Carrier → Carrier → Set
  a ≈ b = < a ≈h b >

  PI : {x y : Carrier}{B : Set}
     (A : x ≈ y → B){p q : x ≈ y}
  → A p ≡ A q
  PI {x} {y} A {p} {q} with Uni (x ≈h y) {p} {q}
  PI A | PE.refl = PE.refl

  reflexive : _≡_ ⇒' _≈_
  reflexive PE.refl = refl
```

A morphism in this category is a function for the underlying sets which respects the equivalence relation.

```
record _⇉_ (A B : hSetoid) : Set₁ where
  constructor fn:_resp:_
  field
    fn   : | A | → | B |
    resp : {x y : | A |} →
      [ A ] x ≈ y →
      [ B ] fn x ≈ fn y
```

The definitions of identity morphism and composition are straightforward and the categorical laws hold trivially as follows.

```
id' : {Γ : hSetoid} → Γ ⇉ Γ
```

```
id' = record { fn = id; resp = id}

_∘c_ : ∀{Γ Δ Z} → Δ ⇉ Z → Γ ⇉ Δ → Γ ⇉ Z
yz ∘c xy = record
  { fn = [ yz ]fn ∘ [ xy ]fn
  ; resp = [ yz ]resp ∘ [ xy ]resp
  }

id₁ : ∀ Γ Δ (ch : Γ ⇉ Δ) → ch ∘c id' ≡ ch
id₁ _ _ ch = PE.refl

id₂ : ∀ Γ Δ (ch : Γ ⇉ Δ) → id' ∘c ch ≡ ch
id₂ _ _ ch = PE.refl

comp : ∀ Γ {Δ Φ} Ψ
  (f : Γ ⇉ Δ)
  (g : Δ ⇉ Φ)
  (h : Φ ⇉ Ψ)
  → h ∘c g ∘c f ≡ h ∘c (g ∘c f)
comp _ _ f g h = PE.refl
```

Combined all components we obtain the category of setoids.

```
setoid-Cat : Category
setoid-Cat = CatC hSetoid _⇉_ (λ _ → id') (λ _ _ → _∘c_)
  (IsCatC id₁ id₂ comp)
```

This category has a terminal object which is just the unit set with trivial equality. As a terminal object there is precisely one morphism from every object to it.

```
⊤-setoid : hSetoid
⊤-setoid = record {
  Carrier = ⊤;
  _≈h_    = λ _ _ → ⊤';
  isEquiv = record {
    refl    = tt;
    sym     = λ _ → tt;
    trans   = λ _ _ → tt } }

⋆ : {Δ : hSetoid} → Δ ⇉ ⊤-setoid
⋆ = record
```

```
  { fn = λ _ → tt
  ; resp = λ _ → tt }

unique⋆ : {Δ : hSetoid} → (f : Δ ⇉ ⊤-setoid) → f ≡ ⋆
unique⋆ f = PE.refl
```

2.5 Category with families of setoids

A Category with families consists of a base category and a functor [6]. We firstly define the category with families of sets in Agda as a guidance for the one for setoids. We would present the setoid one here since it is relevant.

We would like to show two formalisation of category with families for setoids here. The first one is simple and short but not comprehensive. We have to extract all complicated components from the simple definition. However the second one gives these components one by one so that it more understandable and convenient.

The category with families works as a model for type theory. So we will introduce them from a type theoretical point of view.

The base category is the category for contexts. In the setoid version we interpret a context as a setoid as well.

To define the second component, namely the presheaf functor, it is necessary to construct the target category first. The objects of this category are families of setoids.The index setoids are the semantic types and the indexed families of setoids are terms. The morphisms are component-wise morphisms between setoids. All the categorical laws hold trivially.

```
inxSetoids : Set₁
inxSetoids = Σ[ I : hSetoid ] (| I | → hSetoid)

_⇉setoid_ : inxSetoids → inxSetoids → Set₁
(I , f) ⇉setoid (J , g) =
  Σ[ i-map : I ⇉ J ]
    ((i : | I |) → f i ⇉ g ( [ i-map ]fn i))

Fam-setoid : Category
Fam-setoid = CatC
       inxSetoids
       _⇉setoid_
       (λ _ → id' , (λ _ → id'))
       (λ { _ _ (fty , ftm) (gty , gtm) → fty ∘c gty ,
          (λ i → ftm ([ gty ]fn i) ∘c gtm i)})
       (IsCatC
          (λ α β f → PE.refl)
```

```
  (λ a β f → PE.refl)
  (λ a δ f g h → PE.refl))
```

Since we already specify the category of contexts, we only need the presheaf which is a contravariant functor from the category of contexts to the category we defined above. The definition of category with families of setoids could be as simple as follows.

```
record CWF-setoid : Set₁ where
  field
  T : Functor (Op setoid-Cat) Fam-setoid
```

All details of this definition are hidden including the functor laws. Therefore we will show the details as the second version.

The semantic contexts are setoids and the terminal object is just the empty context.

```
Con = hSetoid

emptyCon = ⊤-setoid

emptysub = ⋆
```

A semantic type has following components. fm is a setoid of all types. $substT$ is the substitution between types within the context. It should be a morphism between setoids so it has to preserve the equivalence relation. We also need to specify the computation rules for substitution.

```
record Ty (Γ : Con) : Set₁ where
  field
  fm     : | Γ | → hSetoid

  substT : {x y : | Γ |} →
    [ Γ ] x ≈ y →
    | fm x | →
    | fm y |
  subst* : ∀{x y : | Γ |}
    (p : [ Γ ] x ≈ y)
    {a b : | fm x |} →
    [ fm x ] a ≈ b →
```

```
    [ fm y ] substT p a ≈ substT p b

  refl*   : ∀(x : | Γ |)
    (a : | fm x |) →
    [ fm x ] substT [ Γ ]refl a ≈ a
  trans* : ∀{x y z : | Γ |}
    (p : [ Γ ] x ≈ y)
    (q : [ Γ ] y ≈ z)
    (a : | fm x |)
    → [ fm z ] substT q (substT p a)
      ≈ substT ([ Γ ]trans p q) a
```

There are some useful corollaries hidden since they are not essential to understand the category with families of setoids.

Then we have to define the substituting in a type given a context morphism and verify it preserves equivalence relation as well. It is first component of the morphism part of the presheaf functor.

```
_[_]T : ∀ {Γ Δ : Con} → Ty Δ → Γ ⇉ Δ → Ty Γ
A [ f ]T
  = record
  { fm     = fm ∘ fn
  ; substT = substT ∘ resp
  ; subst* = subst* ∘ resp
  ; refl*  = λ _ _ → subst-pi'
  ; trans* = λ _ _ _ →
    [ fm (fn _) ]trans (trans* _ _ _) subst-pi
  }
  where
    open Ty A
    open _⇉_ f
```

The semantic terms are simpler. It has two parts, *tm* is just the second part of the functor. It should also preserve the equivalence relation on the elements of contexts.

```
record Tm {Γ : Con}(A : Ty Γ) : Set where
  constructor tm:_resp:_
  field
    tm    : (x : | Γ |) → | [ A ]fm x |
    respt : ∀ {x y : | Γ |} →
```

```
    (p : [ Γ ] x ≈ y) →
    [ [ A ]fm y ] [ A ]subst p (tm x) ≈ tm y
```

Substitution for terms can be defined as

```
_[_]m : ∀ {Γ Δ : Con}{A : Ty Δ} →
  Tm A →
  (f : Γ ⇉ Δ)
  → Tm (A [ f ]T)
_[_]m t f = record
  { tm = [ t ]tm ∘ [ f ]fn
  ; respt = [ t ]respt ∘ [ f ]resp
  }
```

Syntactically we can form a new context by using a context Γ and a type $A : Ty\,\Gamma$. To introduce a term of it, we need a term of the semantic context Γ and a term of semantic type A. It is called context comprehension.

```
_&_ : (Γ : Con) → Ty Γ → Con
Γ & A = record
  { Carrier = Σ[ x : | Γ | ] | fm x |
  ; _≈h_    = λ{(x , a) (y , b) →
                Σ'[ p : x ≈h y ] [ fm y ] substT p a ≈h b}
  ; isEquiv =
  record
  { refl    = refl , (refl* _ _)
  ; sym     = λ {(p , q) → (sym p) ,
    [ fm _ ]trans
    (subst* _ ([ fm _ ]sym q))
    trans-refl }
  ; trans = λ {(p , q) (m , n) →
    trans p m ,
    [ fm _ ]trans
    ([ fm _ ]trans
    ([ fm _ ]sym (trans* _ _ _)) (subst* _ q)) n }
  }
  }
```

There are also some other morphisms come with it. Any morphism from a context Γ to a context $\Delta\&A$ consists of a morphism from Γ to Δ and a

term of type A substituted. In other words, There is an isomorphism between $Hom(\Gamma, \Delta \& A)$ and $\Sigma\gamma : Hom(\Gamma, \Delta)A[\gamma]$.

fst projects the morphism and snd projects the term. Indeed the fst operation provides weakening for types, and the snd projection enables us to interpret variables. $fst\&$ defines a morphism for each type A which is a canonical projection of A. We need to use id' which are identity context morphisms to achieve these.

```
fst : {Γ Δ : Con}(A : Ty Δ) → Γ ⇉ (Δ & A) → Γ ⇉ Δ
fst A f = record
      { fn = proj₁ ∘ [ f ]fn
      ; resp = proj₁ ∘ [ f ]resp
      }

fst& : {Γ : Con}(A : Ty Γ) → Γ & A ⇉ Γ
fst& A = fst A id'

_+T_ : {Γ : Con} → Ty Γ → (A : Ty Γ) → Ty (Γ & A)
B +T A = B [ fst& A ]T

snd : {Γ Δ : Con}(A : Ty Δ) →
  (f : Γ ⇉ (Δ & A))
  → Tm (A [ fst A f ]T)
snd A f = record
      { tm = proj₂ ∘ [ f ]fn
      ; respt = proj₂ ∘ [ f ]resp
      }

v0 : {Γ : Con}(A : Ty Γ) → Tm (A +T A)
v0 A = snd A id'
```

Inversely we could define a pairing operation to combine a context morphism with a term. The η-law for the projection and pairing holds trivially.

```
_,,_ : {Γ Δ : Con}{A : Ty Δ}(f : Γ ⇉ Δ) →
  (Tm (A [ f ]T))
  → Γ ⇉ (Δ & A)
f ,, t = record
      { fn = ⟨ [ f ]fn , [ t ]tm ⟩
      ; resp = ⟨ [ f ]resp , [ t ]respt ⟩
      }

&-eta : {Γ Δ : Con}{A : Ty Δ}(f : Γ ⇉ (Δ & A))
```

```
    → _,,_ {A = A} (fst A f) (snd A f) ≡ f
&-eta f = PE.refl
```

Then a lifting operation could help us define Π-types.

```
lift : {Γ Δ : Con}(f : Γ ⇉ Δ)(A : Ty Δ) → Γ & A [ f ]T ⇉ Δ & A
lift f A = record
  { fn = ⟨ [ f ]fn ∘ proj₁ , proj₂ ⟩
  ; resp = ⟨ [ f ]resp ∘ proj₁ , proj₂ ⟩
  }

lift-eta : {Γ Δ : Con}
      (f : Γ ⇉ Δ)(A : Ty Δ)(x : | Γ |)
      (a : | [ A ]fm ([ f ]fn x) |)
      → [ lift f A ]fn (x , a) ≡ ( [ f ]fn x , a)
lift-eta f A x a = PE.refl
```

One of the most complicated part of this definition is the Π-types. Π-types is also called dependent function types. Semantically it is a function type on the underlying semantic types with a proof that the the functions respect the equivalence relation.

```
Π : {Γ : Con}(A : Ty Γ)(B : Ty (Γ & A)) → Ty Γ
```

It also comes with two necessary operation on the terms of Pi-types, λ-abstraction and application. There are $\beta - \eta$ laws to verfify for them so that we could form an isomorphism with these two operations. however technically it causes stack overflow. We may simplify these definition in the future so that we could verify them in Agda.

```
lam : {Γ : Con}{A : Ty Γ}{B : Ty (Γ & A)} → Tm B → Tm (Π A B)
```

```
app : {Γ : Con}{A : Ty Γ}{B : Ty (Γ & A)} → Tm (Π A B) → Tm B
```

Non-dependent version Π-types namely function types can be defined using Π-types with type weakening. Since the dependence disappears, we can also

define it straightforwardly.

```
_⇒'_ : {Γ : Con}(A B : Ty Γ) → Ty Γ
A ⇒' B = Π A (B +T A)
```

2.6 Examples of types

We also implement some common types within the syntactic structure of this type theory. For example, natural numbers and the simply typed universe. I will only present the natural numbers here.

The semantic of natural numbers is just natural numbers and the equivalence relation is defined recursively with respect to case analysis on natural numbers. Other operations and properties of this type can be proved easily.

```
⟦Nat⟧ : {Γ : Con} → Ty Γ
⟦Nat⟧ = record
  { fm = λ γ → record
    { Carrier = ℕ
    ; _≈h_    = _≈nat_
    ; isEquiv = record
      { refl = λ {n} → reflNat {n}
      ; sym = λ {x} {y} → symNat {x} {y}
      ; trans = λ {x} {y} {z} → transNat {x} {y} {z}
      }
    }
  ; substT = λ _ → id
  ; subst* = λ _ → id
  ; refl* = λ x a → reflNat {a}
  ; trans* = λ p q a → reflNat {a}
  }
```

Zero and successor operator can be defined as follows.

```
⟦0⟧ : {Γ : Con} → Tm {Γ} ⟦Nat⟧
⟦0⟧ = record
  { tm = λ _ → 0
  ; respt = λ p → tt
  }

⟦s⟧ : {Γ : Con} → Tm {Γ}⟦Nat⟧ → Tm {Γ} ⟦Nat⟧
```

```
⟦s⟧ (tm: t resp: respt)
  = record
  { tm = suc ∘ t
  ; respt = respt
  }
```

The equality type is an essential part of a type theory. We could define it by using the equivalence relation from the setoid representation of type A. The equivalence relation is trivial since it is proof-irrelevant.

```
Rel : {Γ : Con} → Ty Γ → Set₁
Rel {Γ} A = Ty (Γ & A & A +T A)

⟦Id⟧ : {Γ : Con}(A : Ty Γ) → Rel A
⟦Id⟧ A
  = record
    { fm = λ {((x , a) , b) →    record
      { Carrier = [ [ A ]fm x ] a ≈ b
      ; _≈h_ = λ x₁ x₂ → ⊤'
      ; isEquiv = record
        { refl = λ {x₁} → tt
        ; sym = λ x₂ → tt
        ; trans = λ x₂ x₃ → tt
        }
      } }
    ; substT = λ {((x , a) , b) x0 →
      [ [ A ]fm _ ]trans
      ([ [ A ]fm _ ]sym a)
      ([ [ A ]fm _ ]trans
      ([ A ]subst* _ x0) b)
      }
    ; subst* = λ p x₁ → tt
    ; refl* = λ x a → tt
    ; trans* = λ p q a → tt }
```

The unique inhabitant $refl$ is defined as

```
cm-refl : {Γ : Con}(A : Ty Γ) → Γ & A ⇉ (Γ & A & A +T A)
cm-refl A = record { fn = λ x' → x' , proj₂ x'
      ; resp = λ x' → x' , proj₂ x' }
```

```
⟦refl⟧⁰ : {Γ : Con}(A : Ty Γ)
  → Tm {Γ & A} (⟦Id⟧ A
  [ cm-refl A ]T)
⟦refl⟧⁰ A = record
      { tm = λ {(x , a) → [ [ A ]fm x ]refl {a} }
      ; respt = λ p → tt
      }

⟦refl⟧ : {Γ : Con}(A : Ty Γ)
  → Tm {Γ} (Π A (⟦Id⟧ A
  [ cm-refl A ]T) )
⟦refl⟧ {Γ} A =   lam {Γ} {A} (⟦refl⟧⁰ A)
```

We have an abstracted $refl$ term as well. Using Π-types we could define the eliminator for Id, but it is more involved.

We have done the basics for category of families of setoids. There are more types can be interpreted in this model so that we could show that it is a valid model for Type Theory. We would like to interpret quotient types in this model by following Hofmann's method in [10] or by ourselves.

2.7 Observational equality

Altenkirch further simplifies the setoid model in [2] by adopting McBride's heterogeneous approach to equality. They identifies values up to observation rather than construction which is called observational equality. It is the propositional equality induced by the Setoid model. In general we have a heterogeneous equality which compares terms of types which are different in construction. It only make sense when we can prove the types are the same. It helps us avoids the heavy use of *subst* which makes formalisation and reasoning involved. We could simplify the setoid model by adapting this approach and the implementation could be easier.

3 An implementation of weak ω-groupoids

It is very interesting to investigate the approach to define quotient types in Homotopy Type Theory which is a variant of Martin-Löf type theory. In Homotopy Type Theory, we reject the principle of uniqueness of identity proofs (UIP) but instead we accept the univalence axiom which says that equality of types is weakly equivalent to weak equivalence. Weak equivalence can been seen as a refinement of isomorphism without UIP [3]. To make it more precise, a weak equivalence between two objects A and B in a 2-category is a morphism $f : A \to B$ which has a corresponding inverse morphism $g : B \to A$, but instead of the proofs of isomorphism $f \circ g = 1_B$ and $g \circ f = 1_A$ we have two 2-cell

isomorphisms $f \circ g \cong 1_B$ and $g \circ f \cong 1_A$. Since the setoid interpretation of types in setoid model, as we mentioned before, relies on UIP, it has to be generalised so that we could formalise it in Intensional Type Theory.

The generalised notion is called Grothendieck ω-groupoids. Grothendieck introduced the notion of ω-groupoids in 1983 in a famous Manuscript *Pursuing Stacks* [7]. Maltsiniotis continued his work and suggested a simplification of the original definition which can be found in [11]. Later Ara also present a slight variation of the simplification of weak ω-groupoids in [4]. Categorically speaking an ω-groupoid is an ω-category in which morphisms on all levels are equivalences. As we know that a set can be seen as a discrete category, a setoid is a category where every morphism is unique between two objects. A groupoid is more generalised, every morphism is isomorphism but the proof of isomorphism is unique, namely the composition of a morphism with its inverse is equal to an identity morphism. Similarly, an n-groupoid is an n-category in which morphisms on all levels are equivalence. ω-groupoids which are also called ∞-groupoids is an infinite version of n-groupoids. To model Type Theory without UIP we also require the equalities to be non-strict, in other words, they are not definitionally equalities. Finally we should use weak ω-groupoids to interpret types and eliminate the univalence axiom.

There are several approaches to formalise weak ω-groupoids in Type Theory. For instance, Altenkirch [3], and Bruneries' notes [5]. We work on an implementation of weak ω-groupoids following Brunerie's approach in Agda. The approach is to specify when a globular set is a weak ω-groupoid by first defining a type theory called $\tau_{\infty-groupoid}$ to describe the internal language of Grothendieck weak ω-groupoids, then interpret it with a globular set and a dependent function. All coherence laws of the weak ω-groupoids should be derivable from the syntax, we will present some basic ones, for example reflexivity. One of the main contribution of our work is to use the heterogeneous equality for terms to overcome some very difficult problems when we used the normal homogeneous one. When introducing our implementation, we omit some complicated but less important programs, namely the proofs of some lemmas or the definitions of some auxiliary functions. it is still possible for the reader who is interested in the details to check the code online, in which there are only some minor differences.

3.1 Syntax

Since the definitions of contexts, types and terms involve each others, we adopt a more liberal way to do mutual definition in Agda which is a feature available since version 2.2.10. Something declared is free to use even it has not been completely defined.

Basic Objects We first declare the syntax of our type theory which is called $\tau_{\infty-groupoid}$ namely the internal language of weak ω-groupoids. The following declarations in order are contexts as sets, types are sets dependent on contexts,

terms and variables are sets dependent on types, Contexts morphisms and the contractible contexts.

```
data Con          : Set
data Ty (Γ : Con) : Set
data Tm           : {Γ : Con}(A : Ty Γ) → Set
data Var          : {Γ : Con}(A : Ty Γ) → Set
data _⇒_          : Con → Con → Set

data isContr      : Con → Set
```

Altenkirch also suggests to use Higher Inductive-Inductive definitions for these sets which he coined as Quotient Inductive-Inductive Types (QIIT), in other words, to given an equivalence relation for each of them as one constructor. However we do not use it here.

It is possible to complete the definition of contexts and types first. Contexts are inductively defined as either an empty context or a context with a type of it. Types are defined as either $*$ which we call it 0-cell, or a morphism between two terms of some type A. If the type A is n-cell then we call the morphism $n+1$-cell.

```
data Con where
  ε   : Con
  _,_ : (Γ : Con)(A : Ty Γ) → Con


data Ty Γ where
  *   : Ty Γ
  _=h_ : {A : Ty Γ}(a b : Tm A) → Ty Γ
```

Heterogeneous Equality for Terms One of the big challenge we encountered at first is the difficulty to formalise and to reason about the equalities of terms. When we used the common identity types which is homogeneous, we had to use *subst* function in Agda to unify the types on both sides of the equation. It created a lot of technical issues that made the encoding too involved to proceed. However we found that the syntactic equality of types of given context which will be introduced later, is decidable which means that it is an h-set. In other words, the equalities of types is unique, so that it is safe to use the JM equality (heterogeneous equality) for terms of different types. The equality is inhabited only when they are definitionally equal.

```
data _≅_ {Γ : Con}{A : Ty Γ}
     : {B : Ty Γ} → Tm A → Tm B → Set where
```

```
refl : (b : Tm A) → b ≅ b
```

Once we have the heterogeneous equality for terms, we could define a proof-irrelevant substitution which we call coercion here since it gives us a term of type A if we have a term of type B and the two types are equal. We can also prove that the coerced term is heterogeneously equal to the original term. Combined these definitions, it is much more convenient to formalise and to reason about term equations.

```
_⟦_⟫ : {Γ : Con}{A B : Ty Γ}(a : Tm B) → A ≡ B → Tm A
a ⟦ refl ⟫ = a

cohOp : {Γ : Con}{A B : Ty Γ}{a : Tm B}(p : A ≡ B)
  → a ⟦ p ⟫ ≅ a
cohOp refl = refl _
```

Substitutions With context morphism, we could define substitutions for types variables and terms. Indeed the composition of contexts can be understood as substitution for context morphisms as well.

```
_[_]T  : {Γ Δ : Con}(A : Ty Δ)    (δ : Γ ⇒ Δ) → Ty Γ
_[_]V  : {Γ Δ : Con}{A : Ty Δ}(a : Var A)(δ : Γ ⇒ Δ) → Tm (A [ δ ]T)
_[_]tm : {Γ Δ : Con}{A : Ty Δ}(a : Tm A) (δ : Γ ⇒ Δ) → Tm (A [ δ ]T)
_◎_    : {Γ Δ Θ : Con}   →   Δ ⇒ Θ → Γ ⇒ Δ → Γ ⇒ Θ
```

Weakening Rules we could freely add types to the contexts of given any type judgments, term judgments or context morphisms. We call these rules weakening rules.

```
_+T_   : {Γ : Con}(A : Ty Γ)   → (B : Ty Γ) → Ty (Γ , B)
_+tm_  : {Γ : Con}{A : Ty Γ}(a : Tm A) → (B : Ty Γ) → Tm (A +T B)
_+S_   : {Γ : Con}{Δ : Con}(δ : Γ ⇒ Δ) → (B : Ty Γ) → (Γ , B) ⇒ Δ
```

To define the variables and terms we have to use the weakening rules. A Term can be either a variable or a J-term. We use the unnamed way to define variables as either the immediate variable at the right most of the context, or some variable in the context which can be found by cancelling the right most variable along with each vS. The J-terms are one of the major part of this syntax, which are primitive terms of the primitive types in contractible contexts which will be introduced later. Since contexts, types, variables and terms are all mutually defined, most of the properties of them have to be proved simultaneously as well.

```
data Var where
  v0 : {Γ : Con}{A : Ty Γ}    → Var (A +T A)
  vS : {Γ : Con}{A B : Ty Γ}(x : Var A) → Var (A +T B)

data Tm where
  var : {Γ : Con}{A : Ty Γ} → Var A → Tm A
  JJ  : {Γ Δ : Con} → isContr Δ → (δ : Γ ⇒ Δ) → (A : Ty Δ)
      → Tm (A [ δ ]T)
```

Another core part of the syntactic framework is contractible contexts. Intuitively speaking, a context is contractible if its geometric realization is contractible to a point. It either contains one variable of the 0-cell $*$ which is the base case, or we can extend a contractible context with a variable of an existing type and an n-cell, namely a morphism, between the new variable and some existing variable.

```
data isContr where
  c*  : isContr (ε , *)
  ext : {Γ : Con}
      → isContr Γ → {A : Ty Γ}(x : Var A)
        → isContr ((Γ , A) , (var (vS x) =h var v0))
```

Context morphisms are defined inductively similar to contexts. A context morphism is a list of terms corresponding to the list of types in the context on the right hand side of this morphism.

```
data _⇒_ where
  •   : {Γ : Con} → Γ ⇒ ε
  _,_ : {Γ Δ : Con}(δ : Γ ⇒ Δ){A : Ty Δ}(a : Tm (A [ δ ]T))
      → Γ ⇒ (Δ , A)
```

Lemmas The following four lemmas state that to substitute a type, a variable, a term, or a context morphism with two context morphisms consecutively, is equivalent to substitute with the composition of substitution.

```
[◎]T : {Γ Δ Θ : Con}
  {ϑ : Δ ⇒ Θ}{δ : Γ ⇒ Δ}{A : Ty Θ}
  → A [ ϑ ◎ δ ]T ≡ (A [ ϑ ]T)[ δ ]T

[◎]v : {Γ Δ Θ : Con}
     (ϑ : Δ ⇒ Θ)(δ : Γ ⇒ Δ)(A : Ty Θ)(x   : Var A)
     → x [ ϑ ◎ δ ]V ≅ (x [ ϑ ]V) [ δ ]tm
```

```
[◎]tm : {Γ Δ Θ : Con}
  (ϑ : Δ ⇒ Θ)(δ : Γ ⇒ Δ)(A : Ty Θ)(a : Tm A)
  →   a [ ϑ ◎ δ ]tm ≅ (a [ ϑ ]tm) [ δ ]tm

◎assoc : {Γ Δ Θ Δ₁ : Con}
  (γ : Θ ⇒ Δ₁)(ϑ : Δ ⇒ Θ)(δ : Γ ⇒ Δ)
  → (γ ◎ ϑ) ◎ δ ≡ γ ◎ (ϑ ◎ δ)
```

Weakening inside substitution is equivalent to weakening outside.

```
[+S]T : {Γ Δ : Con}
  {A : Ty Δ}{δ : Γ ⇒ Δ}
  {B : Ty Γ}
  → A [ δ +S B ]T ≡ (A [ δ ]T) +T B

[+S]tm : {Γ Δ : Con}{A : Ty Δ}
    (a : Tm A){δ : Γ ⇒ Δ}
    {B : Ty Γ}
    → a [ δ +S B ]tm ≅ (a [ δ ]tm) +tm B
```

They are useful to derive some auxiliary functions. The following is one of them which is used a lot in proofs.

```
wk-tm+ : {Γ Δ : Con}
  {A : Ty Δ}{δ : Γ ⇒ Δ}
  (B : Ty Γ)
  → Tm (A [ δ ]T +T B) → Tm (A [ δ +S B ]T)
wk-tm+ B t = t ⟦ [+S]T ⟩
```

We could cancel the last term in the substitution for weakened objects since weakening doesn't introduce new variables in types and terms.

```
+T[,]T : {Γ Δ : Con}
  {A : Ty Δ}{δ : Γ ⇒ Δ}
  {B : Ty Δ}{b : Tm (B [ δ ]T)}
  → (A +T B) [ δ , b ]T ≡ A [ δ ]T

+tm[,]tm : {Γ Δ : Con}{A : Ty Δ}
  (a : Tm A)(δ : Γ ⇒ Δ)(B : Ty Δ)
  (c : Tm (B [ δ ]T))
  → (a +tm B) [ δ , c ]tm ≅ a [ δ ]tm
```

Most of the substitutions are defined as usual, except the one for J-terms. We do substitution in the context morphism part of the J-terms.

```
var x    [ δ ]tm = x [ δ ]V
JJ cΔ γ A [ δ ]tm = JJ cΔ (γ ⊚ δ) A ⟦ sym [⊚]T ⟩
```

3.2 Some Important Derivable Constructions

There are some important notions which are missing but are derivable from the syntax. The groupoid laws on all levels should also be derivable using the J-terms. We will show some of them in this section.

Identity context morphism is not a primitive notion in this framework. To define it, we have to declare all the properties it should hold as an identity morphism. In other words, substitution with identity morphism should keep everything unchanged.

```
IdCm : ∀ Γ → Γ ⇒ Γ

IC-T   : ∀ {Γ : Con}(A : Ty Γ) → A [ IdCm Γ ]T ≡ A
IC-v : ∀{Γ : Con}{A : Ty Γ}(x : Var A) →    x [ IdCm Γ ]V ≅ var x
IC-⊚ : ∀{Γ Δ : Con}(δ : Γ ⇒ Δ) → δ ⊚ IdCm Γ ≡ δ
IC-tm : ∀{Γ : Con}{A : Ty Γ}(a : Tm A) → a [ IdCm Γ ]tm ≅ a
```

One of the most important feature of the contractible contexts is that any type in a contractible context is inhabited. It can be simply proved by using J-term and identity morphism.

```
anyTypeInh : ∀{Γ} → {A : Ty Γ} → isContr Γ → Tm {Γ} A
anyTypeInh {A = A} ctr = JJ ctr (IdCm _)    A ⟦ sym (IC-T _) ⟩
```

To show the syntax framework is a valid internal language of weak ω-groupoids, as a first step, we should produce a reflexivity term for the equality of any type.

We use a context with a type to denote the non-empty context.

```
Con* = Σ Con Ty

preCon : Con* → Con
preCon = proj₁

‖_‖ : Con* → Con
‖_‖ = uncurry _,_

lastTy : (Γ : Con*) → Ty (preCon Γ)
lastTy   = proj₂
```

```
lastTy' : (Γ : Con*) → Ty ‖ Γ ‖
lastTy' (_ ,, A) = A +T A
```

We tried several ways to get the reflexivity terms. One of them is to define the suspension of contexts first and then suspend a term of the base type n times to get the n-cell reflexivity. The encoding of the suspension is finished and will be showed later, however we found that there is a simpler way to do it.

Loop Context Here we are going to use some special contexts which are called *loop context* in this paper. For any type in any context, we could always filter out the unrelated part to get a minimum context for this type which is a loop context.

A loop context is either a singleton context with only one variable of the base set, or it is inductively defined by adding a new variable of the same type as the last variable from a loop context and a morphism between these two variables. We will use a function to show what is a loop context.

```
ΩCon : ℕ → Con*
ΩCon 0 = ε ,, *
ΩCon (suc n) = let (Γ ,, A) = ΩCon n in
   (Γ , A , A +T A) ,, (var (vS v0) =h var v0)
```

A variable of the base type is a 0-cell, and the morphism between any two n-cells is an $n+1$-cell as mentioned before. A level-n loop context is the minimum non-empty context where the last variable is n-cell. We could easily prove such a context is contractible. Intuitively it is a special kind of contractible context where the branching approach is unique as we always create an $n+1$ cell for level-n loop context to get a level-(n+1) loop context. Since a loop context is contractible, all types are inhabited. The approach to get the reflexivity is to use J-term with a corresponding loop context. The idea is easy but there are three difficult steps.

First we need to define a function called *loop*Ω to get the required loop context. *loop*Ω' is the complete version which returns five different things, the previous context, the last type, a context morphism between the input previous context and output previous context, a proof term that substitute the output last type with the context morphism is equal to the input last type and another proof term that it is a contractible context. It is necessary to combine them together, because it will become much more involved if they are defined separately.

```
loopΩ' : (Γ : Con)(A : Ty Γ)
       → Σ[ Ω : Con ] Σ[ ω : Ty Ω ] Σ[ γ : Γ ⇒ Ω ]
   Σ[ prf : ω [ γ ]T ≡ A ] isContr (Ω , ω)
loopΩ' Γ * = ε    ,, * ,, • ,, refl ,, c*
```

```
loopΩ' Γ (_=h_ {A} a b) with loopΩ' Γ A
... | (Γ' ,, A' ,, γ' ,, prf' ,, isc) =
      Γ' , A' , A' +T A' ,,
      (var (vS v0) =h var v0) ,,
      γ' , (a ⟦ prf' ⟫) , wk-tm (b ⟦ prf' ⟫) ,,
      (trans wk-hom (trans wk-hom (cohOp-hom prf'))) ,,
      ext isc v0

loopΩ : Con* → Con*
loopΩ (Γ ,, A) with (loopΩ' Γ A)
... |   (Γ' ,, A' ,, γ' ,, prf' ,, isc) = Γ' ,, A'
```

The second problem is to define the context morphism, namely the substitution between the orignal context and the corresponding loop context. And the third problem is to prove type unification for the J-terms. The auxiliary functions make the proofs look much simpler that it was earlier.

```
Tm-refl :   (ne : Con*) → Tm {‖ ne ‖} (var v0 =h var v0)
Tm-refl (Γ ,, A) with loopΩ' Γ A
... | ΩΓ ,, ΩA ,, γ ,, prf ,, isc =
  JJ isc (γ +S A , wk-tm+ A (var v0 ⟦ wk-T prf ⟫)) (var v0 =h var v0)
  ⟦ sym (trans wk-hom (trans wk-hom+ (hom≡ (cohOp (wk-T prf))
  (cohOp (wk-T prf))))) ⟫
```

The one above is special for the reflexivity of the last variable in a non-empty context. We also define a more general version which is the reflexivity for any term of any type in given context.

```
Tm-refl' : (Γ : Con)(A : Ty Γ)(x : Tm A) → Tm (x =h x)
Tm-refl' Γ A x =
  (Tm-refl (Γ ,, A) [ (IdCm _) , (x ⟦ IC-T A ⟫) ]tm)
  ⟦ sym (trans wk-hom (hom≡ (cohOp (IC-T A)) (cohOp (IC-T A)))) ⟫
```

We also construct the symmetry for the morphism between the last two variables.

```
Tm-sym : (Γ : Con)(A : Ty Γ)
  → Tm {Γ , A , A +T A} (var (vS v0) =h var v0)
  → Tm {Γ , A , A +T A} (var v0 =h var (vS v0))
Tm-sym Γ A t =
  (t [ (wk-id ,
  (var v0 ⟦ eq1 ⟫)) ,
  (var (vS v0) ⟦ eq2 ⟫) ]tm)
```

```
  ⟦ sym (trans wk-hom (hom≡ (htrans (cohOp +T[,]T)
    (cohOp eq1))
    (cohOp eq2))) ⟫

  where
    wk-id : (Γ , A , A +T A) ⇒ Γ
    wk-id = (IdCm Γ +S A) +S (A +T A)

    eq1 : A [ wk-id ]T ≡ (A +T A) +T (A +T A)
    eq1 = wkComm (wkComm (IC-T _))

    eq2 : (A +T A) [ wk-id , (var v0 ⟦ eq1 ⟫) ]T
       ≡ (A +T A) +T (A +T A)
    eq2 = trans +T[,]T eq1
```

Then the transitivity for three consecutive variables at the last of a context is as follows.

```
transCon : {Γ : Con}(A : Ty Γ) → Con
transCon {Γ} A = (Γ , A , A +T A , (A +T A) +T (A +T A))

Tm-trans : (Γ : Con)(A : Ty Γ)
  → Tm {transCon A} (var (vS (vS v0)) =h var (vS v0))
  → Tm {transCon A} (var (vS v0) =h var v0)
  → Tm {transCon A} (var (vS (vS v0)) =h var v0)
Tm-trans Γ A p q =
  (q [ (( wk-id ,
  var (vS (vS v0)) ⟦ eq1 ⟫) ,
  var (vS (vS v0)) ⟦ eq2 ⟫) ,
  var v0 ⟦ trans +T[,]T eq2 ⟫ ]tm)
  ⟦ sym (trans wk-hom (hom≡ (htrans wk-coh (cohOp eq2))
  (cohOp (trans +T[,]T eq2)))) ⟫

  where
  wk-id : transCon A ⇒ Γ
  wk-id = ((IdCm _ +S A) +S (A +T A)) +S ((A +T A) +T (A +T A))

  wk-ty : Ty (transCon A)
  wk-ty = ((A +T A) +T (A +T A)) +T ((A +T A) +T (A +T A))

  eq1 : A [ wk-id ]T ≡ wk-ty
  eq1 = wkComm (wkComm (wkComm (IC-T _)))

  wk-id2 : transCon A ⇒ (Γ , A)
  wk-id2 = (IdCm _ +S (A +T A)) +S ((A +T A) +T (A +T A))
```

```
eq2 : (A +T A) [ wk-id , (var (vS (vS v0)) ⟦ eq1 ⟫) ]T ≡ wk-ty
eq2 = trans +T[,]T eq1
```

There are still a lot of coherence laws to prove but it is going to be very sophisticated. We also tried to construct the J-eliminator for equality in this syntactic approach but have not found a solution. To construct more of them, Altenkirch suggests to use a polymorphism theorem which says that given any types, terms and context morphisms, we could replace the base type $*$ in the context of the objects by some type in another context. With this theorem, any lemmas or term inhabited in contractible context should also be inhabited in higher dimensions.

Even though we didn't choose suspension to generate the reflexivity, it should be still useful in the future work.

Like all the other definitions, we have to define a set of operations together. In addition we could also prove that the suspension of a contractible context is still contractible.

```
ΣC : Con → Con
ΣT : {Γ : Con} → Ty Γ → Ty (ΣC Γ)
Σv : {Γ : Con}(A : Ty Γ) → Var A → Var (ΣT A)
Σtm : {Γ : Con}(A : Ty Γ) → Tm A → Tm (ΣT A)
Σs : {Γ Δ : Con} → Γ ⇒ Δ → ΣC Γ ⇒ ΣC Δ
ΣC-Contr : (Δ : Con) → isContr Δ → isContr (ΣC Δ)
```

The suspension of a context is to substitute the base type with the equality of two variables of base type for all occurences. So the base case for a suspension is a context contains two variables of base type. That means we can declare new variables whose type is the equality of these two variables.

```
ΣC ε = ε , * , *
ΣC (Γ , A) = ΣC Γ , ΣT A

*' : {Γ : Con} → Ty (ΣC Γ)
*' {ε} = var (vS v0) =h var v0
*' {Γ , A} = *' {Γ} +T ΣT A

_=h'_ : {Γ : Con}{A : Ty Γ}(a b : Tm A) → Ty (ΣC Γ)
a =h' b = Σtm _ a =h Σtm _ b

ΣT {Γ} * = *' {Γ}
```

```
ΣT (a =h b) = a =h' b
```

There are some lemmas which are neceesary for the definitions. The suspension of terms and context morphisms are too cumbersome to present here.

```
Σ T[+T]   : {Γ : Con}(A : Ty Γ)(B : Ty Γ)
          → ΣT (A +T B) ≡ ΣT A +T ΣT B
Σtm[+tm] : {Γ : Con}{A : Ty Γ}(a : Tm A)(B : Ty Γ)
          → Σtm _ (a +tm B) ≅ Σtm _ a +tm ΣT B
```

3.3 Semantics

Globular Sets To interpret the syntax, we need globular sets. Globular sets are defined coinductively as follows.

```
record Glob : Set₁ where
  constructor _||_
  field
    |_|  : Set
    homo : |_| →  |_| → ∞ Glob
open Glob public
```

Indeed we should assume the 0-level object to be an h-set, namely the equality of any two terms of it should be unique.

As an example, we could contruct the identity globular set called Idw.

```
Idω : (A : Set) → Glob
Idω A = A || (λ a b → ♯ Idω (a ≡ b))
```

Then given a globular set G, we could interpret the objects in syntactic frameworks.

```
⟦_⟧C : Con → Set
⟦_⟧cm : ∀{Γ Δ : Con} → (Γ ⇒ Δ) → ⟦ Γ ⟧C → ⟦ Δ ⟧C
⟦_⟧T : ∀{Γ}(A : Ty Γ)(γ : ⟦ Γ ⟧C) → Glob
⟦_⟧tm : ∀{Γ A}(v : Tm A)(γ :  ⟦ Γ ⟧C) → | ⟦ A ⟧T γ |
```

Another necessary thing is a dependent function Coh [1] should also comes with the globular set. It returns an object for every type in any contractible

[1] it was called J but to make it less ambiguous we renamed it

context, namely what is called a valid coherence in Brunerie's paper. This actually enables us to interpret J-terms in syntax.

```
Coh : (Θ : Con)(ic : isContr Θ)(A : Ty Θ) → (ϑ : ⟦ Θ ⟧C) → | ⟦ A ⟧T ϑ |
```

We temporarily postulate *Coh* function so that we could define the interpretations. However we would adopt the correct way later by defining a record type including the globular set, the interpretations and this function.

There are also some lemmas for weakening to prove as before. The semantic weakening rules tell us how to deal with the weakening inside interpretation.

```
semWK-ty : ∀ {Γ : Con}(A B : Ty Γ)(γ : ⟦ Γ ⟧C)(v : | ⟦ B ⟧T γ |)
  → ⟦ A ⟧T γ ≡ ⟦ A +T B ⟧T (γ , v)

semWK-tm : ∀ {Γ : Con}(A B : Ty Γ)(γ : ⟦ Γ ⟧C)(v : | ⟦ B ⟧T γ |)
  (a : Tm A) → subst |_| (semWK-ty A B γ v) (⟦ a ⟧tm γ)
    ≡ ⟦ a +tm B ⟧tm (γ , v)

semWK-cm : ∀ {Γ Δ : Con}(B : Ty Γ)(γ : ⟦ Γ ⟧C)(v : | ⟦ B ⟧T γ |)
  (δ : Γ ⇒ Δ) → ⟦ δ ⟧cm γ ≡ ⟦ δ +S B ⟧cm (γ , v)
```

4 Summary and Future Work

In this report we first present an implementation of category with families of setoids, which actually serves as a model for Type Theory. We use a proof-irrelevant universe of propositions in the metatheory and formalise this setoid model by defining necessary parts, for instance, contexts, types, terms and some laws. They actually corresponds to the categorical way of defining the category with families of setoids, but it is more readable and more accessible to define them separately. We have define some basic types in this model.

We also show an Agda encoding of weak ω-groupoids following the Brunerie's work. Briefly speaking, we define the syntax of the type theory $\tau_{\infty-groupoid}$, then a weak ω-groupoid is a globular set with the interpretation of the syntax. To overcome some technical problems, we use heterogeneous equality for terms, some auxiliary functions and loop context in all implementation. We construct the identity morphisms and verify some groupoid laws in the syntactic framework. The suspensions for all sorts of objects are also defined for other later constructions.

In the future, for the setoid model, we would like to verify some missing properties and construct some important types. One of the most important task is to extend this model to quotient types. We should also consider how to validate this Setoid model as a model for Type Theory. In addition, this model could also be simplified using the observational equality in Altenkirch's Paper [2]. We should also adapt the judgmental version of *Prop* universe in [1].

There are still a lot of work to do within the syntactic framework of weak ω-groupoids. For instance, we would like to investigate the relation between the $\tau_{\infty-groupoid}$ and a Type Theory with equality types and J eliminator which is called τ_{eq}. One direction is to construct identity types and simulate the J eliminator syntactically in every weak ω-groupoids as we mentioned before, the other direction is to derive J using Coh if we can prove that the τ_{eq} is a weak ω-groupoid. The syntax could be simplified by adopting categories with families. Altenkirch also suggests to use explicit substitution and QIIP which is an alternative way to define the syntax. We would like to formalise a proof of that Idω is an weak ω-groupoids, but the base set in a globular set is an h-set which is incompatible with Idω. Perhaps we could solve the problem by making a syntactic proof. Finally, to model the Type Theory with weak ω-groupoids and to eliminate the univalence axiom would be the most challenging task in the future.

In the last year of my PhD, I will first revise what I have done including the definable quotients, setoid models and the weak ω-groupoids. I will sum them up before the end of this year and submit 2 to 3 papers. One of the objectives is to submit a paper on weak ω-groupoidsto TYPES 2013 in September. To complete the research, I would like to finish quotient types interpretation first, and then do some research relevant to quotient types in Homotopy Type Theory. I will write up my works separately and then organise them into ordered chapters. The estimate time to finish these chapters is next February and I will finish a draft version of my thesis in March. For the next 4 to 5 months, I will refine my thesis and ask some interested researchers to proofread it. The expected time to submit my final version of my PhD thesis is in next September.

References

[1] Thorsten Altenkirch. Extensional Equality in Intensional Type Theory. In *14th Symposium on Logic in Computer Science*, pages 412 – 420, 1999.

[2] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68, New York, NY, USA, 2007. ACM.

[3] Thorsten Altenkirch and Ondrej Rypacek. A syntactical Approach to Weak ω-groupoids. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012*, 2012.

[4] D. Ara. On the homotopy theory of Grothendieck ∞-groupoids. *ArXiv e-prints*, June 2012.

[5] Guillaume Brunerie. Syntactic Grothendieck weak ∞-groupoids. http://uf-ias-2012.wikispaces.com/file/view/SyntacticInfinityGroupoidsRawDefinition.pdf, 2013.

[6] Pierre Clairambault. From categories with families to locally cartesian closed categories. *Project Report, ENS Lyon*, 2005.

[7] Alexander Grothendieck. Pursuing Stacks. 1983. Manuscript.

[8] Martin Hofmann. *Extensional concepts in Intensional Type Theory.* PhD thesis, School of Informatics., 1995.

[9] Martin Hofmann. On the interpretation of type theory in locally cartesian closed categories. In *Computer Science Logic*, pages 427–441. Springer, 1995.

[10] Martin Hofmann. A Simple Model for Quotient Types. In *Proceedings of TLCA'95, volume 902 of Lecture Notes in Computer Science*, pages 216–234. Springer, 1995.

[11] G. Maltsiniotis. Grothendieck ∞-groupoids, and still another definition of ∞-categories. *ArXiv e-prints*, September 2010.

[12] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013.