Type Theory in Type Theory using Quotient Inductive Types*

Thorsten Altenkirch and Ambrus Kaposi
School for Computer Science, University of Nottingham, United Kingdom
{txa, auk}@cs.nott.ac.uk

Abstract

We present an internal formalisation of a type theory with dependent types in Type Theory using a special case of higher inductive types from Homotopy Type Theory which we call quotient inductive types (QITs). Our formalisation of type theory avoids referring to preterms or a typability relation but defines directly well typed objects by an inductive definition. We use the elimination principle to define the set-theoretic and logical predicate interpretation. The work has been formalized using the Agda system extended with QITs using postulates.

Quotient Inductive Types (QITs)

QITs are a special case of Higher Inductive Types in a strict type theory where all higher path spaces are trivial. They allow the definition of usual constructors and equality constructors *at the same time*. We can simulate them in Agda by postulating the equality constructors and defining the eliminator. An example is the type of infinitely branching trees where the actual order of subtrees doesn't matter. The definition below is not the same as quotienting infinite branching trees because we were not able to lift the node constructor to the quotient.

```
data T :  Set where
  leaf  :  T
  node  :  (ℕ → T) → T
postulate
  perm  :  (g : ℕ → T) (f : ℕ → ℕ) → isIso f
        → node g ≡ node (g ∘ f)
module ElimT
  (Tᴹ     :  T → Set)
  (leafᴹ  :  Tᴹ leaf)
  (nodeᴹ  :  {f : ℕ → T} (fᴹ : (n : ℕ) → Tᴹ (f n))
             → Tᴹ (node f))
  (permᴹ  :  {g : ℕ → T} (gᴹ : (n : ℕ) → Tᴹ (g n))
             (f : ℕ → ℕ) (p : isIso f)
             → nodeᴹ gᴹ ≡[ ap Tᴹ (perm g f p) ]≡ nodeᴹ (gᴹ ∘ f))
  where
  Elim : (t : T) → Tᴹ t
  Elim leaf = leafᴹ
  Elim (node f) = nodeᴹ (λ n → Elim (f n))
```

Results

- We have for the first time presented a workable internal syntax of Type Theory which only features typed objects.
- We implemented the following models:
  - Standard model (metacircular interpretation)
  - Logical predicate interpretation (Bernardy-Jansson-Paterson, 2012)
  - Presheaf model
  - In preparation: Normalisation by Evaluation

Template Type Theory

- Internalising type theory opens the possibility of *template type theory*. An interpretation of type theory can be given as an algebra for the syntax and the interpretation of new constants in this algebra. We can then interpret code using these new principles by interpreting it in the given algebra. The new code can use all the conveniences of the host system such as implicit arguments and definable syntactic extensions.
- Some possible applications:
  - Using presheaf models to justify guarded type theory.
  - Modelling the local state monad (Haskell's STM monad)
  - Computational explanation of Homotopy Type Theory by the cubical set model
  - Derivation of parametricity results using the logical predicate interpretation
  - Generic programming

* Supported by EPSRC grant EP/M016951/1.

Syntax of Type Theory

The syntax is presented as a Quotient Inductive Inductive Type. Signature of the types representing the syntax:

```
data Con  : Set
data Ty   : Con → Set
data Tms  : Con → Con → Set
data Tm   : ∀ Γ → Ty Γ → Set
```

Constructors for contexts, types, substitutions and terms:

```
data Con where
  •      : Con
  _,_    : (Γ : Con) → Ty Γ → Con
data Ty where
  _[_]T  : Ty Δ → Tms Γ Δ → Ty Γ
  Π      : (A : Ty Γ) (B : Ty (Γ , A)) → Ty Γ
  U      : Ty Γ
  El     : (A : Tm Γ U) → Ty Γ
data Tms where
  ϵ      : Tms Γ •
  _,_    : (δ : Tms Γ Δ) → Tm Γ (A [ δ ]T) → Tms Γ (Δ , A)
  id     : Tms Γ Γ
  _∘_    : Tms Δ Σ → Tms Γ Δ → Tms Γ Σ
  π₁     : Tms Γ (Δ , A) → Tms Γ Δ
data Tm where
  _[_]t  : Tm Δ A → (δ : Tms Γ Δ) → Tm Γ (A [ δ ]T)
  π₂     : (δ : Tms Γ (Δ , A)) → Tm Γ (A [ π₁ δ ]T)
  app    : Tm Γ (Π A B) → Tm (Γ , A) B
  lam    : Tm (Γ , A) B → Tm Γ (Π A B)
```

Equality constructors for types, substitutions and terms:

```
postulate -- Ty
  [id]T  : A [ id ]T ≡ A
  [][]T  : A [ δ ]T [ σ ]T ≡ A [ δ ∘ σ ]T
  U[]    : U [ δ ]T ≡ U
  El[]   : El A [ δ ]T ≡ El (coe (TmΓ≡ U[]) (A [ δ ]t))
_ ↑ _ : (δ : Tms Γ Δ) (A : Ty Δ) → Tms (Γ , A [ δ ]T) (Δ , A)
δ ↑ A = (δ ∘ π₁ id) , coe (TmΓ≡ [][]T) (π₂ id)
  Π[]    : (Π A B) [ δ ]T ≡ Π (A [ δ ]T) (B [ δ ↑ A ]T)
postulate -- Tms
  idl    : id ∘ δ ≡ δ
  idr    : δ ∘ id ≡ δ
  ass    : (σ ∘ δ) ∘ ν ≡ σ ∘ (δ ∘ ν)
  ,∘     : (δ , t) ∘ σ ≡ (δ ∘ σ) , coe (TmΓ≡ [][]T) (t [ σ ]t)
  π₁β    : π₁ (δ , t) ≡ δ
  πη     : (π₁ δ , π₂ δ) ≡ δ
  ϵη     : {σ : Tms Γ •} → σ ≡ ϵ
postulate -- Tm
  [id]t  : t [ id ]t ≡[ TmΓ≡ [id]T ]≡ t
  [][]t  : (t [ δ ]t) [ σ ]t ≡[ TmΓ≡ [][]T ]≡ t [ δ ∘ σ ]t
  π₂β    : π₂ (δ , a) ≡[ TmΓ≡ (ap (_[_]T A) π₁β) ]≡ a
  Πβ     : app (lam t) ≡ t
  Πη     : lam (app t) ≡ t
  lam[]  : (lam t) [ δ ]t ≡[ TmΓ≡ Π[] ]≡ lam (t [ δ ↑ A ]t)
```