Type theory in Type Theory using Quotient-Inductive-Inductive-Recursive Types

Ambrus Kaposi

Eötvös Loránd University, Budapest

The 60th Thorsten Festival
Nottingham
12 October 2022

Syntax of type theory

- A model of type theory is a Category with Families (CwF) with extra structure for type formers.
- The syntax is the initial model.
  - well-formed
  - well-scoped
  - well-typed (intrinsic)
  - quotiented
- CwF with extra structure is a finitary Generalised Algebraic Theory (GAT).
- Our metatheory is type theory. The initial model is a finitary Quotient Inductive-Inductive Types (QIIT).

Example: canonicity

- Every boolean term in the empty context is either true or false.
- Proof: by induction on the syntax, using the method of proof-relevant logical predicates.
- The syntax is a category (and more), to do induction on it we define a displayed category (and more).

Category, displayed category

```
Con   : Set
Sub   : Con → Con → Set
_∘_   : Sub Δ Γ → Sub Θ Δ → Sub Θ Γ
id    : Sub Γ Γ
idl   : id ∘ γ = γ
```

```
Con• : Con → Set
Sub• : Con• Δ → Con• Γ → Sub Δ Γ → Set
_∘•_ : Sub• Δ• Γ• γ → Sub• Θ• Δ• δ →
       Sub• Θ• Γ• (γ ∘ δ)
id•  : Sub• Γ• Γ• id
idl• : id• ∘• γ• = γ•
```

Category, displayed category

```
Con   : Set
Sub   : Con → Con → Set
_∘_   : Sub Δ Γ → Sub θ Δ → Sub θ Γ
id    : Sub Γ Γ
idl   : id ∘ γ = γ


Con•  : Con → Set
Sub•  : Con• Δ → Con• Γ → Sub Δ Γ → Set
_∘•_  : Sub• Δ• Γ• γ → Sub• θ• Δ• δ →
        Sub• θ• Γ• (γ ∘ δ)
id•   : Sub• Γ• Γ• id
idl•  : id• ∘• γ• = γ•
                    \/
        \________/    :Sub• Δ• Γ• γ
      : Sub• Δ• Γ• (id ∘ γ)
```

Category, displayed category

```
Con   : Set
Sub   : Con → Con → Set
_∘_   : Sub Δ Γ → Sub Θ Δ → Sub Θ Γ
id    : Sub Γ Γ
idl   : id ∘ γ = γ
```

```
Con• : Con → Set
Sub• : Con• Δ → Con• Γ → Sub Δ Γ → Set
_∘•_ : Sub• Δ• Γ• γ → Sub• Θ• Δ• δ →
       Sub• Θ• Γ• (γ ∘ δ)
id•  : Sub• Γ• Γ• id
idl• : transp (Sub• Δ• Γ•) idl (id• ∘• γ•) = γ•
```

The canonicity displayed model

```
Con• Γ         := Sub ◇ Γ → Set
Sub• Δ• Γ• γ   := ∀{δ} → Δ• δ → Γ• (γ ∘ δ)
(γ• ∘• δ•) θ* := transp Γ• (ass ⁻¹) (γ• (δ• θ*))
id• γ*         := transp Γ• (idl ⁻¹) γ*
idl• :

transp (Sub• Δ• Γ•) idl (id• ∘• γ•)
```

The canonicity displayed model

```
Con• Γ         := Sub ◇ Γ → Set
Sub• Δ• Γ• γ   := ∀{δ} → Δ• δ → Γ• (γ ∘ δ)
(γ• ∘• δ•) θ* := transp Γ• (ass ⁻¹) (γ• (δ• θ*))
id• γ*         := transp Γ• (idl ⁻¹) γ*
idl• :

transp (λ γ . {δ◇ : Sub ◇ Δ} → Δ• δ◇ → Γ• (γ ∘ δ◇)) idl
  (λ{δ◇}δ*.transp Γ• (ass ⁻¹) (transp Γ• (idl ⁻¹) (γ• δ*)))
```

The canonicity displayed model

```
Con• Γ          := Sub ◇ Γ → Set
Sub• Δ• Γ• γ   := ∀{δ} → Δ• δ → Γ• (γ ∘ δ)
(γ• ∘• δ•) θ* := transp Γ• (ass ⁻¹) (γ• (δ• θ*))
id• γ*          := transp Γ• (idl ⁻¹) γ*
idl• :

transp (λ γ . {δ◇ : Sub ◇ Δ} → Δ• δ◇ → Γ• (γ ∘ δ◇)) idl
  (λ{δ◇}δ*.transp Γ• (idl ⁻¹ ▪ ass ⁻¹) (γ• δ*))
```

The canonicity displayed model

```
Con• Γ          := Sub ◇ Γ → Set
Sub• Δ• Γ• γ    := ∀{δ} → Δ• δ → Γ• (γ ∘ δ)
(γ• ∘• δ•) θ* := transp Γ• (ass ⁻¹) (γ• (δ• θ*))
id• γ*          := transp Γ• (idl ⁻¹) γ*
idl• :

λ{δ◇}.transp (λ γ . Δ• δ◇ → Γ• (γ ∘ δ◇)) idl
        (λ δ* . transp Γ• (idl ⁻¹ ■ ass ⁻¹) (γ• δ*))
```

The canonicity displayed model

```
Con• Γ          := Sub ◇ Γ → Set
Sub• Δ• Γ• γ   := ∀{δ} → Δ• δ → Γ• (γ ∘ δ)
(γ• ∘• δ•) θ* := transp Γ• (ass ⁻¹) (γ• (δ• θ*))
id• γ*          := transp Γ• (idl ⁻¹) γ*
idl• :

λδ*.transp (λ γ . Γ• (γ ∘ δ◇)) idl
      (transp Γ• (idl ⁻¹ ■ ass ⁻¹) (γ• δ*))
```

The canonicity displayed model

```
Con• Γ          := Sub ◇ Γ → Set
Sub• Δ• Γ• γ   := ∀{δ} → Δ• δ → Γ• (γ ∘ δ)
(γ• ∘• δ•) θ* := transp Γ• (ass ⁻¹) (γ• (δ• θ*))
id• γ*          := transp Γ• (idl ⁻¹) γ*
idl• :

λδ*.transp Γ• (cong (_∘ δ◇) idl)
      (transp Γ• (idl ⁻¹ ▪ ass ⁻¹) (γ• δ*))
```

The canonicity displayed model

```
Con• Γ          := Sub ◇ Γ → Set
Sub• Δ• Γ• γ    := ∀{δ} → Δ• δ → Γ• (γ ∘ δ)
(γ• ∘• δ•) θ* := transp Γ• (ass ⁻¹) (γ• (δ• θ*))
id• γ*          := transp Γ• (idl ⁻¹) γ*
idl• :

λδ*.transp Γ• (idl ⁻¹ ■ ass ⁻¹ ■ cong (_∘ δ◇) idl) (γ• δ*)
```

The canonicity displayed model

```
Con• Γ          := Sub ◇ Γ → Set
Sub• Δ• Γ• γ  := ∀{δ} → Δ• δ → Γ• (γ ∘ δ)
(γ• ∘• δ•) θ* := transp Γ• (ass ⁻¹) (γ• (δ• θ*))
id• γ*          := transp Γ• (idl ⁻¹) γ*
idl• :

λδ*.transp Γ• refl (γ• δ*)
```

The canonicity displayed model

```
Con• Γ         := Sub ◇ Γ → Set
Sub• Δ• Γ• γ   := ∀{δ} → Δ• δ → Γ• (γ ∘ δ)
(γ• ∘• δ•) θ* := transp Γ• (ass ⁻¹) (γ• (δ• θ*))
id• γ*         := transp Γ• (idl ⁻¹) γ*
idl• :

λδ*.γ• δ*
```

The canonicity displayed model

```
Con• Γ          := Sub ◇ Γ → Set
Sub• Δ• Γ• γ    := ∀{δ} → Δ• δ → Γ• (γ ∘ δ)
(γ• ∘• δ•) θ*   := transp Γ• (ass ⁻¹) (γ• (δ• θ*))
id• γ*          := transp Γ• (idl ⁻¹) γ*
idl• :

γ•
```

The canonicity displayed model, again

```
Con• Γ          := Sub ◇ Γ → Set
Sub• Δ• Γ• γ    := ∀{δ} → Δ• δ → Γ• (γ ∘ δ)
(γ• ∘• δ•) θ* := transp Γ• (ass ⁻¹) (γ• (δ• θ*))
id• γ*          := transp Γ• (idl ⁻¹) γ*
idl• :

transp (λ γ . {δ◇ : Sub ◇ Δ} → Δ• δ◇ → Γ• (γ ∘ δ◇)) idl
  (λ{δ◇}δ*.transp Γ• (ass ⁻¹) (transp Γ• (idl ⁻¹) (γ• δ*))) =
transp (λ γ . {δ◇ : Sub ◇ Δ} → Δ• δ◇ → Γ• (γ ∘ δ◇)) idl
  (λ{δ◇}δ*.transp Γ• (idl ⁻¹ ▪ ass ⁻¹) (γ• δ*)) =
λ {δ◇} . transp (λ γ . Δ• δ◇ → Γ• (γ ∘ δ◇)) idl
           (λ δ* . transp Γ• (idl ⁻¹ ▪ ass ⁻¹) (γ• δ*)) =
λ {δ◇} δ* . transp (λ γ . Γ• (γ ∘ δ◇)) idl
              (transp Γ• (idl ⁻¹ ▪ ass ⁻¹) (γ• δ*)) =
λ {δ◇} δ* . transp Γ• (cong (_∘ δ◇) idl)
              (transp Γ• (idl ⁻¹ ▪ ass ⁻¹) (γ• δ*)) =
λ {δ◇} δ* . transp Γ• (idl ⁻¹ ▪ ass ⁻¹ ▪ cong (_∘ δ◇) idl)
              (γ• δ*) =
λ {δ◇} δ* . transp Γ• refl (γ• δ*) =
λ {δ◇} δ* . γ• δ* =
γ•
```

Transport hell

- Escape: equality reflection
  - Agda doesn't know it
- Escape: shallow embedding
  - We define the displayed model over a strict model (standard model). The standard model is equationally complete.
  - Checks correctness, but is not an implementation.
- Decrease a lot: higher order abstract syntax (HOAS)
  - Presheaf internal language
  - Tricky to internalise induction principles
  - Agda doesn't know it
- Decrease: rewrite rules
  - Agda knows it
- Decrease: use cubical/observational metatheory
- Decrease: define some operations recursively

Recursive operations in the syntax

- Some QIITs are *definable* via normalisation (Nuo Li's thesis)
  - integers
  - syntax of first order logic (as a Cw2F with extra structure)
- For some QIITs, parts of the operations might be definable.

```
data Con : Set
data Sub : Con → Con → Set
data Ty  : Con → Set
_[_]     : Ty Γ → Sub Δ Γ → Ty Δ
(t $ u) [ γ ] = t [ γ ] $ u [ γ ]
```

  - Cubical Agda knows QIIRTs
  - QIIRTs have semantics in setoids using IIRTs
  - Easier way:
    1. Define QIIT
    2. Redefine some operations by induction on the QIIT
    3. Redefine the syntax using these (this will be partially strict), and prove its induction principle

Summary

- QIIRTs are interesting only in an intensional setting (except size).
- Match the traditional way of defining syntax using recursive substitution.
- Easy semantics: redefine parts of the syntax by recursion on the QIIT.