Towards an Internalization of the Groupoid Model of Type Theory

Matthieu Sozeau[1,2] and Nicolas Tabareau[1,3]

[1] πr^2 and Ascola teams, INRIA
[2] Preuves, Programmes et Systèmes (PPS)
[3] Laboratoire d'Informatique de Nantes Altantique (LINA)
firstname.surname@inria.fr

Abstract. Homotopical interpretations of Martin-Löf type theory lead toward an interpretation of equality as a richer, more extensional notion. Extensional or axiomatic presentations of the theory with principles based on such models do not yet fully benefit from the power of dependent type theory, that is its computational character. Reconciling intensional type theory with this richer notion of equality requires to move to higher-dimensional structures where equality reasoning is explicit. In this paper, we follow this idea and develop an internalization of a groupoid interpretation of Martin-Löf type theory with one universe respecting the invariance by isomorphism principle. Our formal development relies crucially on ad-hoc polymorphism to overload notions of equality and on a conservative extension of Coq's universe mechanism with polymorphism.

1 Introduction

A notorious difficulty with intensional type theories like Martin-Löf type theory or the calculus of inductive constructions is the lack of extensionality principles in the core theory, and notably in its notion of propositional equality. This makes the theory both inflexible with respect to the notion of equality one might want to use on a given type and also departs from the traditional equality notion from set theory. Functional extensionality (pointwise equal functions are equal), propositional extensionality (logically equivalent propositions are equal) and proof irrelevance (all proofs of the same proposition are equal) are principles that are valid extensions of type theory and can be internalized in an intensional theory, in the manner of Altenkirch et al. [1]. Another extensionality principle coming from homotopy theory is univalence, whose first instance is the idea that isomorphic types are equal [2,3]. All these principles should be imposable on a type theory like MLTT because, intuitively, no term can distinguish between isomorphic types, pointwise equal functions, logically equivalent propositions or two proofs of the same proposition. This hints at the idea that there ought to exist an internal model of type theory where equality is defined on a type by type basis using the aforementioned principles and a translation that witnesses

that any term of the theory is equipped with proofs that they respect these properties. Formalizing this definitional translation is the goal of this paper.

The central change in the theory is in the treatment of equality. Much interest has been devoted to the study of the identity type of type theory and models thereof, starting with the groupoid model of Hofmann and Streicher [4]. This eventually led to the introduction of homotopy type theory and the study of the ω-groupoid model of type theory with identity types, which validates extensionality principles. This model in turn guides work to redesign type theory itself to profit from its richness, and develop a new theory that internalizes the new principles. Preliminary attempts have been made, notably by Licata and Harper [5] who develop a 2-dimensional version of an hybrid intensional/extensional type theory which integrates functional extensionality and univalence in the definition of equality. Work is underway to make it more intensional, and this starts by making the step to higher dimensions, whether finite (weak n-groupoids) or infinite (weak ω-groupoids) [6]. Our work here concentrates on the internalization in Coq of Hofmann and Streicher's groupoid model where we can have a self-contained definition of the structures involved.

Our first motivation to implement this translation is to explore the interpretation of type theory in groupoids in a completely intensional setting and in the type theoretic language, leaving no space for imprecision on the notions of equality and coherence involved. We also hope to give with this translation a basic exposition of the possible type theoretic implications of the groupoid/homotopy models, bridging a gap in the literature. On the technical side, the definition of the groupoid model actually requires to reason at a 2-dimensional level. This is due to the way we interpret the strictness in the definition of groupoids. Indeed, interpreting strictness by propositional equality seems to fail when it comes to define for instance the groupoid on the function space. Our interpretation of strictness is closer to the idea that a groupoid is a weak ω-groupoid for which all equalities at dimension 2 are the same. That is, we only use identity types to express triviality of higher dimension, not coherences themselves. As the model that we use does not have the uniqueness of identity proof principle, the two ways of formalizing groupoids mentionned above are different. Our presentation requires less properties on identity types, but we still need the axiom of functional extensionality. Also, this indicates that if we scale to ω-groupoids, the presence of identity types in the core type theory will not be necessary anymore and so the core type theory will be axiom free. Thus, this paper can be seen as a proof of concept that it is possible to interpret homotopy type theory into type theory without identity types.

The paper is organized as follows: Section 2 introduces the source type theory of the translation and some features of the proof assistant that are used in the formalization. The formal model includes a formalization of groupoids and associated structures (§3.2-3.5) and a construction of the groupoids interpreting the standard type constructors (§3.6-3.7). Section 4 presents a semi-formal interpretation of the dependent type theory in the groupoid model and gives proofs that our interpretation validates a univalence principle at the level of sets (§4.4).

2 Setting of the translation

2.1 Martin-Löf Type Theory with One Universe

For the purpose of this paper we study a restricted source theory resembling a cut-down version of the core language of the Coq system, with only one `Type` universe (see [7] for an in-depth study of this system). This is basically Martin-Löf Type Theory (without `Type` : `Type`). First we introduce the definitions of the various objects at hand. We start with the term language of a dependent λ-calculus: we have a countable set of variables x, y, z, and the usual typed λ-abstraction $\lambda x : \tau, b$, à la Church, application $t\ u$, the dependent product and sum types $\Pi/\Sigma x : A.B$, and an identity type $\mathtt{Id}_T\ t\ u$. There is a single universe $\mathcal{U}$. For T and U in $\mathcal{U}$, the type of (Set)-isomorphisms $T \equiv U$, is definable directly using the other type constructors (see 4.5).

The typing judgment for this calculus is written $\Gamma \vdash t : T$ (Figure 1) where Γ is a context of declarations $\Gamma ::= \cdot \mid \Gamma, x : \tau$, t and T are terms. If we have a valid derivation of this judgment, we say t has type T in Γ.

Most of the rules are standard. The definitional equality $A = B$ is defined as the congruence on type and term formers compatible with β-reductions for abstractions and projections.

Identity type. The identity type in the source theory is the standard Martin-Löf identity type $\mathtt{Id}_T\ t\ u$, generated from an introduction rule for reflexivity with the usual `J` eliminator and its *propositional* reduction rule. The `J` reduction rule will actually be valid definitionally in the model for *closed* terms.

Type equivalence. The new, proof-relevant type equivalence in the source theory for which we want to give a computational model appears in rule Equiv-Intro, extending the formation rules of identity types on $\mathcal{U}$. Type equivalences are introduced by giving a witness of $A \equiv B$ for some A, B *in the universe* $\mathcal{U}$. The `J` rule for type equivalences witnesses the invariance under isomorphism principle of the source type theory.

Universe. The universe $\mathcal{U}$ is closed under Σ, Π, $\mathbb{0}$, $\mathbb{1}$, $\mathbb{2}$ and `Id` in elements of $\mathcal{U}$, *not* type equivalences. The constructors are circumflexed, e.g. $\hat{\Pi}$ and $\hat{\Sigma}$ are introduced with the rules:

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma, x : \mathtt{Elt}(A) \vdash B : \mathcal{U}}{\Gamma \vdash \hat{\Pi}/\hat{\Sigma} x : A.B : \mathcal{U}}\ \textsc{Univ-}\hat{\Pi}\textsc{, -}\hat{\Sigma}$$

For presentation purposes, we do not detail here the treatment of finite types (see [1] for a standard treatment). The extension to W-types would be straightforward. `Elt` is a `type`-forming map from $\mathcal{U}$ that acts as a homomorphism, e.g. its action on products is:

$$\mathtt{Elt}(\hat{\Pi} x : A.B) = \Pi x : \mathtt{Elt}(A).\mathtt{Elt}(B)$$

$$\textsc{Empty}\ \frac{}{\cdot \vdash} \qquad \textsc{Decl}\ \frac{\Gamma \vdash T\ \texttt{type} \quad x \notin \Gamma}{\Gamma, x : T \vdash} \qquad \textsc{Univ}\ \frac{}{\Gamma \vdash \mathcal{U}\ \texttt{type}} \qquad \textsc{Var}\ \frac{\Gamma \vdash \quad (x : T) \in \Gamma}{\Gamma \vdash x : T}$$

$$\textsc{Prod/Sigma}\ \frac{\Gamma, x : A \vdash B\ \texttt{type}}{\Gamma \vdash \Pi/\Sigma x : A.B\ \texttt{type}} \qquad \textsc{Pair}\ \frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B\{t/x\}}{\Gamma \vdash (t, u)_{x:A.B} : \Sigma x : A.B} \qquad \textsc{Proj1}\ \frac{\Gamma \vdash t : \Sigma x : A.B}{\Gamma \vdash \pi_1\ t : A}$$

$$\textsc{Proj2}\ \frac{\Gamma \vdash t : \Sigma x : A.B}{\Gamma \vdash \pi_2\ t : B\{\pi_1\ t/x\}} \qquad \textsc{Conv}\ \frac{\Gamma \vdash t : A \quad \Gamma \vdash B\ \texttt{type} \quad A = B}{\Gamma \vdash t : B}$$

$$\textsc{Lam}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : \Pi x : A.B} \qquad \textsc{App}\ \frac{\Gamma \vdash t : \Pi x : A.B \quad \Gamma \vdash t' : A}{\Gamma \vdash t\ t' : B\{t'/x\}} \qquad \textsc{Id-Intro}\ \frac{\Gamma \vdash t : T}{\Gamma \vdash \texttt{refl}_T\ t : \texttt{Id}_T\ t\ t}$$

$$\textsc{Id}\ \frac{\Gamma \vdash T\ \texttt{type} \qquad \Gamma \vdash A, B : T}{\Gamma \vdash \texttt{Id}_T\ A\ B\ \texttt{type}} \qquad \textsc{Equiv-Intro}\ \frac{\Gamma \vdash i : \texttt{Elt}(A) \equiv \texttt{Elt}(B)}{\Gamma \vdash \texttt{equiv}\ i : \texttt{Id}_{\mathcal{U}}\ A\ B}$$

$$\textsc{Id-Elim}\ (\mathrm{J})\ \frac{\Gamma \vdash i : \texttt{Id}_T\ t\ u \qquad \Gamma, x : T, e : \texttt{Id}_T\ t\ x \vdash P\ \texttt{type} \qquad \Gamma \vdash p : P\{t/x\}\{\texttt{refl}_T\ t/e\}}{\Gamma \vdash \texttt{J}_{\lambda x\ e.P}\ i\ p : P\{u/x, i/e\}}$$

Fig. 1: Typing judgments for our extended MLTT

2.2 The proof assistant

We use an extension of the Coq proof assistant to formally define our translation. Vanilla features of Coq allow us to define overloaded notations and hierarchies of structures through type classes [8], and to separate definitions and proofs using the Program extension [9], they are both documented in Coq's reference manual [10]. We also use the recent extension to polymorphic universes [11].

Classes and projections. The formalization makes heavy use of type classes and sigma types, both defined internally as parameterized records. We also have modified the representation of record projections, making them primitive to allow a more economical representation, leaving out the parameters of the record type they are applied to. This change, which is justified by bidirectional presentations of type theory, makes typechecking exponentially faster in the case of nested structures (see [12] for a detailed explanation of this phenomenon).

One peculiarity of Coq's class system we use is the ability to nest classes. We use the `A_of_B :> A` notation in a type class definition `Class B` as an abbreviation for defining `A_of_B` as an instance of `A`.

Polymorphic Universes. To typecheck our formalization, we also need a stronger universe system than what vanilla Coq offers. Indeed, if we are to give a uniform (shallow) translation of type theory in type theory, we have to define a translation of the `type` universe (a groupoid) as a term of the calculus and equip type constructors like Π and Σ with $[\![\texttt{type}]\!]$ structures as well. As $[\![\texttt{type}]\!]$ itself contains a `Type`, the following situation occurs when we define the translation of, e.g. sums: we should have $[\![\Sigma\ U\ T\ \texttt{type}]\!] = [\![\Sigma]\!]\ [\![U]\!]\ [\![T]\!] : [\![\texttt{type}]\!]$. To ensure consistency of the interpretations of types inside $[\![U]\!]$, $[\![T]\!]$ and the outer one, they must be at different levels, with the outer one at least as large as the inner ones. A universe polymorphic extension of Coq has been designed to allow such highly generic developments [11]. The design was implemented by the first author and is already in use to check the HoTT library in Coq [13].

3 Formalization of groupoids

This section presents our formalization of groupoids in Coq with universe polymorphism. We first explain our overloaded management of equalities and introduce type classes for groupoids and their associated structures, i.e., functors, natural transformations and homotopy equivalences (§3.2-3.4). Natural transformations give access to a homotopic form of functional extensionality, while homotopy equivalences provide extensionality at the level of 0-types. Polymorphic universes are needed to state that setoids and homotopy equivalences form a groupoid. Homotopic equivalences directly provide access to a rewriting mechanism on types (§3.5). This rewriting is used to extend functors and products to dependent functors and dependent sums (§3.6-3.7).

3.1 Notations

We use the following notations throughout: Sigma type introduction is written $(t\ ;\ p)$ when its predicate/fibration is inferrable from the context, and projections are denoted π_1 and π_2. The bracket notation $[\_]$ is an alias for π_1. The following is directly extracted from Coq .v files using the `coqdoc` tool (source files are available at `https://github.com/mattam82/groupoid`). If you are reading the colored version of the paper, keywords are typeset in red, inductive types and classes in blue, inductive constructors in dark red, and defined constants and lemmas in green.

3.2 Definition of groupoids

We formalize groupoids using type classes. Contrarily to what is done in the setoid translation, the basic notion of a morphism is an inhabitant of a relation in `Type` (i.e., a proof-relevant relation):

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

Given x and y in T, HomSet T x y is the type of morphism from x to y. To manipulate different HomSet's at dimension 1 and 2 more abstractly, we use

ad-hoc polymorphism and introduce type classes HomSet_1 and HomSet_2 with according notations.

```
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 that gives the identity morphism, Inverse which corresponds to the existence of an inverse morphism for every morphism (noted f^{-1}) and Composition which corresponds to morphism composition (noted $g \circ f$). Those three properties are gathered by the type class Equivalence. A PreCategory is defined as a category where coherences are given up-to an equivalence relation denoted by $\sim_2$. Ordinary categories are derived with the additional requirement that higher equalities are trivial, which can be expressed using identity types (see the definition of IsType_1).

We do not put this condition into the basic definition because categories and functors form a pre-category but not a 1-category. Thus, working with pre-categories and pre-groupoids allows to share more structure and is closer to the ω-groupoid model which is itself enriched.

```
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' }.
```

In homotopy type theory, coherences are expressed using identity types, with a further requirement that the internal notion of equality induced by the category (isomorphism between two objects) coincides with its identity type [14]. We do not share this point of view because it requires the univalence axiom, which is precisely what we want to give meaning to. Instead, we restrict the use of identity types to the treatment of contractibility for higher cells. Thus, our notion of groupoids is slightly more general as they are not enriched over sets, but over setoids. Note that the comp law is not present in traditional definition of categories because it is automatically satisfied for the identity type.

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 eq₁ ;
    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.

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

This is a way to require that all higher-cells are trivial. In our setting, we do not have the possibility to say that all 2-cells are related by a 3-cell, and so on. The price to pay will be explicit reasoning on identity types when proving for instance contractibility for the function space. In that case, we need the axiom of functional extensionality. By analogy to homotopy type theory, we note IsType_1 the property of being a groupoid.

```
Class IsType₁ 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 IsType₀ T := { S :> IsType₁ T ;
  is_Trunc_1 : ∀ (x y : T) (e e' : x ∼₁ y) , Contr (e = e')}.
```

We note Type_1 for the types that form a IsType_1. The subscript 1 comes from the fact that groupoids are 1-truncated types in homotopy type theory. In the same way, we note Type_0 for the types that form a IsType_0. We define $T_{\restriction s}$ the lifting of setoids to groupoids.

3.3 Functors and natural transformations

A morphism between two groupoids is a functor, i.e., a function between objects of the groupoids that transports homs and subject to compatibility laws.

```
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 ;
  map₂ : ∀ {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}.
```

We note $T \longrightarrow U$ the type of functors from T to U. Note that we only impose compatibility with the composition as compatibilities with identities and inverse Homs can be deduced from it. We note $M \star N$ the application of a function M in the first component of a dependent pair. Equivalence between functors is given by natural transformations. We insist here that this naturality condition in the definition of functional equality is crucial in a higher setting. It is usually omitted in formalizations of homotopy theory in Coq because there they only consider the 1-groupoid case where the naturality becomes trivial, see for instance [15].

```
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}.
```

In our setting, equality between natural transformations is not expressed using identity types, but using the higher categorical notion of modification.

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

We can now equip the functor space with a groupoid structure. Note here that we (abusively) use the same notation for the functor type and its corresponding groupoid.

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

In the definition above, $\mathsf{fun}_{\mathsf{grp}}$ is a proof that nat_trans and modification form a groupoid on $T \longrightarrow U$. In particular, it makes use of functional extensionality, which says that the canonical proof of $f = g \to \forall\ x,\ f\ x = g\ x$ is an equivalence (in the sense of homotopy type theory).

3.4 Homotopic equivalences

The standard notion of equivalence between groupoids is given by adjoint equivalences, that is a map with an adjoint and two proofs that they form a section (or counit of the adjunction) and a retraction (or unit of the adjunction).

```
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. To get an adjoint equivalence, an additional triangle identity between sections and retractions is required. This allows to eliminate a section against a retraction in proofs. A corresponding triangle identity involving adjoint f can also be expressed, but it can be shown that each condition implies the other.

```
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}.
```

It is well known that any equivalence can be turned into an adjoint equivalence by slighty modifying the section. While available in our formalization, this result should be used with care as it opacifies the underlying notion of homotopy and can harden proofs.

Equality of homotopy equivalences is given by equivalence of adjunctions. Two adjunctions are equivalent if their left adjoint are equivalent and they agree on their sections (up-to the isomorphism). Note that equivalence of the right adjoints and agreement on their retractions can be deduced so they are not part of the definition.

```
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. Indeed, whereas the functional extensionality principle makes the use of the identity type and modification equivalent to treat equality of natural transformations, the same is not possible for homotopy equivalences.

We can define the pre-groupoid Type_1^1 of groupoids and homotopy equivalences. However, groupoids together with homotopy equivalences do not form a

groupoid but rather a 2-groupoid. As we only have a formalization of groupoids, this can not be expressed in our setting. Nevertheless, we can state that setoids (inhabitants of Type_0) form a groupoid.

```
Definition Type₀¹ : Type₁ := (Type₀ ; Equiv_Type₀).
```

In the definition above, $\mathsf{Equiv}_{\mathsf{Type}_0}$ is a proof that Equiv and Equiv_eq form a groupoid. It makes again use of functional extensionality to prove contractibility of higher cells. As Type_1 appears both in the term and in the type, the use of polymorphic universe is crucial here to avoid an inconsistency.

3.5 Rewriting in homotopy type theory

When considering a dependent family F of type $[A \longrightarrow \mathsf{Type}_1^1]$, the map function provides a homotopy equivalence between $F \star x$ and $F \star y$ for any x and y such that $x \sim_1 y$. The underlying map of homotopy equivalence can hence be used to cast any term of type $[F \star x]$ to $[F \star y]$.

```
Definition transport A (F:[A ⟶ Type₁¹]) {x y:[A]} (e:x ~₁ y)
  : (F ⋆ x) ⟶ (F ⋆ y) := [map F e].
```

Using compatibility on map, we can reason on different transport paths. Intuitively, any two transport maps with the same domain and codomain should be the same up to homotopy. As we only consider groupoids, there is only one relevant level of compatibilities, higher compatibilities are trivial. $\mathsf{transport}_{\mathsf{eq}}$ is an example of a derivable equality between two transport maps, when the proofs relating x and y are equal.

```
Definition transport_eq A (F:[A ⟶ Type₁¹]) {x y:[A]} {e e':x ~₁ y} (H:e ~₂ e')
  : transport F e ~₁ transport F e' := [map₂ F H].
```

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

3.6 Dependent Product

As for functions, dependent functions will be interpreted as functors. But this time, the compatibilities with higher-order morphisms cannot be expressed as simple equalities, as some transport has to be done to make those equalities typable. We call such a functor a *dependent functor*. Dependent functors are defined between a groupoid T and a functor U from T to Type_1^1 (the pre-groupoid of groupoids). U must be seen as a type depending on T, or as a family of types indexed by T.

```
Class Functor^Π T (U : [T ⟶ Type₁¹]) (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 ⟶ Type₁¹]) := {f : ∀ t, [U ⋆ t] & Functor^Π U f}.
```

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

```
Class NaturalTrans^Π T (U:[T ⟶ Type^1_1]) {f g: Π_T U} (α : ∀ t, f ⋆ t ∼_1 g ⋆ t)
  := α_map^Π : ∀ {t t'} e, α t' ∘ map^Π f e ∼_2 map^Π g e ∘ transport_map U e (α t).

Definition nat_trans^Π T (U:[T ⟶ Type^1_1]) (f g: Π_T U)
  := {α : ∀ t : [T], f ⋆ t ∼_1 g ⋆ t & NaturalTrans^Π α}.

Definition modification^Π T U (f g : Π_T U) : HomSet (f ∼_1 g)
  := λ α β , ∀ t : [T], α ⋆ t ∼_2 β ⋆ t.
```

We can now equip dependent functors with a groupoid structure as we have done for 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 can 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.

3.7 Dependent sums

In the interpretation of Σ types, we pay for the fact that we are missing the 2-dimensional nature of Type^1_1. Indeed, as we will need rewriting in the definition of equality on Σ types, delivering the corresponding groupoid structure requires to reason on compatibility between rewritings, which amount to the missing 2-dimensional laws. However, as Type^1_0 is a groupoid, all 2-dimensional equalities become trivial on a family of setoids, so we can define the groupoid of Σ types over a groupoid T and a morphism of type $[T \longrightarrow \mathsf{Type}^1_0]$.

```
Definition Σ_T T (U : [T ⟶ Type^1_0]) := {t : [T] & [U ⋆ t]}.
```

The 1-equality between dependent pairs is given by 1-equality on the first and second projections, with a transport on the second projection on the left.

```
Definition Σ_Eq T (U : [T ⟶ Type^1_0]) : HomSet (Σ_T U) :=
  λ m n, {P : [m] ∼_1 [n] & transport (U_⌈s) P ⋆ (π_2 m) ∼_1 π_2 n}.
```

In the same way, 2-equality between 1-equalities is given by projections and rewriting.

```
Definition Σ_Eq_2 T (U : [T ⟶ Type^1_0]) (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))}.
```

This way, we can define the groupoid $\Sigma\ U$ of dependent sums for any family of setoids. When T is a setoid, $\Sigma\ U$ is also a setoid.

4 The groupoid interpretation

We now organize our formalization of groupoids into a model of the dependent type theory introduced in Section 2.1. The interpretation is based on the notion of categories with families introduced by Dybjer [16] later used in [4]. This interpretation can also be seen as an extension of the Takeuti-Gandy interpretation

of simple type theory, recently generalized to dependent type theory by Coquand et al. using Kan semisimplicial sets or cubical sets [17].

There are two main novelties in our interpretation. First, we take advantage of universe polymorphism to interpret dependent types directly as functors into Type^1_0. Second, we provide an interpretation in a model where structures that are definitionally equal for Kan semisimplicial sets are only homotopically equal, which requires more care in the definitions (see for instance the definition of Lam in Section 4.3 which mixes two points of view on fibrations).

We only present the computational part of the interpretation, the proofs of functoriality and naturality are not detailed but are valid in the model. Some of them are available in the Coq development but we have also admitted some lemmas, due to the complete absence of automated rewriting on Type in the current version of Coq. This will be addressed by the first author in a future version, relying on the polymorphic universe extension.

4.1 Dependent types

The judgment context $\Gamma \vdash$ of Section 2.1 is represented in Coq as a groupoid, noted $\mathtt{Context} := \mathsf{Type}_1$. The empty context (Rule Empty) is interpreted as the groupoid with exactly one element at each dimension. Types in a context Γ, noted $\mathsf{Typ}\ \Gamma$, are (context) functors from Γ to the groupoid of setoids Type^1_0. Thus, a judgment $\Gamma \vdash A : \mathtt{Type}$ is represented as a term A of type $\mathsf{Typ}\ \Gamma$. Context extension (Rule Decl) is given by dependent sums, i.e., the judgment $\Gamma, x : A \vdash$ is represented as $\Sigma\ A$.

Terms of A introduced by a sequent $\Gamma \vdash x : A$ 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).

A dependent type $\Gamma, x : A \vdash B$ is interpreted in two equivalent ways: simply as a type $\mathsf{TypDep}\ A := \mathsf{Typ}\ (\Sigma\ A)$ over the dependent sum of Γ and A or as a type family $\mathsf{TypFam}\ A$ over A (corresponding to a family of sets in constructive mathematics). A type family can be seen as a fibration (or bundle) from B to A. In what follows, the suffix $_{\mathsf{comp}}$ is given to proofs of (dependent) functoriality.

```
Definition TypFam {Γ : Context} (A: Typ Γ) :=
  [Π (λ γ,   (A ⋆ γ) ↑s ⟶ Type¹₀; TypFam_comp _)].
```

Terms of $\mathsf{TypDep}\ A$ and $\mathsf{TypFam}\ A$ 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).
```

4.2 Substitutions

A substitution is represented by a context morphism $[\Gamma \longrightarrow \Delta]$. Note that although a substitution σ can be composed with a dependent type A by using composition of functors, we need to define a fresh notion of composition, noted A

$\cdot\,\sigma$, with the same computational content but with different universe constraints, to avoid universe inconsistency: composition otherwise forces all endpoints to be at the same level, which is too restrictive for Type-valued functors and context morphisms.

A substitution σ can be extended by a term a: Tm $(A \cdot \sigma)$ of A : Typ Δ.

```
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 _ _).
```

where $\mathsf{SubExt}_{\mathsf{comp}}$ is a proof that it is functorial.

```
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).
```

A substitution σ can be applied to a type family F using the composition of a functor with a dependent functor. We abusively note all those different compositions with $\circ$ as it is done in mathematics, whereas they are distinct operators in the Coq development. The weakening substitution of $\Gamma, x : A \vdash$ is given by the first projection.

A type family F in TypFam A can be partially substituted with an term a in Tm A, noted F $\{\{a\}\}$, to get its value (a type) at a. This process is defined as F $\{\{a\}\} := (\lambda\,\gamma, (F \star \gamma) \star (a \star \gamma)\ ;\ \_)$ (where $\_$ is a proof it is functorial). Note that this pattern of application *up-to a context* γ will be used later to defined other notions of application. Although the computational definitions are the same, the compatibility conditions are always different. This notion of partial 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 _ _ _).
```

4.3 Interpretation of the typing judgment

The typing rules of Figure 1 are interpreted in the groupoid model as described below.

Var. The rule Var is given by the second projection plus a proof that the projection is dependently functorial. Note the explicit weakening of A in the returned type. This is because we need to make explicit that the context used to type A is extended with an term of type A. The rule of Figure 1 is more general as it performs an implicit weakening. We do not interpret this part of the rule as weakening is explicit in our model.

```
Definition Var {Γ} (A:Typ Γ) : Tm ↑A := (λ t, π_2 t; Var_comp A).
```

Prod. The rule Prod is interpreted using the dependent functor space, plus a proof that equivalent contexts give rise to isomorphic dependent functor spaces. Note that the rule is defined on type families and not on the dependent type formulation because here we need a fibration point of view.

```
Definition Prod {Γ} (A:Typ Γ) (F:TypFam A)
  : Typ Γ := (λ s, Π_0 (F ⋆ s); Prod_comp A F).
```

App. The rule App is interpreted using an up-to context application and a proof of dependent functoriality. We abusively note $M \star N$ the application of App.

```
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).
```

Lam. Term-level λ-abstraction is defined with the same computational meaning as type-level λ-abstraction, but it differs on the proof of dependent functoriality. Note that we use Λ in the definition because we need both the fibration (for Prod) and the morphism (for Tm B) point of view.

```
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).
```

Sigma, Pair *and* Projs. The rules for Σ types are interpreted using the dependent sum Σ on setoids.

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

Pairing and projections are obtained by a context lift of pairing and projection of the underlying dependent sum.

Conv. It is not possible to prove in Coq that the conversion rule is preserved because the application of this rule is implicit and can not be reified. Nevertheless, to witness this preservation, we show that beta conversion is valid as a definitional equality on the first projection. As conversion is only done on types and interpretation of types is always projected, this is enough to guarantee that the conversion rule is also admissible.

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

where SubExtId is a specialization of SubExt with the identity substitution.

4.4 Identity Types

One of the main interest of the groupoid interpretation is that it allows to interpret a type directed notion of equality which validates the J eliminator of identity types but also various extensional principles. For any terms a and b of a dependent type A:Typ Γ, we note Id a b the equality type between a and b obtained by lifting $\sim_1$ to get a type depending on Γ.

```
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 of identity types which corresponds to reflexivity is interpreted by the (lifting of) 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 of product (Π_{comp}). In the definition of J, the predicate P depends on the

proof of equality, which is interpreted using a Sigma type. The functoriality of P is used on the term J_Pair e P γ, which is a proof that (a;Refl a) is equal to (b;e). The notation $\uparrow$ is used to convert the type of terms according to equality on Λ.

```
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.
```

4.5 Universe

To interpret the universe $\mathcal{U}$, we need to define its syntax and interpretation of syntax as setoids altogether. That is, $\mathcal{U}$ requires inductive-recursive definitions to be interpreted. As such definition are not available in Coq, we cannot completely interpret $\mathcal{U}$[4]. Nevertheless, we present a way to interpret the identity type on $\mathcal{U}$ and Rule Id-Equiv-Intro which defines equality of types in $\mathcal{U}$ as isomorphism.

We interpret the identity type on $\mathcal{U}$ in the same way as Id, except that it relates two dependent types A and B instead of terms of a dependent type.

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

To define the notion of isomorphism, we need to define a proper notion of function (noted $A \longrightarrow_{\mathcal{U}} B$) that does not use the restriction of Prod to constant type families. This is because the definition of an isomorphism involves two functions that have to be composed in both ways, which lead to universe inconsistency if we use our asymmetric dependent products to encode these functions. We define the notion of application (noted $g \star_{\mathcal{U}} f$) for this kind of functions.

```
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}.
```

Then, we can show that this definition of isomorphism corresponds to equivalence of setoids. Again, the only extra work is with the management of context lifting. This provides a computational content to the univalence principle restricted to setoids.

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

5 Related Work and Conclusion

We have presented an internalization of the groupoid interpretation of Martin-Löf type theory with one universe respecting the invariance under isomorphism principle in Coq's type theory with universe polymorphism.

[4] The folklore coding trick to encode inductive-recursive definition using an indexed family as done in [1] does not work here because it transforms an inductive-recursive Agda Set into a larger universe which is no longer a setoid.

The groupoid interpretation is due to Hofmann and Streicher [4]. This interpretation is based on the notion of categories with families introduced by Dybjer [16]. This framework has recently been used by Coquand et al. to give an interpretation in semi-simplicial sets and cubical sets [17,18]. Although very promising, the interpretation based on cubical sets has not yet been mechanically checked and only an extraction procedure based on it has been implemented in Haskell.

Altenkirch et al. have introduced Observational Type Theory (OTT) [1], an intentional type theory where functional extensionality is native, but equality in the universe is structural. To prove expected properties on OTT such as strong normalization, decidable typechecking and canonicity, they use embeddings into Agda and extensional type theory. A setoid interpretation clearly guides their design, and our model could be adapted to interpret this theory as well.

We have strived for generality in our definitions and while our interpretation is done for groupoids, it illustrates the main structures of the model and should adapt to higher dimensional models, specifically ω-groupoids. The next step of our work is to generalize the construction to higher dimensions. We already have a formalization of weak 2-groupoids based on inductive definitions (*à la* enriched categories) on the computational structure. The only mechanism that is not inductive is the generation of higher-order compatibilities between coherence maps. This is because category enrichment provides a co-inductive definition for *strict* ω-groupoids only. Formalizing in Coq the recent work of Cheng and Leinster [19] on weak enrichment should provide a way to define *weak* ω-groupoids co-inductively using operads to parameterize the compatibility required on coherence maps at higher levels.

References

1. Altenkirch, T., McBride, C., Swierstra, W.: Observational Equality, Now! In: PLPV'07, Freiburg, Germany (2007)
2. Voevodsky, V. In: Univalent Foundations of Mathematics. Volume 6642. Springer Berlin Heidelberg (2011) 4–4
3. Pelayo, Á., Warren, M.A.: Homotopy type theory and Voevodsky's univalent foundations. (10 2012)
4. Hofmann, M., Streicher, T.: The Groupoid Interpretation of Type Theory. In: Twenty-five years of constructive type theory (Venice, 1995). Volume 36 of Oxford Logic Guides. Oxford Univ. Press, New York (1998) 83–111
5. Licata, D.R., Harper, R.: Canonicity for 2-dimensional type theory. In Field, J., Hicks, M., eds.: POPL, ACM (2012) 337–348
6. Altenkirch, T., Rypacek, O.: A Syntactical Approach to Weak omega-Groupoids. In Cégielski, P., Durand, A., eds.: CSL. Volume 16 of LIPIcs., Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012) 16–30
7. Hoffman, M.: Syntax and Semantics of Dependent Types. In: Semantics and Logics of Computation. (1997) 241–298
8. Sozeau, M., Oury, N.: First-Class Type Classes. In Otmane Ait Mohamed, C.M., Tahar, S., eds.: TPHOLs. Volume 5170 of LNCS., Springer (August 2008) 278–293

9. Sozeau, M.: Program-ing Finger Trees in Coq. In: ICFP'07, Freiburg, Germany, ACM Press (2007) 13–24
10. The Coq development team: Coq 8.4 Reference Manual. Inria. (2012)
11. Sozeau, M., Tabareau, N.: Universe Polymorphism in Coq. Submitted (January 2014)
12. Garillot, F.: Generic Proof Tools and Finite Group Theory. PhD thesis, Ecole Polytechnique X (December 2011)
13. The HoTT Development Team: Homotopy Type Theory in Coq
14. Ahrens, B., Kapulkin, C., Shulman, M.: Univalent categories and the Rezk completion. ArXiv e-prints (March 2013)
15. Bauer, A., LeFanu Lumsdaine, P.: A Coq proof that Univalence Axioms implies Functional Extensionality. (2011)
16. Dybjer, P.: Internal type theory. In Berardi, S., Coppo, M., eds.: Types for Proofs and Programs. Volume 1158 of LNCS. Springer Berlin Heidelberg (1996) 120–134
17. Barras, B., Coquand, T., Huber, S.: A Generalization of Takeuti-Gandy Interpretation. (2013)
18. Bezem, M., Coquand, T., Huber, S.: A Model of Type Theory in Cubical Sets. (December 2013)
19. Cheng, E., Leinster, T.: Weak ω-categories via terminal coalgebras. (2012)