



Type Theory in Type Theory using Quotient Inductive Types

Thorsten Altenkirch, Ambrus Kaposi
University of Nottingham

POPL, St Petersburg, Florida
20 January 2016

Goal

- To represent the syntax of Type Theory inside Type Theory
- Why?
 - ▶ Study the metatheory in a nice language
 - ▶ Template type theory

Expressing the judgements of Type Theory

$$\Gamma \vdash t : A$$

will be formalised as

$$t' : Tm \Gamma A$$

(We are not interested in untyped terms)

Simple type theory in Agda (i)

```
data Ty      : Set where  
  ι          : Ty  
  _⇒_       : Ty → Ty → Ty  
data Con    : Set where  
  •         : Con  
  _',_     : Con → Ty → Con  
data Var    : Con → Ty → Set where  
  zero     : Var (Γ , A) A  
  suc      : Var Γ A → Var (Γ , B) A  
data Tm     : Con → Ty → Set where  
  var      : Var Γ A → Tm Γ A  
  app     : Tm Γ (A ⇒ B) → Tm Γ A → Tm Γ B  
  lam     : Tm (Γ , A) B → Tm Γ (A ⇒ B)
```

Simple type theory in Agda (ii)

- In addition, we need substitutions:

$$\begin{aligned} \text{Tms} &: \text{Con} \rightarrow \text{Con} \rightarrow \text{Set} \\ _[_] &: \text{Tm } \Gamma \text{ A} \rightarrow \text{Tms } \Delta \Gamma \rightarrow \text{Tm } \Delta \text{ A} \end{aligned}$$

- Now we can define a conversion relation:

$$_ \sim _ : \text{Tm } \Gamma \text{ A} \rightarrow \text{Tm } \Gamma \text{ A} \rightarrow \text{Set}$$

eg. $\text{app } (\text{lam } t) \text{ u} \sim t \text{ [id , u]}$

- The intended syntax is a quotient:

$$\text{Tm } \Gamma \text{ A} / \sim$$

The syntax of Dependent Type Theory (i)

- Types depend on contexts
- Substitutions are mentioned in the application rule:
$$\text{app} : \text{Tm } \Gamma (\Pi A B) (a : \text{Tm } \Gamma A) \rightarrow \text{Tm } \Gamma (B [a])$$
- We need an inductive-inductive definition:

data Con : Set

data Ty : Con \rightarrow Set

data Tms : Con \rightarrow Con \rightarrow Set

data Tm : (Γ : Con) \rightarrow Ty Γ \rightarrow Set

The syntax of Dependent Type Theory (ii)

- In addition, there is a coercion rule for terms:

$$\frac{\Gamma \vdash A \sim B \quad \Gamma \vdash t : A}{\Gamma \vdash t : B}$$

- This forces us to define conversion relations mutually:

```
data Con      : Set
data Ty      : Con → Set
data Tms     : Con → Con → Set
data Tm      : (Γ : Con) → Ty Γ → Set
data _~Con_  : Con → Con → Set
data _~Ty_   : Ty Γ → Ty Γ → Set
data _~Tms_  : Tms Δ Γ → Tms Δ Γ → Set
data _~Tm_   : Tm Γ A → Tm Γ A → Set
```

Lots of boilerplate

- The \sim_X relations are equivalence relations
- Coercion rules
- Congruence rules
- We need to work with setoids

The identity type $_ \equiv _$

- Equality (the identity type) is an equivalence relation
- We can coerce between equal types
- Equality is a congruence
- What about the extra equalities (eg. β, η for Π)?

Higher inductive types

- An idea from homotopy type theory: constructors for equalities.
- Example:

```
data I      : Set where  
  zero     : I  
  one      : I  
  segment  : zero  $\equiv$  one
```

Higher inductive types

- An idea from homotopy type theory: constructors for equalities.
- Example:

data I : Set **where**

zero : I

one : I

segment : zero \equiv one

Recl : (I^M : Set)

(zero^M : I^M)

(one^M : I^M)

(segment^M : zero^M \equiv one^M)

$\rightarrow I \rightarrow I^M$

Quotient inductive types (QITs)

- A higher inductive type which is truncated to an h-set.
- They are *not* the same as quotient types: equality constructors are defined at the same time
- QITs can be simulated in Agda

The syntax of Dependent Type Theory (iii)

- We defined the syntax of a basic Type Theory as a quotient inductive inductive type (with Π and an uninterpreted family of types U, El)
- We don't need to state the equivalence relation, coercion, congruence laws anymore
- We collect the arguments of the recursor into a record:

```
record Model : Set where  
  field ConM : Set  
        TyM   : ConM → Set  
        ...
```

- which is the type of algebras for the QIT
= the type of models of Type Theory, close to CwF

Applications (i): standard model

- A sanity check
- Every syntactic construct is interpreted as the corresponding metatheoretic construction.

$$\begin{aligned} \text{Con}^M &= \text{Set} \\ \top y^M \llbracket \Gamma \rrbracket &= \llbracket \Gamma \rrbracket \rightarrow \text{Set} \\ \Pi^M \llbracket A \rrbracket \llbracket B \rrbracket \gamma &= (x : \llbracket A \rrbracket \gamma) \rightarrow \llbracket B \rrbracket (\gamma, x) \\ \text{lam}^M \llbracket t \rrbracket \gamma &= \lambda x \rightarrow \llbracket t \rrbracket (\gamma, x) \\ \dots & \end{aligned}$$

Applications (ii): logical predicate interpretation

- An interpretation from the syntax into the syntax
- Bernardy-Jansson-Paterson: Parametricity and Dependent Types, 2012
- A type is interpreted as a logical predicate over that type
- A term is interpreted as a proof that it satisfies the predicate
- Automated derivation of free theorems

Applications (iii): presheaf model

- Given a category \mathcal{C}
- Contexts are presheaves over \mathcal{C}
- Types are families of presheaves, terms are sections
- Normalisation by evaluation (NBE):
 - ▶ A presheaf over the category of renamings
 - ▶ We can generalise NBE from Simple Type Theory to Type Theory (formalisation in progress)

Further work

- We internalized a very basic type theory, but this can be extended easily with universes and inductive types.
- We used axioms (quotient inductive types, functional extensionality) in our metatheory. This can be solved by using cubical type theory.
- If we work within HoTT, we can only eliminate into h-sets. Hence, the standard model doesn't work as described.

Template type theory

- Given a model of type theory, together with new constants in that model
- We can interpret code that uses the new constants inside the model
- The code can use all the conveniences such as implicit arguments, pattern matching etc.
- This way we can justify extensions of type theory:
 - ▶ guarded type theory
 - ▶ local state monad
 - ▶ parametricity
 - ▶ homotopy type theory