

Representing type theory in type theory with inspiration from natural models

Fredrik Nordvall Forsberg
University of Strathclyde, Glasgow, Scotland

Joint work Liang-Ting Chen and Tzu-Chun Tsai

Types and Topology

17 December 2025

Motivation

As a general-purpose foundation of mathematics, type theory should be able to represent its own syntax.

However ergonomic representations of well typed terms only (“no junk”) have turned out to be a long-standing open problem.

Practical uses: formalising metatheory, developing metaprogramming.

Representing simple type theory is simple

First we inductively define $Ty : Set$:

$$\begin{aligned} N &: Ty \\ _ \Rightarrow _ &: Ty \rightarrow Ty \rightarrow Ty \end{aligned}$$

Representing simple type theory is simple

First we inductively define $Ty : Set$:

$$\begin{aligned} N &: Ty \\ _ \Rightarrow _ &: Ty \rightarrow Ty \rightarrow Ty \end{aligned}$$

Then we define $Ctx : Set$ as a (backwards) list of types:

$$\begin{aligned} \cdot &: Ctx \\ _, _ &: Ctx \rightarrow Ty \rightarrow Ctx \end{aligned}$$

Representing simple type theory is simple

First we inductively define $Ty : Set$:

$$\begin{aligned} N &: Ty \\ _ \Rightarrow _ &: Ty \rightarrow Ty \rightarrow Ty \end{aligned}$$

Then we define $Ctx : Set$ as a (backwards) list of types:

$$\begin{aligned} \cdot &: Ctx \\ _, _ &: Ctx \rightarrow Ty \rightarrow Ctx \end{aligned}$$

Afterwards, we can inductively define well typed terms in a given context

$Tm : Ctx \rightarrow Ty \rightarrow Set$, e.g.

$$\text{lam} : Tm (\Gamma, S) T \rightarrow Tm \Gamma (S \Rightarrow T)$$

Representing dependent type theory is dependent

Dependent types are more challenging:

Representing dependent type theory is dependent

Dependent types are more challenging:

- ▶ Types only make sense in context (is `Fin n` a type?).

Representing dependent type theory is dependent

Dependent types are more challenging:

- ▶ Types only make sense in context (is `Fin n` a type?).
- ▶ Terms depends on types.

Representing dependent type theory is dependent

Dependent types are more challenging:

- ▶ Types only make sense in context (is `Fin n` a type?).
- ▶ Terms depends on types.
- ▶ Types depends on terms.
 - ↪ All sorts depend on each other.

Representing dependent type theory is dependent

Dependent types are more challenging:

- ▶ Types only make sense in context (is `Fin n` a type?).
- ▶ Terms depends on types.
- ▶ Types depends on terms.
 - ↪ All sorts depend on each other.
- ▶ Definitional equality between types (`refl : 2 = 1 + 1?`).

Representing dependent type theory is dependent

Dependent types are more challenging:

- ▶ Types only make sense in context (is $\text{Fin } n$ a type?).
- ▶ Terms depends on types.
- ▶ Types depends on terms.
 - ↪ All sorts depend on each other.
- ▶ Definitional equality between types ($\text{refl} : 2 = 1 + 1?$).
- ▶ Substitutions needed for typing rules (e.g. $f a : B[a/x]$).
 - ↪ Substitutions and reductions in the syntax.

History

McKinna and Pollack [1999] represented Pure Type Systems in Lego.

- ▶ Untyped terms later refined by a typing relation (removing “junk” by hand).

History

McKinna and Pollack [1999] represented Pure Type Systems in Lego.

- ▶ Untyped terms later refined by a typing relation (removing “junk” by hand).

Danielsson [2007] represented type theory in AgdaLight.

- ▶ Explicit equality relation, and proves that all constructions respect it (an instance of “setoid hell”).

History

McKinna and Pollack [1999] represented Pure Type Systems in Lego.

- ▶ Untyped terms later refined by a typing relation (removing “junk” by hand).

Danielsson [2007] represented type theory in AgdaLight.

- ▶ Explicit equality relation, and proves that all constructions respect it (an instance of “setoid hell”).

Chapman [2009] represented type theory in Agda 2.

- ▶ Foundationally more straightforward (“only” inductive-inductive definitions).
- ▶ Still an explicit equality relation, with manual proof effort.

History

McKinna and Pollack [1999] represented Pure Type Systems in Lego.

- ▶ Untyped terms later refined by a typing relation (removing “junk” by hand).

Danielsson [2007] represented type theory in AgdaLight.

- ▶ Explicit equality relation, and proves that all constructions respect it (an instance of “setoid hell”).

Chapman [2009] represented type theory in Agda 2.

- ▶ Foundationally more straightforward (“only” inductive-inductive definitions).
- ▶ Still an explicit equality relation, with manual proof effort.

Altenkirch and Kaposi [2016] used **quotient**-inductive-inductive types.

- ▶ Definitional equality in object theory is “real” equality in host theory.
- ▶ All constructions automatically respect object equality.
- ▶ Cannot treat internally equal terms differently (**a feature!**).

The initial Category with Families

One way to understand [Altenkirch and Kaposi](#)'s construction is as the initial *category with families*, represented as a QIT.

[Dybjer \[1996\]](#) introduced categories with families (CwFs) as a notion of model of type theory. Basic ingredients:

- ▶ Category \mathcal{C} (“contexts”)
- ▶ Functor $(\text{Ty}, \text{Tm}) : \mathcal{C}^{\text{op}} \rightarrow \text{Fam Set}$
- ▶ Context extension $_ , _ : (\Gamma : \mathcal{C}) \rightarrow \text{Ty}(\Gamma) \rightarrow \mathcal{C}$ with universal property.

The initial Category with Families

One way to understand [Altenkirch and Kaposi](#)'s construction is as the initial *category with families*, represented as a QIT.

[Dybjer \[1996\]](#) introduced categories with families (CwFs) as a notion of model of type theory. Basic ingredients:

- ▶ Category \mathcal{C} (“contexts”)
- ▶ Functor $(\text{Ty}, \text{Tm}) : \mathcal{C}^{\text{op}} \rightarrow \text{Fam Set}$
- ▶ Context extension $_, _ : (\Gamma : \mathcal{C}) \rightarrow \text{Ty}(\Gamma) \rightarrow \mathcal{C}$ with universal property.

The theory of CwFs is a *generalised algebraic theory* [[Cartmell 1986](#), [Bezem, Coquand, Dybjer and Escardó 2021](#)] and so has an initial model (say with extensional type theory as metatheory).

The initial CwF, in practice

Some CwF equations are only well typed because of earlier equations, e.g.:

$$A[\text{id}]_{\mathcal{T}} = A$$

$$\vdots$$

$$t[\text{id}]_t = t$$

where $t : \mathbb{T}m \Gamma A$ and $t[\text{id}]_t : \mathbb{T}m \Gamma A[\text{id}]_{\mathcal{T}}$. As a QIIT definition, we turn to explicit transports:

$$[\text{id}]_t : \text{transport}(\mathbb{T}m \Gamma, [\text{id}]_{\mathcal{T}}, t[\text{id}]_t) =_{\mathbb{T}m \Gamma A} t$$

The initial CwF, in practice

Some CwF equations are only well typed because of earlier equations, e.g.:

$$\begin{aligned} A[\text{id}]_{\mathcal{T}} &= A \\ &\vdots \\ t[\text{id}]_t &= t \end{aligned}$$

where $t : \mathbb{T}m \Gamma A$ and $t[\text{id}]_t : \mathbb{T}m \Gamma A[\text{id}]_{\mathcal{T}}$. As a QIIT definition, we turn to explicit transports:

$$[\text{id}]_t : \text{transport}(\mathbb{T}m \Gamma, [\text{id}]_{\mathcal{T}}, t[\text{id}]_t) =_{\mathbb{T}m \Gamma A} t$$

Unfortunately, such **transports** make Cubical Agda (erroneously) reject the definition as not strictly positive.

The initial CwF, in practice

Some CwF equations are only well typed because of earlier equations, e.g.:

$$\begin{aligned} A[\text{id}]_{\mathcal{T}} &= A \\ &\vdots \\ t[\text{id}]_t &= t \end{aligned}$$

where $t : \text{Tm } \Gamma A$ and $t[\text{id}]_t : \text{Tm } \Gamma A[\text{id}]_{\mathcal{T}}$. As a QIIT definition, we turn to explicit transports:

$$[\text{id}]_t : \text{transport}(\text{Tm } \Gamma, [\text{id}]_{\mathcal{T}}, t[\text{id}]_t) =_{\text{Tm } \Gamma A} t$$

Unfortunately, such **transports** make Cubical Agda (erroneously) reject the definition as not strictly positive. In this case we could use **PathP** instead, but that is rather Cubical Type Theory specific, and hence not satisfactory.

Another attempt

Why do we need **transport**, in general?

It is because we demand precise types, e.g. in

$$_,_ : (\sigma : \text{Sub } \Gamma \Delta) \rightarrow (t : \text{Tm } \Gamma (A[\sigma]_{\mathcal{T}})) \rightarrow \text{Sub } \Gamma (\Delta, A)$$

we insist that t is of a certain type.

Another attempt

Why do we need **transport**, in general?

It is because we demand precise types, e.g. in

$$_,_ : (\sigma : \text{Sub } \Gamma \Delta) \rightarrow (t : \text{Tm } \Gamma (A[\sigma]_{\mathcal{T}})) \rightarrow \text{Sub } \Gamma (\Delta, A)$$

we insist that t is of a certain type. Equivalently, we could ask for

$$_,_ : [_] : (\sigma : \text{Sub } \Gamma \Delta) \rightarrow (t : \text{Tm } \Gamma B) \rightarrow B \equiv A[\sigma]_{\mathcal{T}} \rightarrow \text{Sub } \Gamma (\Delta, A)$$

instead — turning $(\sigma, \text{transport}(\text{Tm } \Gamma, p, t))$ into $(\sigma, t : [p])$ (without **transport!**).

Another attempt

Why do we need **transport**, in general?

It is because we demand precise types, e.g. in

$$_,_ : (\sigma : \text{Sub } \Gamma \Delta) \rightarrow (t : \text{Tm } \Gamma (A[\sigma]_{\mathcal{T}})) \rightarrow \text{Sub } \Gamma (\Delta, A)$$

we insist that t is of a certain type. Equivalently, we could ask for

$$_,_ : [_] : (\sigma : \text{Sub } \Gamma \Delta) \rightarrow (t : \text{Tm } \Gamma B) \rightarrow B \equiv A[\sigma]_{\mathcal{T}} \rightarrow \text{Sub } \Gamma (\Delta, A)$$

instead — turning $(\sigma, \text{transport}(\text{Tm } \Gamma, p, t))$ into $(\sigma, t : [p])$ (without **transport!**).

But if we do this everywhere, there is no need to keep the index B locally anymore; instead we can change the type of **Tm** to $\text{Tm} : \text{Ctx} \rightarrow \text{Set}$, and introduce a function **tyOf** : $\text{Tm } \Gamma \rightarrow \text{Ty } \Gamma$ to compute the types of terms.

A QIIR representation of the syntax of type theory

We simultaneously define (changes to QIIT definition **highlighted**)

```
data Ctx : Type
data Sub : (Γ : Ctx) → (Δ : Ctx) → Set
data Ty  : (Γ : Ctx) → Set
data Tm  : (Γ : Ctx) → Set
tyOf : Tm Γ → Ty Γ
```

Since `tyOf` is defined recursively, this is a *quotient-inductive-inductive-recursive* definition. (By saying “: `Set`”, we mean that we add implicit set truncations, hence quotients rather than higher types.)

Note: This is reminiscent of Fiore [2012] and Awodey [2016]’s *natural models* formulation of CwFs.

The substitution calculus as a QIIRT

data _where

$\emptyset : \text{Ctx}$

$_,_ : (\Gamma : \text{Ctx})(A : \text{Ty } \Gamma) \rightarrow \text{Ctx}$

$_[_]_{\mathcal{T}} : (A : \text{Ty } \Delta)(\sigma : \text{Sub } \Gamma \Delta) \rightarrow \text{Ty } \Gamma$

$_[_]_t : (t : \text{Tm } \Delta)(\sigma : \text{Sub } \Gamma \Delta) \rightarrow \text{Tm } \Gamma$

$\emptyset : \text{Sub } \Gamma \emptyset$

$_,_ : [_] : (\sigma : \text{Sub } \Gamma \Delta)(t : \text{Tm } \Gamma) \rightarrow$
 $\text{tyOf } t \equiv A[\sigma]_{\mathcal{T}} \rightarrow \text{Sub } \Gamma(\Delta, A)$

id : Sub $\Gamma \Gamma$

$_ \circ _ : \text{Sub } \Delta \Theta \rightarrow \text{Sub } \Gamma \Delta \rightarrow \text{Sub } \Gamma \Theta$

$\pi_1 : \text{Sub } \Gamma(\Delta, A) \rightarrow \text{Sub } \Gamma \Delta$

$\pi_2 : \text{Sub } \Gamma(\Delta, A) \rightarrow \text{Tm } \Gamma$

id \circ $_ : \text{id} \circ \sigma \equiv \sigma$

$_ \circ \text{id} : \sigma \circ \text{id} \equiv \sigma$

assoc : $(\gamma \circ \tau) \circ \sigma \equiv \gamma \circ (\tau \circ \sigma)$

$[_]_{\mathcal{T}} : A[\tau]_{\mathcal{T}}[\sigma]_{\mathcal{T}} \equiv A[\tau \circ \sigma]_{\mathcal{T}}$

$[\text{id}]_{\mathcal{T}} : A \equiv A[\text{id}]_{\mathcal{T}}$

$[\text{id}]_t : t \equiv t[\text{id}]_t$

$[\circ]_t : t[\tau]_t[\sigma]_t \equiv t[\tau \circ \sigma]_t$

$[\circ] : (q : \text{tyOf}(t[\tau]_t) \equiv A[\sigma \circ \tau]_{\mathcal{T}})$
 $\rightarrow (\sigma, t : [pt]) \circ \tau \equiv (\sigma \circ \tau, t[\tau]_t : [qt])$

$\text{tyOf}(\pi_2 \{A = A\} \sigma) = A[\pi_1 \sigma]_{\mathcal{T}}$

data _where

$\eta\pi : \sigma \equiv (\pi_1 \sigma, \pi_2 \sigma : [\text{refl}])$

$\eta\emptyset : \sigma \equiv \emptyset$

$\beta\pi_1 : \pi_1(\sigma, t : [p]) \equiv \sigma$

$\beta\pi_2 : (q : A[\pi_1(\sigma, t : [p])]_{\mathcal{T}} \equiv \text{tyOf } t)$
 $\rightarrow \pi_1(\sigma, t : [p]) \equiv t$

$\text{tyOf}(\beta\pi_2 q i) = q i$

$\text{tyOf}(t[\sigma]_t) = (\text{tyOf } t)[\sigma]_{\mathcal{T}}$

$\text{tyOf}([\text{id}]_t i) = [\text{id}]_{\mathcal{T}} i$

$\text{tyOf}([\circ]_t i) = [\circ]_{\mathcal{T}} i$

The substitution calculus as a QIIRT

data _where

$\emptyset : \text{Ctx}$

$_,_ : (\Gamma : \text{Ctx})(A : \text{Ty } \Gamma) \rightarrow \text{Ctx}$

$_[_]_{\mathcal{T}} : (A : \text{Ty } \Delta)(\sigma : \text{Sub } \Gamma \Delta) \rightarrow \text{Ty } \Gamma$

$_[_]_t : (t : \text{Tm } \Delta)(\sigma : \text{Sub } \Gamma \Delta) \rightarrow \text{Tm } \Gamma$

$\emptyset : \text{Sub } \Gamma \emptyset$

$_,_ : [_] : (\sigma : \text{Sub } \Gamma \Delta)(t : \text{Tm } \Gamma) \rightarrow$
 $\text{tyOf } t \equiv A[\sigma]_{\mathcal{T}} \rightarrow \text{Sub } \Gamma(\Delta, A)$

$[\text{id}]_{\mathcal{T}} : A \equiv A[\text{id}]_{\mathcal{T}}$

$[\text{id}]_t : t \equiv t[\text{id}]_t$

$[\circ]_t : t[\tau]_t[\sigma]_t \equiv t[\tau \circ \sigma]_t$

$[\circ] : (q : \text{tyOf}(t[\tau]_t) \equiv A[\sigma \circ \tau]_{\mathcal{T}})$
 $\rightarrow (\sigma, t : [pt]) \circ \tau \equiv (\sigma \circ \tau, t[\tau]_t : [qt])$

$\text{tyOf}(\pi_2 \{A = A\} \sigma) = A[\pi_1 \sigma]_{\mathcal{T}}$

data _where

$\eta\pi : \sigma \equiv (\pi_1 \sigma, \pi_2 \sigma : [\text{refl}])$

$\eta\emptyset : \sigma \equiv \emptyset$

$\beta\pi_1 : \pi_1(\sigma, t : [p]) \equiv \sigma$

$\beta\pi_2 : (q : A[\pi_1(\sigma, t : [p])]_{\mathcal{T}} \equiv \text{tyOf } t)$
 $\rightarrow \pi_1(\sigma, t : [p]) \equiv t$

$\text{tyOf}(\beta\pi_2 q i) = q i$

$\text{tyOf}(t[\sigma]_t) = (\text{tyOf } t)[\sigma]_{\mathcal{T}}$

$\text{tyOf}([\text{id}]_t i) = [\text{id}]_{\mathcal{T}} i$

$\text{tyOf}([\circ]_t i) = [\circ]_{\mathcal{T}} i$

What is different?

1. **tyOf** constraint in $_,_ : [_]$.
2. No **transport** in $[\text{id}]_t$ and $[\circ]_t$.
3. Derivable arguments q in $[\circ]$ and $\beta\pi_2$.
4. Interleaving definition of **tyOf** $(\pi_2 \sigma)$.

Other type formers

In the same way, we can introduce other type formers such as Π -types, inductive types such as the Booleans \mathbb{B} , and a universe $(\mathbf{U}, \mathbf{EI})$.

Other type formers

In the same way, we can introduce other type formers such as Π -types, inductive types such as the Booleans \mathbb{B} , and a universe $(\mathbf{U}, \mathbf{El})$.

To avoid Cubical Agda complaining about strict positivity problems, we often found it useful to include “superfluous” `tyOf` proofs in the definition, rather than constructing them from other pieces, e.g.

$$\mathbb{B}[]_2 : \text{tyOf} (\pi_2 \{\Gamma, \mathbb{B}\} \text{id}) \equiv B [\tau]_{\mathcal{T}}$$

Other type formers

In the same way, we can introduce other type formers such as Π -types, inductive types such as the Booleans \mathbb{B} , and a universe $(\mathbf{U}, \mathbf{El})$.

To avoid Cubical Agda complaining about strict positivity problems, we often found it useful to include “superfluous” `tyOf` proofs in the definition, rather than constructing them from other pieces, e.g.

$$\mathbb{B}[]_2 : \text{tyOf} (\pi_2 \{\Gamma, \mathbb{B}\} \text{id}) \equiv B [\tau]_{\mathcal{T}}$$

Since `Ty Γ` is a set by construction, $\mathbb{B}[]_2$ is equal to the canonical proof of the same fact anyway.

Elimination principles

As expected, we can use pattern matching to define recursion- and induction principles, thus witnessing that the syntax is the initial model.

Elimination principles

As expected, we can use pattern matching to define recursion- and induction principles, thus witnessing that the syntax is the initial model.

Annoyingly, we have to mark the definitions as `TERMINATING`, even though recursive calls are on structurally smaller arguments — possibly because of the simultaneous proof

$$\text{recTyOf} : S.\text{tyOf } t \equiv B \rightarrow \llbracket \text{tyOf} \rrbracket (\text{recTm } t) \equiv \text{recTy } B$$

Elimination principles

As expected, we can use pattern matching to define recursion- and induction principles, thus witnessing that the syntax is the initial model.

Annoyingly, we have to mark the definitions as `TERMINATING`, even though recursive calls are on structurally smaller arguments — possibly because of the simultaneous proof

$$\text{recTyOf} : S.\text{tyOf } t \equiv B \rightarrow \llbracket \text{tyOf} \rrbracket (\text{recTm } t) \equiv \text{recTy } B$$

Surprisingly, it is actually better to also let users define methods corresponding to superfluous equality constructors, because this sometimes allows stricter definitions.

Constructing models

Using the elimination principle, we can construct the standard **Set** model where

$$\llbracket \text{Ctx} \rrbracket = \text{Type}$$

$$\llbracket \text{Ty} \rrbracket \Gamma = \Gamma \rightarrow \text{Type}$$

$$\llbracket \text{Sub} \rrbracket \Gamma \Delta = \Gamma \rightarrow \Delta$$

$$\llbracket \text{Tm} \rrbracket \Gamma = (\Sigma A : \Gamma \rightarrow \text{Type})((\gamma : \Gamma) \rightarrow A \gamma)$$

$$\llbracket \text{tyOf} \rrbracket (A, t) = A$$

if we assume UIP so that $\llbracket \text{Ty} \rrbracket \Gamma$ is a set.

Constructing models

Using the elimination principle, we can construct the standard **Set** model where

$$\begin{aligned} \llbracket \text{Ctx} \rrbracket &= \text{Type} \\ \llbracket \text{Ty} \rrbracket \Gamma &= \Gamma \rightarrow \text{Type} \\ \llbracket \text{Sub} \rrbracket \Gamma \Delta &= \Gamma \rightarrow \Delta \\ \llbracket \text{Tm} \rrbracket \Gamma &= (\Sigma A : \Gamma \rightarrow \text{Type})((\gamma : \Gamma) \rightarrow A \gamma) \\ \llbracket \text{tyOf} \rrbracket (A, t) &= A \end{aligned}$$

if we assume UIP so that $\llbracket \text{Ty} \rrbracket \Gamma$ is a set. Similarly we can define the term model

$$\begin{aligned} \llbracket \text{Ctx} \rrbracket &= \text{Ctx} \\ \llbracket \text{Ty} \rrbracket &= \text{Ty} \\ &\vdots \end{aligned}$$

Model constructions

Normalisation by Evaluation

NbE was surprisingly easy to implement, and actually computes in Cubical Agda.

Model constructions

Normalisation by Evaluation

NbE was surprisingly easy to implement, and actually computes in Cubical Agda.

Logical predicates model

The logical predicates displayed model interprets types over A as

$$\mathsf{Ty}^P \Gamma A = \mathsf{Ty} (\Gamma, A)$$

(suitably Kripke-ified). This brings us back to the same transport hell that we were trying to escape from for $\mathsf{Tm} \Gamma A$.

Model constructions

Normalisation by Evaluation

NbE was surprisingly easy to implement, and actually computes in Cubical Agda.

Logical predicates model

The logical predicates displayed model interprets types over A as

$$\mathsf{Ty}^P \Gamma A = \mathsf{Ty}(\Gamma, A)$$

(suitably Kripke-ified). This brings us back to the same transport hell that we were trying to escape from for $\mathsf{Tm} \Gamma A$.

Strictification

Kaposi and Pujet [2025] show how to strictify the category laws and functor laws of a given CwF in the QIIT formulation, and similar ideas apply also to our QIIRT definition.

(Notably, this requires contexts to form a set, which they do for the syntax.)

Model constructions

Normalisation by Evaluation

NbE was surprisingly easy to implement, and actually computes in Cubical Agda.

Logical predicates model

The logical predicates displayed model interprets types over A as

$$\mathsf{Ty}^P \Gamma A = \mathsf{Ty}(\Gamma, A)$$

(suitably Kripke-ified). This brings us back to the same transport hell that we were trying to escape from for $\mathsf{Tm} \Gamma A$.

Strictification

Kaposi and Pujet [2025] show how to strictify the category laws and functor laws of a given CwF in the QIIT formulation, and similar ideas apply also to our QIIRT definition.

(Notably, this requires contexts to form a set, which they do for the syntax.)

However, this only makes part of the model strict, and does not solve e.g. our logical predicates model issue.


Summary and conclusions

We developed a representation of the syntax of type theory in type theory inspired by natural models, with a typing function $\text{tyOf} : \text{Tm } \Gamma \rightarrow \text{Ty } \Gamma$.

This formulation leads to fewer **transports** in the definition of the syntax, which in turns makes it easier for Cubical Agda to accept the definition as strictly positive.

However, many uses of **transport** have a tendency to come back when defining concrete models or model constructions.

Finally, the definitional principle of QIIRT is not well understood — we hope this case study is a good motivation to study such definitions further.

 **Can We Formalise Type Theory Intrinsically without Any Compromise?**
Liang-Ting Chen, Fredrik Nordvall Forsberg, and Tzu-Chun Tsai
CPP 2026. Formalisation: <https://github.com/L-TChen/TTasQIIRT>.

Summary and conclusions


We developed a representation of the syntax of type theory in type theory inspired by natural models, with a typing function $\text{tyOf} : \text{Tm } \Gamma \rightarrow \text{Ty } \Gamma$.

This formulation leads to fewer **transports** in the definition of the syntax, which in turns makes it easier for Cubical Agda to accept the definition as strictly positive.

However, many uses of **transport** have a tendency to come back when defining

Thank you, and congratulations Martín!

case study is a good motivation to study such definitions further.

 Can We Formalise Type Theory Intrinsically without Any Compromise?
Liang-Ting Chen, Fredrik Nordvall Forsberg, and Tzu-Chun Tsai
CPP 2026. Formalisation: <https://github.com/L-TChen/TTasQIIRT>.

References

- ▶ Thorsten Altenkirch and Ambrus Kaposi. 2016. “Type theory in type theory using quotient inductive types”. In *Principles of Programming Languages (POPL '16)*, 18–29. DOI: 10.1145/2837614.2837638.
- ▶ Steve Awodey. 2016. “Natural models of homotopy type theory”. *Mathematical Structures in Computer Science*. DOI: 10.1017/S0960129516000268.
- ▶ Marc Bezem, Thierry Coquand, Peter Dybjer and Martín Escardó. 2021, “On generalized algebraic theories and categories with families”. In *Mathematical Structures in Computer Science*. DOI: 10.1017/S0960129521000268.
- ▶ James Chapman. 2009. “Type Theory Should Eat Itself”. In *Workshop on Mathematically Structured Functional Programming (MSFP '06)*, 21–36. DOI: 10.1016/j.entcs.2008.12.114.
- ▶ Nils Anders Danielsson. 2007. “A formalisation of a dependently typed language as an inductive-recursive family”. In *Types for Proofs and Programs (TYPES '06)*. DOI: 10.1007/978-3-540-74464-1_7.
- ▶ Peter Dybjer. 1996. “Internal type theory”. In *Types for Proofs and Programs (TYPES '95)*. 120–134. DOI: 10.1007/3-540-61780-9_66
- ▶ Marcelo Fiore. 2012. “Discrete generalised polynomial functors”. Slides from talk at ICALP '12. <https://www.cl.cam.ac.uk/~mpf23/talks/ICALP2012.pdf>
- ▶ Ambrus Kaposi and Loïc Pujet. 2025. “Type theory in type theory using a strictified syntax”. In *International Conference on Functional Programming (ICFP '25)*, 855–885. DOI: 10.1145/3747535.