Univalence For Free

Matthieu Sozeau and Nicolas Tabareau

INRIA
firstname.surname@inria.fr

Abstract. We present an internalization of the 2-groupoid interpretation of the calculus of construction that allows to realize the univalence axiom, proof irrelevance and reasoning modulo. As an example, we show that in our setting, the type of Church integers is equal to the inductive type of natural numbers.

1 Introduction

A notorious difficulty with intensional type theories like Martin-Löf Type Theory or the Calculus of Inductive Constructions (CIC) 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 but are not currently internalized in it, except for the efforts of Altenkirch *et al.*[1]. Another extensionality principle coming from homotopy theory is Univalence, which translates to the idea that isomorphic types are equal [15,11]. All these principles should be imposeable on a type theory like CIC because, intuitively, no terms 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 translation is the goal of this paper.

The central change in the theory is in the definition 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 [6]. This eventually led, by a detour to homotopy theory, to the study of the ω-groupoid model of Type Theory with identity types, which validates extensionality principles [10]. 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 [9] 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) [2]. Our work here is more modest in the sense that our model will be truncated at dimension 2 where we can have a self-contained definition of the structures involved. Truncation at that level means that there are equalities and equalities between equalities, and that is all. But all "lower" notions of equality including isomorphism of types will have computational content.

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, even at dimension 2, the structures are already quite tedious to manipulate and we found some interesting conditions on the structure of Π and Σ types that we believe were never presented in this form before. In the development, we strived to be as generic as possible and use the abstract structures of category theory to not be essentially tied to dimension 2. That level of genericity required a universe polymorphic type system that we showcase here as well.

Also, we shed light on the fact that usual Coq rewriting using *eq_rect* is given here by the functoriality maps of a fibration of type $T \rightarrow$ `Type`. And we show how to use the induced rewriting terms to define the notion of dependent functors, that has to be introduced to handle dependent product types. This is to our knowledge the first formulation of dependent functoriality condition in Type Theory.

A second motivation for this translation is to complete a forcing translation of type theory into type theory we developed[7]. By building up forcing layers on top of a core type theory one can introduce new logical operators or type constructors (e.g. modal logic, recursive types), defined by translation. For correctness, this translation relies on a type theory that integrates proof-irrelevance and functional extensionality. The present translation gives us the expressive power we need to compose with the forcing translation and get a fully definitional extension of type theory with forcing.

To summarize, our contributions are:

– A translation from type theory to (weak) 2-groupoids, or an internal model of type theory in 2-groupoids that has been fully checked in a universe polymorphic version of Coq.
– A type-theoretic description of the necessary conditions on the interpretation of function types, including dependent product types which give rise to dependent 2-groupoid functors and dependent sums.
– An example use of this theory showing that the interpretations of Church integers and the inductive definition of natural numbers are *equal*.

The paper is organized as follows: in section 2 we define the setting of the translation and some features of the proof assistant that will be used in the formalization. In section 3 we present the translation, which includes a formalization of weak 2-groupoids (§3.1), the interpretations of the various type constructors at hand (§3.2-3.7) and proofs that our interpretation validates all the extensionality principles we care about (§3.9). We present an example in section 4 and conclude in section 5.

2 The proof assistant

We will 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 [13], and separate definitions from proofs using the Program extension [12], they are both documented in Coq's reference manual [14]. One peculiarity of Coq's class system we use is the ability to nest classes. We use the :> A notation in a type class definition Class B as an abbreviation for defining A as a sub-class of B. This declares the projection as an instance of the subclass, hence to find an A we can lookup a B.

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. If you are reading the colored, hypertextual version of the paper, all definitions are hyperlinked, including the ones refering to Coq's standard library. Red is used for keywords, blue for inductive types and classes, dark red for inductive constructors, and green for defined constants and lemmas.

2.1 Polymorphic Universes

To be able to typecheck our formalization, we need a stronger universe system than what vanilla Coq offers. Indeed if we are to give a uniform translation of type theory in type theory, we will have to form a translation of type universes $[\![\mathtt{Type}]\!]$ and equip type constructors like Π and Σ with $[\![\mathtt{Type}]\!]$ structures as well. As $[\![\mathtt{Type}]\!]$ itself will contain a Type, the following situation will occur when we define the translation of sums: we should have $[\![\Sigma\ U\ T : \mathtt{Type}_j]\!] = [\![\Sigma]\!]\ [\![U]\!]\ [\![T]\!] : [\![\mathtt{Type}_j]\!]$. To ensure consistency the interpretations of Types inside $[\![U]\!]$, $[\![T]\!]$ and the outer one, they must be at different levels, with the outer one larger than the inner ones. This is however not supported in the current version of Coq, as the universe system does not allow a definition to live at different levels. Hence, there could be only one universe level assigned to the translation of any Type and they couldn't be nested, as this would result in an obvious inconsistency: the usual Type : Type inconsistency would show up as $[\![\mathtt{Type}]\!] : [\![\mathtt{Type}]\!]$. One solution to this problem would be to have n different interpretations of Type to handle n different levels of universes. This is clearly unsatisfactory, as this would mean also duplicating every lemma and every structure depending on the translation of types. Instead, we can extend the system with universe polymorphic definitions that are parametric on universe levels and instantiate them at different levels, just like parametric polymorphism is used to instantiate a definition at different types. This can be interpreted as (recursively) building fresh instances of the constant that can be handled by the core type checker without polymorphism.

The first author has developed a version of Coq with universe polymorphism that implements this system and can be used to check the translation (which expectedly fails with a universe inconsistency otherwise). The changes to the trusted code base of Coq are minimal, and amount to allow definitions to take a list of universes as parameters. When typechecking a constant application, we simply get the instantiated universe constraints, instead of a fixed one. This system is inspired from Harper and Pollack's [5] design for LEGO, adding a layer of inference that makes it entirely transparent to the user, much like Hindley-Milner polymorphic type inference in ML. Unlike the explicit universe polymorphism implemented in Agda, it handles typical ambiguity and cumulativity (subtyping for universes), making universe management completely implicit for the user.

This extension of Coq[1] was originally made to support the formalization of Homotopy Type Theory and is able to check for example Voevodsky's proof that Univalence implies Functional Extensionality. It can also check the formalization of 2-groupoids we will present now[2] This formalization is a great benchmark for universe polymorphism as it stresses the universe system by constructing a hierarchy of types embedded in nested structures.

3 The translation

3.1 Definition of weak-2-groupoids

We formalize weak-2-groupoids using type classes. Contrarily to what is done in the usual Setoid translation, the basic notion of morphism is a relation in `Type`:

```
Definition Hom (A : Type) := A → A → Type.
```

Given a morphism, we define type classes that represents that the Hom-set of morphisms on a `Type` A is reflexive (which corresponds to the identity morphism), symmetric (which corresponds to the existence of an inverse morphism for every morphism) and transitive (which corresponds to morphisms composition).

```
Class Identity {A} (M : Hom A) :=
  identity : ∀ x, M x x.

Class Inverse {A} (M : Hom A) :=
  inverse : ∀ x y:A, M x y → M y x.

Class Composition {A} (M : Hom A) :=
  composition : ∀ {x y z:A}, M x y → M y z → M x z.

Notation "g ∘ f" := (composition f g) (at level 50).
```

[1] Available at http://github.com/mattam82/CoqUnivs
[2] Available at http://tabareau.fr/univalence_for_free.

In a 2-groupoid, all 2-morphisms are invertible and higher equalities are trivial. Thus the set of 2-Homs denoted by $\sim_2$ corresponds to an equivalence relation.

```
Class Equivalence T (Eq : Hom T):= {
  Equivalence_Identity :> Identity Eq ;
  Equivalence_Inverse :> Inverse Eq ;
  Equivalence_Composition :> Composition Eq
}.
```

```
Class EquivalenceType (T : Type) : Type := {
  m2: Hom T;
  equiv_struct :> Equivalence m2 }.
```

```
Infix "∼2" := m2 (at level 80).
```

We start with the definition of 2-1 categories, that is weak 2-categories where 2-Homs are isoHoms. Technically, we use it for the 2-1 category of functors and iso-natural transformations, which is not a 2-groupoid but will be a useful structure in the definitions and proofs about type equivalences.

```
Class _1Hom T := {m1 : Hom T}.
```

```
Infix "∼1" := m1 (at level 80).
```

```
Class Weak2_1Category T := {
  M :> _1Hom T;
  eq_m :> ∀ x y, EquivalenceType (x ∼1 y) ;

  Weak2_1Category_Identity :> Identity m1 ;
  Weak2_1Category_Composition :> Composition m1;

  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'
}.
```

```
Definition Weak2_1CatType := {T:Type & Weak2_1Category T }.
```

Now, a weak 2-groupoid is just a 2-1 category where all 1-Homs are invertible and subject to additional compatibility laws on the inversion.

```
Class Weak2Groupoid T := {
  Weak2Groupoid_Weak2_1Category :> Weak2_1Category T ;
  Weak2Groupoid_Inverse :> Inverse m1 ;

  inv_R : ∀ x y (f: x ∼1 y), f ∘ (inverse _ _ f) ∼2 identity _ ;
  inv_L : ∀ x y (f: x ∼1 y), (inverse _ _ f) ∘ f ∼2 identity _ ;
  inv : ∀ x y (f f':x ∼1 y), f ∼2 f' → inverse _ _ f ∼2 inverse _ _ f'
}.
```

```
Definition Weak2GroupoidType := {T:Type & Weak2Groupoid T }.
```

Notation. We introduce the following notation that defines an application when the function is part of a dependent sum.

```
Notation "M ⋆ N" := ([M] N) (at level 55).
```

3.2 Prop extensionality and proof irrelevance

Equality on proofs is irrelevant. What we mean by irrelevant is that the set of (iso-)Homs between any two propositions is a singleton.

```
Definition Hom_irr (T : Type) : Hom T := λ _ _, unit.
```

We define an instance IrrRelWeak2Groupoid T m for Weak2Groupoid T when m is an equivalence and the second equality is relevant. We will use this instance to define 2-groupoid degenerated at level 2, as for instance for Prop.

```
Class PropIrr (P:Prop) : Type :=
  { prop_irr_groupoid := IrrRelWeak2Groupoid (m:= Hom_irr P) _}.
Program Definition Propositions := { P : Prop & PropIrr P }.
```

Equality between propositions of type Propositions is given by logical equivalence on the underlying propositions, i.e. propositional extensionality. This is a degenerate case of univalence, where the proofs that the two maps form an isomorphism is trivially true due to the above definition of equality of witnesses: they are all equal.

```
Definition eq_prop (P Q : Propositions) := [P] ↔ [Q].

Program Definition _Prop : Weak2GroupoidType :=
  (Propositions ; IrrRelWeak2Groupoid (m:=eq_prop) _).
```

3.3 Functional Extensionality and natural transformations

A morphism between two 2-groupoids is a 2-functor, i.e. a function between objects of the 2-groupoids that transports higher Homs, subject to compatibility laws.

```
Class Functor {T U : Weak2GroupoidType} (f : [T] → [U]) : Type := {
 map : ∀ {x y}, x ∼₁ y → f x ∼₁ f y ;
 map2 : ∀ x y (e e': x ∼₁ y), e ∼₂ e' → map e ∼₂ map e' ;
 map_comp : ∀ x y z (e:x ∼₁ y) (e':y ∼₁ z), map (e' ∘ e) ∼₂ map e' ∘ map e
}.
Definition Fun_Type (T U : Weak2GroupoidType) :=
  {f : [T] → [U] & Functor f}.
Infix "⟶" := Fun_Type (at level 55).
```

Note that we only impose compatibility with the composition as compatibilities with identities and inverse Homs can be deduced from it.

```
Lemma map_id {T U} (f : T ⟶ U) x :
  map [f] (identity x) ~2 identity (f ⋆ x).
Lemma map_inv {T U} (f : T ⟶ U) :
  ∀ x y (e : x ~1 y) , map [f] (inverse _ _ e) ~2 inverse _ _ (map [f] e).
```

Equivalence between functors is given by (iso-)natural transformations, which are actually equivalent to natural transformations for groupoids. We would like to insist here that this naturality condition in the definition of functional extensionality 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 [3].

```
Definition nat_trans T U (f g : T ⟶ U) :=
  {α : ∀ t : [T], f ⋆ t ~1 g ⋆ t &
  ∀ t t' (e : t ~1 t'), (α t') ∘ (map [f] e) ~2 (map [g] e) ∘ (α t)}.
```

The equivalence at the second level corresponds to a higher natural transformation but as the higher equivalences are trivial, there is no need for naturality.

```
Definition nat_trans2 T U (f g : T ⟶ U) : Hom (nat_trans f g) :=
  λ α β , ∀ t : [T], α ⋆ t ~2 β ⋆ t.
```

We can now equip the function space with its 2-groupoid structure. Note here that we (abusively) use the same notation for the functor type and its corresponding 2-groupoid.

```
Program Definition _fun T U : Weak2GroupoidType := (T ⟶ U ; _).
Infix "⟶" := _fun.
```

In the definition above, _ is instantiated by a proof that nat_trans and nat_trans2 form a 2-groupoid on $T \longrightarrow U$.

3.4 Homotopic equivalences

The natural notion of equivalence between 2-groupoids is given by homotopic equivalences, that is a map with its adjoint, and 2 proofs that they form a section and a retraction.

We could have used adjoint equivalences, that require two triangle identities between sections and retractions, but those compatibilities do not seem to be useful. Also, any equivalence can be turned into an adjoint equivalence by modifying the proof of the section.

```
Class Equiv_struct T U (f : T ⟶ U) := {
  adjoint : U ⟶ T ;
  section : f ∘ adjoint ~1 identity U ;
  retraction : adjoint ∘ f ~1 identity T }.

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

2-groupoids and homotopic equivalences between them form a 3-groupoid. Equality of homotopic equivalences is given by (iso-)natural transformations on underlying maps. As we only consider 2-groupoids, the fact that 2-groupoids form a 3-groupoid is not explicit in the formalism. We will see in the next sections that it will cause some problems in the equality that can derived on homotopic rewriting, with repercussion in the definition of dependent products and dependent sums.

Definition Equiv_eq T U : Hom (Equiv T U) := $\lambda\, f\ g$, $[f] \sim_1 [g]$.

In the definition below, _ is instantiated by a proof that Equiv and Equiv_eq form a 2-groupoid.

Program Definition _Type : Weak2GroupoidType
  := (Weak2GroupoidType ; _).

3.5 Rewriting in homotopy type theory

When considering a dependent type F: $[A \longrightarrow$ _Type$]$, the map function provides an homotopic equivalence between $F \star x$ and $F \star y$ for any x and y such that $x \sim_1 y$. But the underlying map of homotopic equivalence can be use to rewrite any term of type $[F \star x]$ to a term of type $[F \star y]$.

Definition eq_rect' (A : [_Type]) (x : $[A]$) (F: $[A \longrightarrow$ _Type$]$) (y : $[A]$)
  (e : $x \sim_1 y$) (p : $[F \star x]$) : $[F \star y]$:= [map $[F]$ e] $\star\ p$.

In the same way, a dependent type F: $[A \longrightarrow$ _Prop$]$ gives a way to transform any proof of $[F \star x]$ to a proof of $[F \star y]$.

Definition eq_ind' (A : [_Type]) (x : $[A]$) (F: $[A \longrightarrow$ _Prop$]$) (y : $[A]$)
  (e : $x \sim_1 y$) (p : $[F \star x]$) : $[F \star y]$:= let (l,r) := map $[F]$ e in l p.

Using compatibility on map, we can reason on different ways of rewriting. Intuitively, any two rewriting maps with the same domain and codomain should be the same up to homotopy. But as we only consider 2-groupoids, we are missing higher-order compatibilities. Here is an example of two derivable equalities between two rewriting maps.

Definition eq_rect'_eq (A : [_Type]) (x : $[A]$) (F: $[A \longrightarrow$ _Type$]$)
  (y : $[A]$) (e e': $x \sim_1 y$) (H : $e \sim_2 e$') (p : $[F \star x]$) :
  eq_rect' x F y e p $\sim_1$ eq_rect' x F y e' p := (map2 $[F]$ H) $\star\ p$.

Definition eq_rect'_map (A : [_Type]) (x : $[A]$) (F: $[A \longrightarrow$ _Type$]$)
  (p q : $[F \star x]$) (y : $[A]$) (e : $x \sim_1 y$) (H : $p \sim_1 q$) :
  eq_rect' x F y e p $\sim_1$ eq_rect' x F y e q := map [[map $[F]$ e]] H.

Definition eq_rect'_comp (A : [_Type]) (x y z : $[A]$) (F: $[A \longrightarrow$ _Type$]$)
  (e : $x \sim_1 y$) (e' : $y \sim_1 z$) (p : $[F \star x]$) :
  eq_rect' x F z (e' $\circ$ e) p $\sim_1$ (eq_rect' y F z e' (eq_rect' x F y e p)) :=
  (map_comp $[F]$ e e') $\star\ p$.

The fact that we are missing higher equalities on rewriting maps will become more apparent in the next two sections.

3.6 Dependent Product

As for function, a dependent function will be interpreted as a functor. But this time, the compatibilities with higher-order morphisms can not be expressed as simple equalities, as some rewriting as to be done to make those equalities typable. We call such a functor a *dependent functor*.

```
Class DependentFunctor (T:[_Type]) (U : [T ⟶ _Type]) (f : ∀ t, [U ⋆ t])
  : Type :=
{
  Dmap : ∀ {x y} (e: x ∼1 y), eq_rect' _ U _ e (f x) ∼1 f y ;
  Dmap2 : ∀ x y (e e': x ∼1 y) (H: e ∼2 e'),
      Dmap e ∼2 Dmap e' ∘ eq_rect'_eq x U y e e' H (f x) ;
  Dmap_comp : ∀ x y z (e : x ∼1 y) (e' : y ∼1 z),
      Dmap (e' ∘ e) ∘ inverse _ _ (eq_rect'_comp _ _ _ U e e' _) ∼2
      Dmap e' ∘ eq_rect'_map _ U _ _ _ _ (Dmap e)
}.
```

```
Definition Prod_Type (T:[_Type]) (U:[T ⟶ _Type]) :=
  {f : ∀ t, [U ⋆ t] & DependentFunctor U f}.
```

As it is the case for functor between 2-groupoids, the compatibilities with the identity and inverse morphism can be deduced from the compatibility with the composition. But for that, we need to reason on higher-order equalities that are not derivable in an n-groupoid setting. Of course, this problem would not appear when using ∞-groupoids. To show the validity of the approach, we have decided to prove the compatibility with the identity up-to a higher-order axiom on rewriting. This shows that in a 3-groupoid setting, the very same proof could be done, but this time without an axiom.

```
Axiom map2_id_L : ∀ T (U : [T ⟶ _Type]) (x y : [T]) (e:x ∼1 y),
  map2 [U] (id_L _ _ e) ∼2
  id_L _ _ (map [U] e) ∘
  comp _ _ _ _ _ _ _ _ (identity (map [U] e)) (map_id U _) ∘
  map_comp _ _ _.
```

```
Lemma Dmap_id (T:[_Type]) (U : [T ⟶ _Type]) (f : Prod_Type U) :
  ∀ x, Dmap [f] (identity x) ∼2 eq_rect'_id _ x (f ⋆ x).
```

Equality between dependent functors is given by dependent natural transformations. Again, at level 2, the naturality condition is trivial.

```
Definition Dnat_trans T (U:[T ⟶ _Type]) (F G: Prod_Type U) :=
  {α : ∀ t : [T], F ⋆ t ∼1 G ⋆ t & ∀ t t' e,
    (α t') ∘ (Dmap _ e) ∼2
    (Dmap _ e) ∘ eq_rect'_map _ U (F ⋆ t) (G ⋆ t) _ e (α t)}.
```

```
Definition Dnat_trans2 T U (f g : Prod_Type U) : Hom (Dnat_trans f g) :=
  λ α β , ∀ t : [T], α ⋆ t ∼2 β ⋆ t.
```

We can now equip the dependent functors with its 2-groupoid structure using Dnat_trans and Dnat_trans2 as underlying equalities.

```
Program Definition _Prod T (U:[T ⟶ _Type]) : [_Type] :=
  (Prod_Type U ; _).
```

3.7 Σ types and groupoid levels

In the interpretation of Σ types, the missing equalities coming from the truncation at level 2 of ∞-groupoids will be even more apparent. In the same way that a Σ type on a fibration $F : T \to$ Type in Coq lives in universe $n + 1$ when F lives in universe n, our interpretation of Σ type requires that F defines n-groupoids to get a n+1-groupoid Σ type. Of course, this problem doesn't show up with ∞-groupoids.

As for dependent products, we could solve this issue by assuming the missing compatibilities. We prefer here to define a 2-groupoid Σ type on weak 1-fibrations, that is functors of type $T \longrightarrow$ _Type that send every $t{:}T$ to a weak 1-groupoid, without resorting to axioms.

```
Definition Weak1Groupoid (T : [_Type]) : Type :=
  ∀ (x y : [T]) (f f': x ∼1 y), f ∼2 f'.
Definition Weak1Fibration (T : [_Type]) : Type :=
  { F : [T ⟶ _Type] & ∀ t, Weak1Groupoid (F ⋆ t)}.
```

As explained above, we need to restrict the construction of Σ types to Weak1Fibration.

```
Definition sum_type (T: [_Type]) (F : Weak1Fibration T) :=
  {t : [T] & [[F] ⋆ t]}.
```

1-equality between dependant pairs is given by 1-equality on the first and second projections, with a rewriting/transport on one second projection by the first equality.

```
Definition sum_eq (T: [_Type]) (F : Weak1Fibration T) :=
  λ (m n : sum_type F), {P : [m] ∼1 [n] & eq_rect' _ [F] _ P (π2 m) ∼1 π2 n}.
```

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

```
Definition sum_eq2 T (F:Weak1Fibration T) (M N : sum_type F) : Hom
(sum_eq M N) :=
  λ e e' , {P : [e] ∼2 [e'] & π2 e ∼2 π2 e' ∘ eq_rect'_eq _ _ _ _ _ P (π2 M)}.
```

```
Program Definition _Sum T (F:Weak1Fibration T) : [_Type] :=
  (sum_type F ; _).
```

The proof that we actually have a 2-groupoid makes use of the fact that $\sim_2$ on $F \star t$ is always trivial to complete proofs that would have been derivable at level 3 only.

3.8 The translation process

We now present a translation that internalizes homotopy type theory into the Calculus of Constructions using our 2-groupoid interpretation.

- $[\![$ Type $]\!] \equiv$ _Type
- $[\![$ Prop $]\!] \equiv$ _Prop
- $[\![\ T \to U\ ]\!] \equiv [\![\ T\ ]\!] \longrightarrow [\![\ U\ ]\!]$
- $[\![\ \forall\ t : T,\ U\ ]\!] \equiv$ _Prod $[\![\ (\lambda\ t,\ U\ ;\ \_)\ ]\!]$
- $[\![\ \lambda\ t : T,\ \mathrm{M}\ ]\!] \equiv (\lambda\ t : [\![\ T\ ]\!]\ ,\ [\![\ \mathrm{M}\ ]\!]\ ;\ \_)$
- $[\![\ x{:}A\ ]\!] \equiv x : [\ [\![\ T\ ]\!]\ ]$
- $[\![\ \mathrm{M}\ N\ ]\!] \equiv [\![\ \mathrm{M}\ ]\!] \star [\![\ N\ ]\!]$
- $[\![\ \{t : T,\ U\}\ ]\!] \equiv$ _Sum $[\![\ (\lambda\ t,\ U\ ;\ \_)\ ]\!]$
- $[\![\ \pi_i\ \mathrm{M}\ ]\!] \equiv \pi_i\ [\![\ \mathrm{M}\ ]\!]$
- $[\![\ x = y\ ]\!] \equiv [\![\ x\ ]\!] \sim_1 [\![\ y\ ]\!]$

Note that in the translation of products, functions and sums, there is a remaining proof obligation that the defined function is actually a functor or a dependent functor.

It is still an open problem to know whether every such obligations can be automatically computed from the original term before translation. The connection between ∞-groupoids and homotopy type theory guarantees that such proofs exist but does not say much about their shape.

3.9 Univalence axiom and others as lemmas

As a sanity check, we now prove that the equality we defined on propositions, Prop, functions, dependent pairs and Type behave as expected.

First, we state that equality on proofs is irrelevant and equality on propositions is given by $\leftrightarrow$.

Lemma prop_extensional $(P\ Q : [\_\mathrm{Prop}]) : [P] \leftrightarrow [Q] \to P \sim_1 Q$.

Lemma proof_irrelevant $(P : [\_\mathrm{Prop}])\ (p\ q : [P]) : p \sim_1 q$.

We also have functional extensionality, for the dependent and non-dependent function spaces, at the price of a naturality condition (see nat_trans).

Lemma functional_extensionality $A\ B\ (f\ g : [A \longrightarrow B])$:
nat_trans $f\ g \to f \sim_1 g$.

Lemma functional_extensionality_dep $T\ U\ (f\ g : [\_\mathrm{Prod}\ (T{:=}T)\ U])$:
Dnat_trans $f\ g \to f \sim_1 g$.

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

Lemma sum_extensional $T\ F\ (m\ n : [\_\mathrm{Sum}\ (T{:=}T)\ F])$:
$\forall\ (P : [m] \sim_1 [n])$, eq_rect' _ $[F]$ _ $P\ (\pi_2\ m) \sim_1 \pi_2\ n \to m \sim_1 n$.

Finally, we have the univalence principle on types.

Lemma univalence_statement $(U\ V : [\_\mathrm{Type}]) : (\mathrm{Equiv}\ U\ V) \to U \sim_1 V$.

4 An example using univalence

To illustrate the use of univalence in our equality, we will define the type $\forall\ X$: Type, $(X \rightarrow X) \rightarrow X \rightarrow X$ of Church naturals and show that it is *equal* to the inductive type nat of inductive Coq naturals.

Showing that those two types are equal amounts to constructing a homotopic equivalence between them. The proof that we actually have an equivalence relies on the fact that every inhabitant of cnat is parametric with respect to the variable X. This principle is not provable in Coq and will be posed as an axiom in our development. However it is validated meta-theoretically by a parametricity result on (a slight refinement of) CIC in Keller and Lasson's work [8].

The first problem to define the type of Church integers is to derive the (nested) proofs of functoriality of the function $\lambda\ X, (X \rightarrow X) \rightarrow X \rightarrow X$. To do that, we show that the arrow $(\longrightarrow)$ is functorial. Based on this, we can construct an other arrow $(\twoheadrightarrow)$ on endofunctors on _Type.

```
Program Definition endo_fun (f g : [_Type ⟶ _Type]) : [_Type ⟶ _Type]
  := (λ X, (f ⋆ X) ⟶ (g ⋆ X); _).
Infix ”↠” := endo_fun (at level 80, right associativity).
```

This arrow expects two endofunctors on _Type and pre-composes them with $(\longrightarrow)$ to get a new endofunctor on _Type.

The idea is to use an encoding of λ-terms with one free variable as endofunctors to define the type of Church naturals. In this encoding, the variable is seen as the identity functor.

```
Definition Var := identity _Type : [_Type ⟶ _Type].
```

Then, the functor cnatT corresponding to the term $\lambda\ X, (X \rightarrow X) \rightarrow X \rightarrow X$ can be defined directly.

```
Definition cnatT : [_Type ⟶ _Type] := (Var ↠ Var) ↠ Var ↠ Var.
```

As all our constructions are functorial, there is no need to prove extra compatibilities to define the type of Church naturals.

```
Program Definition cnat := _Prod cnatT : [_Type].
```

Let us now define the 2-groupoid of inductive natural numbers. It has natural numbers as objects, the Leibniz equality (eq of Coq) for $\sim_1$ and is irrelevant on $\sim_2$. This corresponds to the fact that uniqueness of (Leibniz) identity proofs for nat holds in CIC, as for any decidable type.

```
Program Definition _nat : [_Type] :=
  (nat ; IrrRelWeak2Groupoid (m := eq) _ ).
```

The zero of _nat is still the zero 0 of nat, but we need to promote the successor constructor S to a functor succ that contains the (trivial) proofs of compatibility with Leibniz equality.

```
Program Definition succ : [_nat ⟶ _nat] := (λ n , S n; _).
```

We now turn into the definition of the map from inductive naturals to Church naturals using an iterator. This iterator takes an integer k and applies the first argument (the successor function) k times to the second argument (the zero).

```
Fixpoint iter_ k (X : [_Type]) (s : [X ⟶ X]) (z : [X]) : [X] :=
  match k with
    | 0 ⇒ z
    | S p ⇒ s ⋆ (iter_ p X s z)
  end.
```

Again, we also need a proof that this iterator is functorial.

```
Program Definition nat_to_cnat : [_nat ⟶ cnat] :=
  (λ k, (λ X, iter k X ; _ ); _).
```

The definition of the functor in the other direction is more direct. It just consists in applying _nat, succ and 0 to a (parametric) Church integer to get the natural number that it computes.

```
Program Definition cnat_to_nat : [cnat ⟶ _nat] :=
  (λ cn, cn ⋆ _nat ⋆ succ ⋆ 0 ; _).
```

In that case, the proof of functoriality is easy as it lives in _nat which is a 1-groupoid with the Leibniz equality.

Proving functoriality and naturality. In the rest of this example, we have to deal with very large proofs of functoriality and naturality. Those proofs requires a lot of rewriting which could be done with a generalization of the `setoid_rewrite` tactic.

This generalized tactic has not been defined for the moment, so we have posed functoriality and naturality as axioms to discharge the proof of the remaining definitions. Again, this is not problematic as the model guarantees the existence of such proofs, but it is nevertheless a bit unsatisfactory and we hope to solve this issue soon.

The axioms are only used in the definition of the parametricity axiom (which involves many quantifications), and the definition Church_exists which uses the parametricity axiom.

Using those axioms, we can define a ∀' notation that takes a function $\lambda\ (x : T),\ U$ and returns a dependent type by using the axiom that every dependent function is a dependent functor.

```
Notation "'∀'' ( x : T ) , U" := (_Prod' T (λ x ,U))
(at level 200, x at level 99).
```

To show that _nat $\sim_1$ cnat, it now remains to prove that cnat_to_nat and nat_to_cnat define an isomorphism. But without any parametricity argument, there is no hope to show that all inhabitants of cnat behave uniformly with respect to the quantified type X.

Note that the requirement that any Church integer has to be a dependent functor guarantees that a Church integer behaves informally with respect to isomorphic types, but not with respect to any two types.

```
Axiom cnat_parametricity :
 [∀' (cn : cnat),
  ∀' (X : _Type), ∀' (X' : _Type),
  ∀' (R : X ⟶ X' ⟶ _Type), ∀' (H : X ⟶ X),
  ∀' (H0 : X' ⟶ X'),
    (∀' (H1 : X), ∀' (H2 : X'),
     R ⋆ H1 ⋆ H2 ⟶ R ⋆ (H ⋆ H1) ⋆ (H0 ⋆ H2)) ⟶
    (∀' (H1 : X), ∀' (H2 : X'),
     R ⋆ H1 ⋆ H2 ⟶ R ⋆ ((cn ⋆ X) ⋆ H ⋆ H1) ⋆ ((cn ⋆ X') ⋆ H0 ⋆ H2))].
```

Using parametricity, we can prove that any Church integer *cn* is actually equal to the image of some *k* through nat_to_cnat. In the proof, *k* is actually instantiated by cnat_to_nat ⋆ *cn*.

```
Program Definition Church_exists :
  ∀ (cn : [cnat]), {k:[_nat] & cn ~1 nat_to_cnat ⋆ k}.
```

Using these two lemmas, it is not difficult to show that we have a section and a retraction, forming an homotopic equivalence.

```
Program Instance nat_to_cnat_eq : Equiv_struct nat_to_cnat :=
  {adjoint := cnat_to_nat}.

Definition ι : _nat ~1 cnat := (nat_to_cnat ; nat_to_cnat_eq).
```

The witness of equality *ι* can be used to import directly, for Church naturals, all definitions and lemmas valid for inductive naturals.

For instance, we show how to define the addition on Church naturals and prove its commutativity using map [*ι*] and the proof of commutativity on nat.

```
Definition cnat_plus (cn cn': [cnat]) :=
  [ι] ⋆ ((adjoint [ι] ⋆ cn) + (adjoint [ι] ⋆ cn')).

Infix "+c" := cnat_plus (at level 50).

Definition cnat_plus_comm (cn cn': [cnat]) : cn +c cn' ~1 cn' +c cn :=
  map [[ι]] (plus_comm _ _).
```

5 Conclusion and future work

We have presented a 2-groupoid interpretation of type theory in a version of Coq supporting polymorphic universes. In the course of the formalization we uncovered necessary naturality and truncation conditions along with a definition of dependent functor that are required in that setting. Finally we saw how to build on this construction to transport definitions and functions between isomorphic types: the rewriting involved there has the actual computational content of applying the isomorphism wherever needed.

To really complete the formalization, we need better support for automatic rewriting with computational relations, to prove higher-dimensional proofs. This is a first step to be able to automatically discharge the naturality conditions that arise during translation. Having a direct translation would be a convenient laboratory to test new principles coming from homotopy type theory. For instance, a complete definition of inductives in that setting is still an open question and being able to translate and type-checked the new definitions will be a nice way to certify their correction.

Another direct perspective is to combine this translation with our Forcing translation as it requires proof irrelevance and functional extensionality which are not natively available in Coq.

References

1. Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational Equality, Now! In *PLPV'07: Proceedings of the Programming Languages meets Program Verification Workshop*, Freiburg, Germany, 2007.
2. Thorsten Altenkirch and Ondrej Rypacek. A Syntactical Approach to Weak omega-Groupoids. In Cégielski and Durand [4], pages 16–30.
3. Andrej Bauer and Peter LeFanu Lumsdaine. A Coq proof that Univalence Axioms implies Functional Extensionality. `http://ncatlab.org/nlab/files/BauerLumsdaineUnivalence.pdf`, 2011.
4. Patrick Cégielski and Arnaud Durand, editors. *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
5. Robert Harper and Robert Pollack. Type checking with universes. *Theor. Comput. Sci.*, 89(1):107–136, 1991.
6. Martin Hofmann and Thomas Streicher. The Groupoid Interpretation of Type Theory. In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 83–111. Oxford Univ. Press, New York, 1998.
7. Guilhem Jaber, Nicolas Tabareau, and Matthieu Sozeau. Extending Type Theory with Forcing. In *Proceedings of LICS'12*, Dubrovnik, Croatie, June 2012.
8. Chantal Keller and Marc Lasson. Parametricity in an Impredicative Sort. In Cégielski and Durand [4], pages 381–395.
9. Daniel R. Licata and Robert Harper. Canonicity for 2-dimensional type theory. In John Field and Michael Hicks, editors, *POPL*, pages 337–348. ACM, 2012.
10. Peter LeFanu Lumsdaine. Weak omega-categories from intensional type theory. *Logical Methods in Computer Science*, 6(3), 2010.
11. Álvaro Pelayo and Michael A. Warren. Homotopy type theory and Voevodsky's univalent foundations. 10 2012.
12. Matthieu Sozeau. Program-ing Finger Trees in Coq. In *ICFP'07: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming*, pages 13–24, Freiburg, Germany, 2007. ACM Press.
13. Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In César Muñoz Otmane Ait Mohamed and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21th International Conference*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, August 2008.

14. The Coq development team. *Coq 8.4 Reference Manual*. INRIA, 2012.
15. Vladimir Voevodsky. Univalent Foundations of Mathematics. In LevD. Beklemishev and Ruy Queiroz, editors, *Logic, Language, Information and Computation*, volume 6642, pages 4–4. Springer Berlin Heidelberg, 2011.