Towards an Internalization of the Groupoid Model of Type Theory

Matthieu Sozeau – Inria Paris & PPS
Joint work with Nicolas Tabareau – Inria Rennes

Deducteam Seminar
April 25th 2014

Internalizing the Groupoid Model

1. Groupoid interpretations of type theory
   - History
   - Groupoids vs setoids
   - Intensional vs extensional metatheory
   - ∞-groupoids
   - Our model

2. The model structures
   - Defining groupoids
   - Functors and natural transformations
   - Homotopy equivalences
   - Rewriting
   - Dependent Product
   - Dependent Sum

3. The model construction and interpretation
   - Contexts, Types
   - Substitutions
   - The translation
   - Identity types
   - Universe

Groupoids are richer (than sets)

Moving away from the identity sets, we can realize a richer model:

Principle	Definition of equality
Proof-irrelevance	Irrelevant equality
Propositional extensionality	Logical equivalence
Functional extensionality	Pointwise equality
Univalence	Isomorphism

History

- The Groupoid Model – Hofmann and Streicher '94, '98
- Univalent Foundations – Voevodsky, Awodey, Warren, ...
- Takeuti-Gandy for Type Theory (Setoids) – Barras & Coquand, 2013

Groupoids vs setoids

The setoid model (at the basis of e.g. OTT), assumes a proof-irrelevant equality:

$$\Sigma A : \texttt{Type}.\Sigma \sim : A \to A \to \texttt{Prop}.$$
$$\Sigma \sim_{equiv} : \forall\ x\ y, \text{Equivalence}\ (x \sim y)\ \text{eq}.$$
$$\Sigma\ irrel : \forall\ x\ y\ (p\ q : x \sim y), p = q \ldots$$

In Hofmann and Streicher: $[\![T]\!] : GPD$. GPD come with a relevant equality:

$$\Sigma A : \texttt{Type}.\Sigma \sim : A \to A \to \texttt{Type}.$$
$$\Sigma \sim_{equiv} : \forall\ x\ y, \text{Equivalence}\ (x \sim y)\ \text{eq} \ldots$$

But morphisms representing identities are still identified up to *propositional* equality ($\text{eq} : \Pi A, A \to A \to \texttt{Prop}$).

Intensional vs extensional metatheory

In Hofmann and Streicher, the model is built in an *extensional* metatheory, so eq is reflective there.

In *intensional* type theory, we rather represent groupoids with an additional notion of (2-)equality.

$$\Sigma A : \mathtt{Type}.\Sigma \sim_1 : A \to A \to \mathtt{Type}.$$
$$\Sigma \sim_2 : \forall\ \{x\ y\}\ (p\ q : x \sim_1 y), \mathtt{Type}.$$
$$\Sigma \sim_{2-equiv} : \forall\ x\ y, \mathsf{Equivalence}\ (x \sim_1 y)\ \ \sim_2 \ldots.$$

E.g: $[\![\mathtt{Prop}]\!] := (\mathtt{Prop}, \mathrm{iff}, \mathrm{irrel}, \ldots)$.

∞-Groupoids

Ideally, we'd like to model ∞-groupoids, avoiding the need for truncation. The groupoid case limits us:

- We need functional extensionality.
- We need existing identity sets to express it.

A truely infinite-dimensional extension (e.g, globular sets, operads, cubical sets) would lift these.

Our groupoid model

- Stays agnostic w.r.t. future extension, entirely in categorical language (i.e. Dybjer's CwFs).
- Interprets MLTT with *one* universe, Π, Σ, Id-types (no type polymorphism) into CIC (actually the ECC fragment + Id-types for truncations).
- Validates extensional equality principles + one additional rule:

$$\frac{\Gamma \vdash i : \mathtt{Elt}\ A \equiv \mathtt{Elt}\ B}{\Gamma \vdash \mathtt{equiv}\ i : \mathtt{Id}\ \mathcal{U}\ A\ B}$$

"Isomorphic types are equal"

Internalizing the Groupoid Model

Defining groupoids

Defininging groupoids: relations

Start with computational, proof-relevant relations:

```
Definition HomSet (T : Type) := T → T → Type.
```

Using classes and universe polymorphism, we get notations for 1- and 2-dimensional equalities:

```
Class HomSet₁ T := {eq₁ : HomSet T}.
Infix "∼₁" := eq₁ (at level 80).

Class HomSet₂ {T} (Hom : HomSet T) :=
  {eq₂ : ∀ {x y : T}, HomSet (Hom x y)}.
Infix "∼₂" := eq₂ (at level 80).
```

Given a HomSet, we define type classes: Identity, Inverse (noted f^{-1}) and Composition. An Equivalence packs these.

Pre-categories

In a PreCategory, coherences are given up-to $\sim_2$. Ordinary categories additionally require that $\sim_2$ is trivial.

```
Class PreCategory T := { Hom₁ :> HomSet₁ T; Hom₂ :> HomSet₂ eq₁;
    Id :> Identity eq₁; Comp :> Composition eq₁;
    Equivalence₂ :> ∀ x y, (Equivalence (eq₂ (x:=x) (y:=y)));
    id_R : ∀ x y (f : x ∼₁ y), f ∘ identity x ∼₂ f ;
    id_L : ∀ x y (f : x ∼₁ y), identity y ∘ f ∼₂ f ;
    assoc : ∀ x y z w (f: x ∼₁ y) (g: y ∼₁ z) (h: z ∼₁ w),
            (h ∘ g) ∘ f ∼₂ h ∘ (g ∘ f);
    comp : ∀ x y z (f f': x ∼₁ y) (g g': y ∼₁ z),
           f ∼₂ f' → g ∼₂ g' → g ∘ f ∼₂ g' ∘ f' }.
```

Pre-groupoids

A PreGroupoid is a PreCategory where all 1-Homs are invertible and subject to additional compatibility laws for inverses.

```
Class PreGroupoid T := { C :> PreCategory T ; Inv :> Inverse eq_1 ;
      inv_R : ∀ x y (f: x ~_1 y), f ∘ f^-1 ~_2 identity y ;
      inv_L : ∀ x y (f: x ~_1 y), f^-1 ∘ f ~_2 identity x ;
      inv : ∀ x y (f f': x ~_1 y), f ~_2 f' → f^-1 ~_2 f'^-1}.
```

Groupoids are then pre-groupoids where equality at dimension 2 is irrelevant. This irrelevance is defined using a notion of contractibility expressed with (relevant) identity types.

Contractibility

```
Class Contr (A : Type) := {
  center : A ;
  contr : ∀ y : A, center = y}.
```

By analogy to homotopy type theory, we note IsType_1 the property of being a groupoid.

```
Class IsType_1 T := { G :> PreGroupoid T ;
  is_Trunc_2 : ∀ (x y : T) (e e' : x ∼_1 y)
    (E E' : e ∼_2 e'), Contr (E = E')}.
```

In the same way, we define IsType_0 when equality is irrelevant at dimension 1.

```
Class IsType_0 T := { S :> IsType_1 T ;
  is_Trunc_1 : ∀ (x y : T) (e e' : x ∼_1 y) , Contr (e = e')}.
```

Functors and natural transformations

Functors and natural transformations

Groupoid morphisms are functors:

```
Class Functor {T U : Type₁} (f : [T] → [U]) : Type :=
{ map : ∀ {x y}, x ∼₁ y → f x ∼₁ f y ;
  map_comp : ∀ {x y z} (e:x ∼₁ y) (e':y ∼₁ z),
    map (e' ∘ e) ∼₂ map e' ∘ map e ;
  map_2 : ∀ {x y:[T]} {e e' : x ∼₁ y}, (e ∼₂ e') → map e ∼₂ map e' }.

Definition Fun_Type (T U : Type₁) := {f : [T] → [U] & Functor f}.
```

$T \longrightarrow U$ are functors from T to U and $M \star N$ the application of a function M in the first component of a dependent pair.

Natural transformations

Equivalence between functors is given by natural transformations:

```
Class NaturalTrans T U {f g : T ⟶ U} (α : ∀ t : [T], f ⋆ t ∼₁ g ⋆ t)
  := α_map : ∀ {t t'} (e : t ∼₁ t'), α t' ∘ map f e ∼₂ map g e ∘ α t.

Definition nat_trans T U : HomSet (T ⟶ U)
 := λ f g, {α : ∀ t : [T], f ⋆ t ∼₁ g ⋆ t & NaturalTrans α}.
```

Modifications and the function space groupoid structure

Equality between natural transformations:

```
Definition modification T U (f g : T ⟶ U) : HomSet (f ∼₁ g)
  := λ α β, ∀ t : [T], α ⋆ t ∼₂ β ⋆ t.

Definition _fun T U : Type₁ := (T ⟶ U ; fun_grp T U).
```

$\mathsf{fun}_{\mathsf{grp}}$ is a proof that $\mathrm{nat\_trans}$ and $\mathrm{modification}$ form a groupoid on $T \longrightarrow U$. It requires *functional extensionality* to prove the truncation property.

Homotopy equivalences

Homotopy equivalences

Equivalence between groupoids is given by adjoint equivalences.

```
Class Iso_struct T U (f : [T ⟶ U]) :=
{ adjoint : [U ⟶ T] ;
  section : f ∘ adjoint ∼₂ identity U ;
  retraction : adjoint ∘ f ∼₂ identity T}.
```

This type class defines usual equivalences. We need the triangle identity to get adjoint equivalences (it turns it into an hProp).

```
Class Equiv_struct T U (f : T ⟶ U) :=
{ iso : Iso_struct f;
  triangle : ∀ t, section ⋆ (f ⋆ t) ∼₂ map f (retraction ⋆ t)}.

Definition Equiv A B := {f : A ⟶ B & Equiv_struct f}.
```

Adjoint equivalence equality

Equality of homotopy equivalences is given by equivalence of adjunctions.

```
Class EquivEq {T U} {f g : Equiv T U} (α : [f] ∼₂ [g]) : Type :=
  _eq_section : section f ∼₂ section g ∘ (α ∘ (Equiv_adjoint α)).

Definition Equiv_eq T U (f g : Equiv T U) :=
  {α : nat_trans [f] [g] & EquivEq α}.
```

It is crucial here to be able to express the 2-dimensional equality between groupoids as a particular Type and not directly using the identity type.

Groupoid of types

- We can define the *pre-groupoid* Type_1^1 of groupoids and homotopy equivalences. It does not form a groupoid.
- Setoids (inhabitants of Type_0) do form a groupoid:

  `Definition` $\mathsf{Type}_0^1 : \mathsf{Type}_1 :=$ `(`Type_0 `;` $\mathsf{Equiv}_{\mathsf{Type}_0}$`).`

  $\mathsf{Equiv}_{\mathsf{Type}_0}$ is a proof that Equiv and $\mathrm{Equiv\_eq}$ form a groupoid (using funext again).

Note that Type_1 appears both in the type and in the term (through the proof).

Rewriting

Transport

$$\texttt{Definition}\ \mathrm{transport}\ A\ (F{:}[A \longrightarrow \mathsf{Type}^1_1])\ \{x\ y{:}[A]\}\ (e{:}x \sim_1 y)$$
$$: (F \star x) \longrightarrow (F \star y) := [\texttt{map}\ F\ e].$$

The equational theory is derivables from the groupoid laws, e.g.:

$$\texttt{Definition}\ \mathrm{transport}_{\mathsf{eq}}\ A\ (F{:}[A \longrightarrow \mathsf{Type}^1_1])\ \{x\ y{:}[A]\}$$
$$\{e\ e'{:}x \sim_1 y\}\ (H{:}e \sim_2 e')$$
$$: \mathrm{transport}\ F\ e \sim_1 \mathrm{transport}\ F\ e' := [\mathrm{map}_2\ F\ H].$$

We also use $\mathrm{transport}_{\mathsf{id}}$, $\mathrm{transport}_{\mathsf{comp}}$ and $\mathrm{transport}_{\mathsf{map}}$ for compatibilities with identities, composition and for the functoriality of $\mathrm{transport}$.

Dependent Product

Dependent Product

As for functions, dependent functions are interpreted as functors, but of a dependent kind.

```
Class Functor^Π T (U : [T ⟶ Type^1_1]) (f : ∀ t, [U ⋆ t]) : Type := {
  map^Π : ∀ {x y} (e: x ∼₁ y), transport U e ⋆ (f x) ∼₁ f y ;
  map^Π_id : ∀ x, map^Π (identity x) ∼₂ transport_id U ⋆ (f x);
  map^Π_comp : ∀ x y z (e : x ∼₁ y) (e' : y ∼₁ z),
    map^Π (e' ∘ e) ∼₂ map^Π e' ∘ transport_map U _ (map^Π e) ∘
                          (transport_comp U e e' ⋆ _);
  map^Π_2 : ∀ x y (e e': x ∼₁ y) (H: e ∼₂ e'),
    map^Π e ∼₂ map^Π e' ∘ (transport_eq U H ⋆ (f x))}.

Definition Π_T T (U:[T ⟶ Type^1_1]) :=
  {f : ∀ t, [U ⋆ t] & Functor^Π U f}.
```

Dependent natural transformations

Equality between dependent functors is given by dependent natural transformations and equality at level 2 is given by dependent modifications.

$$\texttt{Class}\ \mathsf{NaturalTrans}^{\Pi}\ T\ (U{:}[T \longrightarrow \mathsf{Type}_1^1])\ \{f\ g{:}\ \Pi_T\ U\}$$
$$(\alpha : \forall\ t,\ f \star t \sim_1 g \star t) :=$$
$$\alpha_{\mathsf{map}^{\Pi}} : \forall\ \{t\ t'\}\ e,\ \alpha\ t' \circ \mathrm{map}^{\Pi}\ f\ e \sim_2 \mathrm{map}^{\Pi}\ g\ e \circ \mathsf{transport}_{\mathsf{map}}\ U\ e\ (\alpha\ t).$$

$$\texttt{Definition}\ \mathsf{nat\_trans}^{\Pi}\ T\ (U{:}[T \longrightarrow \mathsf{Type}_1^1])\ (f\ g{:}\ \Pi_T\ U)$$
$$:= \{\alpha : \forall\ t : [T],\ f \star t \sim_1 g \star t\ \&\ \mathsf{NaturalTrans}^{\Pi}\ \alpha\}.$$

$$\texttt{Definition}\ \mathsf{modification}^{\Pi}\ T\ U\ (f\ g : \Pi_T\ U) : \mathsf{HomSet}\ (f \sim_1 g)$$
$$:= \lambda\ \alpha\ \beta\ ,\ \forall\ t : [T],\ \alpha \star t \sim_2 \beta \star t.$$

The groupoid structure on dependent functors

We note $\Pi\ U$ the dependent product over a family of groupoids U.

- A family of setoids can be seen as a family of groupoids using a lifting that we abusively note $U_{\restriction s}$.
- We prove that the dependent product over a family of setoids is also a setoid.
- We note Π_0 the restriction of Π to families of setoids.

Dependent Sum

Dependent Sum

In the interpretation of Σ types, we pay for the fact that we are missing the 2-dimensional nature of Type_1^1. We must restrict to codomains in Type_0^1.

$$\texttt{Definition}\ \Sigma_{\mathrm{T}}\ T\ (U : [T \longrightarrow \mathsf{Type}_0^1]) := \{t\ :\ [T]\ \&\ [U \star t]\}.$$

$$\texttt{Definition}\ \Sigma_{\mathrm{Eq}}\ T\ (U : [T \longrightarrow \mathsf{Type}_0^1]) : \mathsf{HomSet}\ (\Sigma_{\mathrm{T}}\ U) :=$$
$$\lambda\ m\ n,\ \{P\ :\ [m] \sim_1 [n]\ \&\ \mathrm{transport}\ (\ U_{\restriction s})\ P \star (\pi_2\ m) \sim_1 \pi_2\ n\}.$$

2-equality for dependent sums

The 2-equality between 1-equalities is given by projections and rewriting:

```
Definition Σ_Eq2 T (U : [T ⟶ Type_0^1]) (M N : Σ_T U) :
    HomSet (M ∼_1 N) :=
  λ e e' , {P : [e] ∼_2 [e'] &
    π_2 e ∼_2 π_2 e' ∘ (transport_eq ( U_⌈s) P ⋆ (π_2 M))}.
```

- We define the groupoid $\Sigma\ U$ of dependent sums for any family of setoids.
- When T is a setoid, $\Sigma\ U$ is also a setoid.

Internalizing the Groupoid Model

Contexts, Types

Contexts, judgments

- The context is a groupoid (formed by dependent sums).
- Terms of type A introduced by a sequent $\Gamma \vdash t : A$ are dependent (context) functors from Γ to A that return for each context valuation γ, an object of $A \star \gamma$ respecting equality of contexts.
- The type of terms of A is noted $\mathsf{Tm}\ A := [\Pi\ A]$ (context is implicit).

Dependent types

A *dependent* type $\Gamma, x : A \vdash B$ is interpreted in two equivalent ways:

- As a type $\mathrm{TypDep}\ A := \mathrm{Typ}\ (\Sigma\ A)$ over the dependent sum of Γ and A
- As a type family $\mathrm{TypFam}\ A$ over A (corresponding to a family of sets in constructive mathematics). A type family can be seen as a fibration from B to A.

```
Definition TypFam {Γ : Context} (A: Typ Γ) :=
  [Π (λ γ, (A ⋆ γ) ↾s ⟶ Type⁰₁; TypFamcomp _)].
```

Two views on dependent types

Terms of $\mathrm{TypDep}\ A$ and $\mathrm{TypFam}\ A$ can be related using a dependent closure at the level of types. In the interpretation of typing judgments, this connection will be used to switch between the fibration and the morphism points of view.

```
Definition Λ {Γ: Context} {A : Typ Γ} (B: TypDep A)
  : TypFam A := (λ γ, (λ t, B ⋆ (γ; t) ; _); Λ_comp B).
```

Substitutions

- A substitution is a context morphism $[\Gamma \longrightarrow \Delta]$.
- The weakening substitution of $\Gamma, x : A \vdash$ is given by the first projection.
- A substitution σ can be extended by a term a: $\mathsf{Tm}\ (A \cdot \sigma)$.
- To compose a substitution σ with a dependent type A we need to define a *fresh* notion of composition, noted $A \cdot \sigma$, with the same computational content as functor composition but with different universe constraints, to avoid inconsistency.

```
Definition SubExt {Γ Δ : Context} {A : Typ Δ}
  (σ: [Γ ⟶ Δ]) (a: Tm (A · σ)) : [Γ ⟶ Σ A ] :=
    (λ γ, (σ ⋆ γ; a ⋆ γ) ; SubExt_comp _ _).

Definition substF {T Γ} {A:Typ Γ} (F:TypFam A) (σ:[T ⟶ Γ]) :
  TypFam (A · σ) :=
    ([F ∘ σ] : ∀ t : [T], A · σ_|s ⋆ t ⟶ Type^1_0; substF_comp F σ).
```

We abusively note all the different compositions with $\circ$ in this presentation.

Type families

- F in $\mathrm{TypFam}\ A$ can be partially substituted with an term a in $\mathsf{Tm}\ A$, noted $F\ \{\{a\}\}$, to get its value (a type) at a.
- This is defined as $F\ \{\{a\}\} := (\lambda\ \gamma,\ (F \star \gamma) \star (a \star \gamma)\ ;\ \_)$.
- This notion of substitution in a type family enables to state that Λ defines a type level λ-abstraction.

```
Definition BetaT Δ Γ (A:Typ Γ) (B:TypDep A) (σ:[Δ ⟶ Γ])
  (a:Tm (A · σ)) : Λ B ∘ σ {{a}} ∼₁ B · (SubExt σ a) :=
    (λ _, identity _ ; BetaT_comp _ _ _).
```

The translation

Translation

Ideally:

- $[\![\texttt{Type}]\!] \equiv \_\mathrm{Type}$
- $[\![\texttt{Prop}]\!] \equiv \_\mathrm{Prop}$
- $[\![T \to U]\!] \equiv [\![T]\!] \longrightarrow [\![U]\!]$
- $[\![\forall\ t : T,\ U]\!] \equiv \_\mathrm{Prod}\ [\![(\lambda\ t,\ U\ ;\ \_)]\!]$
- $[\![\lambda\ t : T,\ m]\!] \equiv (\lambda\ t : [\![T]\!] ,\ [\![m]\!]\ ;\ \_)$
- $[\![x : T]\!] \equiv x : [\ [\![T]\!]\ ]$
- $[\![m\ n]\!] \equiv [\![m]\!] \star [\![n]\!]$
- $[\![\Sigma\ t : T,\ U]\!] \equiv \_\mathrm{Sum}\ [\![(\lambda\ t,\ U\ ;\ \_)]\!]$
- $[\![\pi_i\ m]\!] \equiv \pi_i\ [\![m]\!]$

Underscores represent obligations to show functoriality/naturality conditions.

The definitions

```
Definition Var {Γ} (A:Typ Γ) : Tm ↑A := (λ t, π₂ t; Var_comp A).

Definition Prod {Γ} (A:Typ Γ) (F:TypFam A)
  : Typ Γ := (λ s, Π₀ (F ⋆ s); Prod_comp A F).

Definition App {Γ} {A:Typ Γ} {F:TypFam A}
  (c:Tm (Prod F)) (a:Tm A) : Tm (F {{a}}) :=
    (λ s, (c ⋆ s) ⋆ (a ⋆ s); App_comp c a).

Definition Lam {Γ} {A:Typ Γ} {B:TypDep A} (b:Tm B)
  : Tm (Prod (Λ B)) := (λ γ, (λ t, b ⋆ (γ ; t) ; _); Lam_comp b).

Definition Sigma {Γ} (A:Typ Γ) (F:TypFam A)
  : Typ Γ := (λ γ: [Γ], Σ (F ⋆ γ); Sigma_comp A F).

Definition Beta {Γ} {A:Typ Γ} {F:TypDep A} (b:Tm F) (a:Tm A)
  : [Lam b ⋆ a] = [b ∘ SubExtId a] := eq_refl _.
```

Identity types

```
Definition Id {Γ} (A: Typ Γ) (a b : Tm A)
  : Typ Γ := (λ γ, (a ⋆ γ ∼₁ b ⋆ γ ; _); Id_comp A a b).
```

The introduction rule just lifts the identity of the underlying setoid:

```
Definition Refl Γ (A: Typ Γ) (a : Tm A)
  : Tm (Id a a) := (λ γ, identity (a ⋆ γ); Refl_comp _).
```

We can interpret the J eliminator of MLTT on Id using functoriality of P and products (Π_{comp}).

```
Definition J Γ (A:Typ Γ) (a b:Tm A)
    (P:TypFam (Sigma (Λ (Id (a ∘ Sub) (Var A)))))
    (e:Tm (Id a b))
    (p:Tm (P{{Pair ↑ (Refl a)}}))
  : Tm (P{{Pair ↑e}}) :=
      Π_comp (λ γ, (map (P ⋆ γ) (J_Pair e P γ)); J_comp _ _) ⋆ p.
```

The J equality rule holds up to $\sim_2$ in the model.

Universe

To interpret the universe $\mathcal{U}$, we need to define its syntax and interpretation of syntax as setoids altogether, which requires induction-recursion. The notion of equality on $\mathcal{U}$ is isomorphism.

```
Definition ≡ {Γ} (A B: Typ Γ) : Typ Γ :=
    (λ γ, (A ⋆ γ ∼₁ B ⋆ γ ; _); ≡comp A B).

Class iso_struct (Γ: Context) (A B : Typ Γ) (f : Tm (A ⟶𝒰 B)) :=
{ iso_adjoint : Tm (B ⟶𝒰 A) ;
  iso_section : Tm (Prod (Λ (Id (iso_adjoint ⋆𝒰 (f ⋆𝒰 Var A)) (Var A)))) ;
  iso_retract : Tm (Prod (Λ (Id (f ⋆𝒰 (iso_adjoint ⋆𝒰 Var B)) (Var
B)))))}.

Definition iso (Γ: Context) (A B : Typ Γ) :=
    {f : Tm (A ⟶𝒰 B) & iso_struct f}.

Definition Ueq (Γ: Context) (A B : Typ Γ) (e : iso A B) : Tm (A ≡ B).
```

Derived facts

```
Lemma proof_irrelevant (P : [_Prop]) (p q : [P]) : p ∼₁ q.
Proof. exact tt. Qed.

Lemma prop_ext (P Q : [_Prop]) : [P] ↔ [Q] → P ∼₁ Q.
Proof. firstorder. Qed.

Lemma fun_ext A B (f g : [A ⟶ B]) :
  nat_trans f g → f ∼₁ g.
Proof. auto. Qed.
```

(nat_trans is not just functional extensionality)

```
Lemma fun_ext_dep T U (f g : [_Prod (T:=T) U]) :
      Dnat_trans f g → f ∼₁ g.
Proof. auto. Qed.
```

Derived facts

To prove equality on dependent pairs, it is enough to prove equality of the corresponding projections:

```
Lemma sum_ext T F (m n : [_Sum (T:=T) F]) :
  ∀ (P : [m] ∼₁ [n]), eq_rect P (π₂ m) ∼₁ π₂ n → m ∼₁ n.
Proof. intros. ∃ P. auto. Qed.
```

Finally, we have isomorphism implies equality.

```
Lemma isoeq (U V : [_Type]) : (Equiv U V) → U ∼₁ V.
Proof. auto. Qed.
```

Conclusion

- An internalization of the groupoid interpretation in intensional type theory, with universe checking on.
- It is missing some proofs, but we're hopeful it can be completed in a reasonable amount of time using automated rewriting.
- Hopefully abstract enough so that we can extend to the 2-dimensional case and above.