Formalisations Using Two-Level Type Theory

Danil Annenkov[1] Paolo Capriotti[2] Nicolai Kraus[2]

[1]University of Copenhagen

[2]University of Nottingham

September 9, 2017

Two-Level Type Theory

Two-level type theory - a version of Martin-Löf type theory with two equality types: the usual equality of HoTT, and the strict equality.

The plan for this talk:

- discuss a motivation for two-level type theory;
- give a definition of two-level type theory;
- describe our implementation in the Lean proof assistant;
- show some applications.

This talk is based on:
Danil Annenkov, Paolo Capriotti, and Nicolai Kraus.
Two-Level Type Theory and Applications. Submitted to TOCL.
ArXiv e-prints, May 2017.
https://arxiv.org/abs/1705.03307
And the previous work:
Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus.
Extending Homotopy Type Theory with Strict Equality. CSL 2016.

Motivation for Two-Level Type Theory

- Complete internalisation of results, which are only *partially* internal to HoTT (n-restricted semi-simplicial types, univalent n-categories, inverse diagrams).
- Allows to extend homotopy type theory in a "controlled" way (add additional axioms).

Definition of *n*-restricted semi-simplicial types

For any *externally* fixed n one can write a definition in any proof assistant (we use Lean)

n-restricted semi-simplicial types for $n = 3$

```
definition SST₃ :=
Σ (X₀ : Type)
  (X₁ : X₀ → X₀ → Type),
   Π (x₀ x₁ x₂ : X₀),
     X₁ x₀ x₁ → X₁ x₁ x₂ → X₁ x₀ x₁ → Type
```

Or even write a script that generates definitions for a given n.

But the general definition of n-restricted semi-simplicial types for arbitrary n in HoTT is an open problem.

Internalising inverse diagrams

- Work on inverse diagrams by Michael Shulman[1];
- one can do constructions in type theory fixing a (finite) inverse category *in the meta-theory*;
- inverse diagrams and *n*-restricted semi-simplicial types can be internalised in two-level type theory
- we will discuss the example of inverse diagrams later in the talk.

[1] Michael Shulman. Univalence for Inverse Diagrams and Homotopy Canonicity. Mathematical Structures in Computer Science, pages 1–75, 2015.

Two-Level Type Theory

- *strict* fragment: a form of Martin-Löf Type Theory (MLTT) with Uniqueness of Identity Proofs (UIP);
- *fibrant* fragment: Homotopy Type Theory;

Inspired by Homotopy Type System (HTS)[2], but with some important differences.

[2]Vladimir Voevodsky. A simple type system with two identity types, 2013.Unpublished note.

Differences with HTS

- UIP instead of equality reflection;
- HTS assumes that $\mathbf{0}$, $\mathbb{N}$ and $+$ from the fibrant fragment eliminate to arbitrary types, we leave it open;
- the conservativity result[3].

[3] Paolo Capriotti. Models of Type Theory with Strict Equality. PhD thesis, School of Computer Science, University of Nottingham, Nottingham, UK, 2016.

Two-Level Type Theory : Types and Type Formers

- **Fibrant fragment**:
  all the basic types and type formers found in HoTT
  $\mathbf{1}$, $\mathbf{0}$, $\mathbb{N}$, $=$ (the equality type);
  Π, Σ, $+$;
  a hierarchy $\mathcal{U}_0, \mathcal{U}_1, \ldots$ of universes;
  elements of $\mathcal{U}_i$ - *fibrant types* (or just **types**)

- **Strict fragment**:
  $\mathbf{0}^{\mathrm{s}}$, $\mathbb{N}^{\mathrm{s}}$, $+^{\mathrm{s}}$, $\overset{\mathrm{s}}{=}$ (the strict equality);
  a hierarchy $\mathcal{U}^{\mathrm{s}}_0, \mathcal{U}^{\mathrm{s}}_1, \ldots$ of strict universes;
  elements of $\mathcal{U}^{\mathrm{s}}_i$ - **pretypes**

The type formers Π, Σ, $\mathbf{1}$ are shared by the two fragments.

Two-Level Type Theory : Fibrancy Rules

Every type is also a pretype

$$\frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A : \mathcal{U}_i^{\mathrm{s}}} \quad \text{FIB-PRE}$$

Π and Σ preserve fibrancy

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma.A \vdash B : \mathcal{U}_i}{\Gamma \vdash \Pi_A B : \mathcal{U}_i} \quad \text{PI-FIB}$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma.A \vdash B : \mathcal{U}_i}{\Gamma \vdash \Sigma_A B : \mathcal{U}_i} \quad \text{SIGMA-FIB}$$

Two-Level Type Theory : Fibrant Equality

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash a_1, a_2 : A}{\Gamma \vdash a_1 = a_2 : \mathcal{U}_i}\ \text{FORM-=}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma(b : A)(p : a = b) \vdash P : \mathcal{U}_i \qquad \Gamma \vdash d : P[a, \mathsf{refl}_a]}{\Gamma(b : a)(p : a = b) \vdash J_P(d) : P}\ \text{ELIM-=}$$

Note: FORM-= and ELIM-= only work for **(fibrant) types!**
The computation rule

$$J_P(d)[a, \mathsf{refl}_a] \stackrel{s}{=} d$$

Note: the computation rule defined using strict equality.
Univalence: for any two **(fibrant) types** $X, Y : \mathcal{U}$, the map $(X = Y) \to (X \simeq Y)$ is an equivalence.

Two-Level Type Theory : Strict Equality

$$\frac{\Gamma \vdash A : \mathcal{U}_i^{\mathrm{s}} \qquad \Gamma \vdash a, b : A}{\Gamma \vdash a \overset{\mathrm{s}}{=} b : \mathcal{U}_i^{\mathrm{s}}}\,\text{FORM-}\overset{\mathrm{s}}{=}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma(b : A)(p : a \overset{\mathrm{s}}{=} b) \vdash P : \mathcal{U}_i^{\mathrm{s}} \qquad \Gamma \vdash d : P[a, \mathsf{refl}_a^{\mathrm{s}}]}{\Gamma(b : a)(p : a \overset{\mathrm{s}}{=} b) \vdash J_P^{\mathrm{s}}(d) : P}\quad\text{ELIM-}\overset{\mathrm{s}}{=}$$

The computation rule holds judgmentally:

$$J_P^{\mathrm{s}}(d)[a, \mathsf{refl}_a^{\mathrm{s}}] \equiv d.$$

Strict equality satisfies UIP

$$\frac{\Gamma \vdash a_1, a_2 : A \qquad \Gamma \vdash p, q : a_1 \overset{\mathrm{s}}{=} a_2}{\Gamma \vdash K^{\mathrm{s}}(p, q) : p \overset{\mathrm{s}}{=} q}\quad\text{UIP}$$

Our Lean development

- We start in "strict" (proof irrelevant) Lean mode (we use Lean 2)
- Lean's Type is now a pretype;
- we use Lean's type classes to encode fibrancy;
- fibrant types:

```
constant is_fibrant_internal : Type → Prop
structure is_fibrant [class] (X : Type) := mk ::
  fib_internal : is_fibrant_internal X

structure Fib : Type := mk ::
  (pretype : Type)
  (fib : is_fibrant pretype)
```

Attributes

- implements the FIB-PRE rule

```
attribute Fib.pretype [coercion]
```

- makes an instance available for Lean's instance resolution mechanism

```
attribute Fib.fib [instance]
```

Type Class Instances Resolution

```
variables {A : Fib} {B : Fib} {C : Fib}
definition prod_assoc : A × (B × C) ≃ (A × B) × C :=
    sorry
```

```
  prod_assoc :
   Π {A} {B} {C},
    @fib_equiv (prod A (prod B C)) (prod (prod A B) C)
      -- inferred by Lean --
      (@prod_is_fibrant A (prod B C) (Fib.fib A)
          (@prod_is_fibrant B C (Fib.fib B) (Fib.fib C)))
      (@prod_is_fibrant (prod A B) C
          (@prod_is_fibrant A B (Fib.fib A) (Fib.fib B))
              (Fib.fib C))
      ----------------------
```

Lean vs. Agda

Corresponding code in Agda fails to infer implicit arguments ("$\sim$" means fibrant equality)

```
definition pi_eq {A : Type} [fibA : is_fibrant A]
                 {Q : A → Type}
                 [fibB : Π a, is_fibrant (Q a)]
    : Π (f : Π (a :A), Q a), f ∼ f := λ x, refl _
```

Resulting term in Lean:

```
pi_eq : Π {A} [fibA] {Q} [fibB] f,
  @fib_eq (Π a, Q a) (@pi_is_fibrant A Q fibA fibB) f f
```

Example

Let's consider an example from the HoTT Lean library
Note: here "=" is fibrant (and the only available) equality.

```
definition prod_transport (p : a = a′) (u : P a × Q a) :
  p ▷ u = (p ▷ u.1, p ▷ u.2) :=
by induction p; induction u; reflexivity
```

After induction on p and u:

$$\mathtt{refl}_a \triangleright (a_1,\ a_2) = (\mathtt{refl}_a \triangleright (a_1,\ a_2).1,\ \mathtt{refl}_a \triangleright (a_1,\ a_2).2)$$

Computation rule for transport holds judgmentally, so, we can prove this by $\mathrm{refl}_{(a_1,a_2)}$

Proof in the Fibrant Fragment

The same lemma in the fibrant fragment:

```
definition prod_transport (p : a ∼ a′) (u : P a  × Q a) :
  p ▷ u ∼ (p ▷ u.1, p ▷ u.2) :=
by induction p; induction u; repeat rewrite transport_β
```

After induction on p and u:

$\mathtt{refl}_a \triangleright (\mathtt{a}_1, \mathtt{a}_2) \sim (\mathtt{refl}_a \triangleright (\mathtt{a}_1, \mathtt{a}_2).1, \mathtt{refl}_a \triangleright (\mathtt{a}_1, \mathtt{a}_2).2)$

Simplification of the goal only makes projections go away:

$\mathtt{refl}_a \triangleright (\mathtt{a}_1, \mathtt{a}_2) \sim (\mathtt{refl}_a \triangleright \mathtt{a}_1, \mathtt{refl}_a \triangleright \mathtt{a}_2)$

Have to rewrite explicitly with "propositional" computation rule transport$_\beta$, or use the simp tactic:

```
by induction p; induction u; simp
```

Some Complications

The computation rule for apd (doesn't work!):

$$\texttt{apd f refl}_x \overset{\mathrm{s}}{=} \texttt{refl}_{(f\,x)},$$

Sides of the equation are of the different type:

$$\texttt{apd f refl}_x : \texttt{refl}_x \triangleright \texttt{(f x)} \sim \texttt{f x}$$
$$\texttt{refl}_{(f\,x)} : \texttt{f x} \sim \texttt{f x}$$

Definitions become awkward ($\triangleright_s$ is transport along the strict equality):

$$\texttt{apd}_\beta\ \texttt{\{P : X} \to \texttt{Fib\} (f : } \Pi \texttt{ x, P x) \{x y : X\} :}$$
$$\texttt{(transport}_\beta\ \texttt{(f x))} \triangleright_s \texttt{(apd f refl}_x\texttt{)} \overset{\mathrm{s}}{=} \texttt{refl}_{(f\,x)}$$

Some Complications: Possible Solution

- Keep only some basic computation rules (like $\texttt{elim}_\beta$ for fibrant equality elimination, maybe couple more);
- annotate these rules with the `[simp]` attribute;
- unfold definitions to get a goal where these basic rules are applicable;
- rewrite with basic rules or use `simp`.

Pros: Worked for proofs we ported from the Lean HoTT library so far (not too many).
Cons: More complicated situations where computation in types can happen could still be a problem.

Note: the Coq development by Simon Boulier and Nicolas Tabareau makes use of private inductive types to resolve this issue.

Application: Inverse Diagrams

Definitions from the Lean's standard library used in our formalisation:

- categories;
- functors;
- natural transformations.

The following notions we had to implement:

- pullbacks and general limits;
- construction of the limit for the Pretype category;
- coslice and reduced coslice;
- matching object;
- inverse categories;
- properties of the strict isomorphism and lemmas about finite sets

Inverse Diagrams

We have fully formalised in Lean the following theorem[4]:

Theorem (Fibrant limit)

Assume that $\mathcal{C}$ is an inverse category with a finite type of objects $|\mathcal{C}|$. Assume further that $X : \mathcal{C} \to \mathcal{U}^{\mathrm{s}}$ is a Reedy fibrant diagram which is pointwise essentially fibrant (which means we may assume that it is given as a diagram $\mathcal{C} \to \mathcal{U}$).
Then, X has a fibrant limit.

[4] cf. lemma 11.8 in Michael Shulman. Univalence for Inverse Diagrams and Homotopy Canonicity. Mathematical Structures in Computer Science, pages 1–75, 2015.

Type Classes And Proofs

Lean resolves instances of strict isomorphism to complete the proof (if there are enough instances in scope)

```
definition singleton_contr_fiberₛ {E B : Type}
                                  {p : E → B}
  : (Σ b, fibreₛ p b) ≃ₛ E :=
    calc
    (Σ b x, p x = b) ≃ₛ (Σ x b, p x = b) : _
                 ...  ≃ₛ (Σ (x : E), poly_unit) : _
                 ...  ≃ₛ E : _
```

Inverse Diagrams

Formalisation went reasonably well

- proof of the theorem in Lean is quite close to the representation in paper;
- the `calc` environment is convenient to write reasoning steps involving isomorphisms;
- Lean's type classes are helpful.

Tricky/tedious bits:

- choosing and removing the element with the maximal rank from $\mathcal{C}$, and showing that resulting $\mathcal{C}'$ is still finite, inverse and $X : \mathcal{C}' \to \mathcal{U}$ is Reedy fibrant (a lot of boilerplate);
- it would be nice to have a more developed library of strict categories (could save us some time);
- Lean error messages could have been more informative :)

Conclusion

- two-level type theory gives a uniform framework for internalising results which cannot be fully internalised in HoTT;
- it is possible to implement two-level type theory in an existing proof assistant, although require significant efforts;
- we demonstrated the prototype implementation in Lean, which uses type classes and some proof automation;
- we developed an internalisation of some results on inverse diagrams in Lean.

Further work

- extend our development with more results from the paper;
- explore the conservativity result.

Thank you

Thank you for your attention!