Can we formalise type theory intrinsically without any compromise?

A case study in Cubical Agda

Liang-Ting Chen[1] **Fredrik Nordvall Forsberg**[2] Tzu-Chun Tsai[1]

[1] Academia Sinica [2] University of Strathclyde

Certified Programs and Proofs 2026, Rennes, France

13 January 2026

Motivation

Formalised metatheory, and more generally metaprogramming, require an internal representation of the syntax of type theory.

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

Goal: A type $\mathsf{Tm}\ \Gamma\ A$ whose elements are terms of type A in context Γ, which is convenient to work with.

Why are intrinsic representations hard?

Why are intrinsic representations hard?

- Dependent types only make sense in context (is Fin n a type?).

Why are intrinsic representations hard?

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

Why are intrinsic representations hard?

- Dependent 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.

Why are intrinsic representations hard?

- Dependent types only make sense in context (is Fin n a type?).
- Terms depends on types.
- Types depends on terms.
  - $\rightsquigarrow$ All sorts depend on each other.
- Definitional equality between types (what are terms of type Fin $(1 + 1)$?).

Why are intrinsic representations hard?

- Dependent types only make sense in context (is Fin n a type?).
- Terms depends on types.
- Types depends on terms.
  - $\rightsquigarrow$ All sorts depend on each other.
- Definitional equality between types (what are terms of type Fin $(1+1)$?).
- Substitutions needed for typing rules (e.g. $f\ a : B[a/x]$).
  - $\rightsquigarrow$ 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" (prop.) 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* [Dybjer 1996], represented as a QIIT.

The initial Category with Families

One way to understand Altenkirch and Kaposi's construction is as the initial *category with families* [Dybjer 1996], represented as a QIIT.

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

The initial Category with Families

One way to understand Altenkirch and Kaposi's construction is as the initial *category with families* [Dybjer 1996], represented as a QIIT.

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

Cubical Agda has support for QIITs (and more), so we could hope to formalise this construction in it. (Altenkirch and Kaposi's formalisation pre-dates Cubical Agda, and used Licata's Trick [2011] with postulates in standard Agda.)

The initial CwF, in practice

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

$$A[\mathsf{id}]_T = A$$

$$\vdots$$

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

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

$$[\mathsf{id}]_t : \mathsf{transport}(\mathsf{Tm}\,\Gamma, [\mathsf{id}]_T, t[\mathsf{id}]_t) =_{\mathsf{Tm}\,\Gamma\,A} t$$

The initial CwF, in practice

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

$$A[\mathrm{id}]_T = A$$

$$\vdots$$

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

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

$$[\mathrm{id}]_t : \mathrm{transport}(\mathrm{Tm}\,\Gamma, [\mathrm{id}]_T, t[\mathrm{id}]_t) =_{\mathrm{Tm}\,\Gamma\,A} t$$

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.:

$$A[\mathrm{id}]_T = A$$

$$\vdots$$

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

where $t : \mathsf{Tm}\,\Gamma\,A$ and $t[\mathrm{id}]_t : \mathsf{Tm}\,\Gamma\,A[\mathrm{id}]_T$. As a QIIT definition, we turn to explicit transports:

$$[\mathrm{id}]_t : \mathrm{transport}(\mathsf{Tm}\,\Gamma, [\mathrm{id}]_T, t[\mathrm{id}]_t) =_{\mathsf{Tm}\,\Gamma\,A} t$$

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

$$\_,\_ : (\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \to (t : \mathsf{Tm}\,\Gamma\,(A\,[\sigma]_T)) \to \mathsf{Sub}\,\Gamma\,(\Delta, A)$$

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

$$\_,\_ : (\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \to (t : \mathsf{Tm}\,\Gamma\,(A\,[\sigma]_T)) \to \mathsf{Sub}\,\Gamma\,(\Delta, A)$$

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

$$\_,\_:[\_] : (\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \to (t : \mathsf{Tm}\,\Gamma\,B) \to B \equiv A\,[\sigma]_T \to \mathsf{Sub}\,\Gamma\,(\Delta, A)$$

instead — turning $(\sigma, \mathsf{transport}(\mathsf{Tm}\,\Gamma, p, t))$ 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

$$\_,\_ : (\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \to (t : \mathsf{Tm}\,\Gamma\,(A\,[\sigma]_T)) \to \mathsf{Sub}\,\Gamma\,(\Delta, A)$$

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

$$\_,\_{:}[\_] : (\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \to (t : \mathsf{Tm}\,\Gamma\,B) \to B \equiv A\,[\sigma]_T \to \mathsf{Sub}\,\Gamma\,(\Delta, A)$$

instead — turning $(\sigma, \mathsf{transport}(\mathsf{Tm}\,\Gamma, p, t))$ 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 $\mathsf{Tm} : \mathsf{Ctx} \to \mathsf{Set}$, and introduce a function $\mathsf{tyOf} : \mathsf{Tm}\,\Gamma \to \mathsf{Ty}\,\Gamma$ to compute the types of terms.

A QIIR representation of the syntax of type theory

We simultaneously define (changes to QIIT definition highlighted)

$$\mathsf{data}\ \mathsf{Ctx} : \mathsf{Type}$$
$$\mathsf{data}\ \mathsf{Sub} : (\Gamma : \mathsf{Ctx}) \to (\Delta : \mathsf{Ctx}) \to \mathsf{Set}$$
$$\mathsf{data}\ \mathsf{Ty} : (\Gamma : \mathsf{Ctx}) \to \mathsf{Set}$$
$$\mathsf{data}\ \mathsf{Tm} : (\Gamma : \mathsf{Ctx}) \to \mathsf{Set}$$
$$\mathsf{tyOf} : \mathsf{Tm}\ \Gamma \to \mathsf{Ty}\ \Gamma$$

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

$\emptyset : \mathsf{Ctx}$

$\_,\_ : (\Gamma : \mathsf{Ctx})(A : \mathsf{Ty}\,\Gamma) \to \mathsf{Ctx}$

$\_[\_]_T : (A : \mathsf{Ty}\,\Delta)(\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \to \mathsf{Ty}\,\Gamma$

$\_[\_]_t : (t : \mathsf{Tm}\,\Delta)(\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \to \mathsf{Tm}\,\Gamma$

$\emptyset : \mathsf{Sub}\,\Gamma\,\emptyset$

$\_,\_:[\_] : (\sigma : \mathsf{Sub}\,\Gamma\,\Delta)(t : \mathsf{Tm}\,\Gamma) \to$
$\qquad \mathsf{tyOf}\,t \equiv A\,[\sigma]_T \to \mathsf{Sub}\,\Gamma\,(\Delta, A)$

$\mathsf{id} : \mathsf{Sub}\,\Gamma\,\Gamma$

$\_\circ\_ : \mathsf{Sub}\,\Delta\,\Theta \to \mathsf{Sub}\,\Gamma\,\Delta \to \mathsf{Sub}\,\Gamma\,\Theta$

$\pi_1 : \mathsf{Sub}\,\Gamma\,(\Delta, A) \to \mathsf{Sub}\,\Gamma\,\Delta$

$\pi_2 : \mathsf{Sub}\,\Gamma\,(\Delta, A) \to \mathsf{Tm}\,\Gamma$

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

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

$\mathsf{assoc} : (\gamma \circ \tau) \circ \sigma \equiv \gamma \circ (\tau \circ \sigma)$

$[\circ]_T : A\,[\tau]_T\,[\sigma]_T \equiv A\,[\tau \circ \sigma]_T$

$[\mathsf{id}]_T : A \equiv A\,[\mathsf{id}]_T$

$[\mathsf{id}]_t : t \equiv t\,[\mathsf{id}]_t$

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

$,\circ : (q : \mathsf{tyOf}\,(t\,[\tau]_t) \equiv A\,[\sigma \circ \tau]_T)$
$\qquad \to (\sigma, t :[\,pt\,]) \circ \tau \equiv (\sigma \circ \tau, t\,[\tau]_t :[\,qt\,])$

$\mathsf{tyOf}\,(\pi_2\,\{A = A\}\,\sigma) = A\,[\pi_1\,\sigma]_T$

data _ where

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

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

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

$\beta\pi_2 : (q : A\,[\pi_1\,(\sigma, t :[\,p\,])]_T \equiv \mathsf{tyOf}\,t)$
$\qquad \to \pi_1\,(\sigma, t :[\,p\,]) \equiv t$

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

$\mathsf{tyOf}\,(t\,[\sigma]_t) = (\mathsf{tyOf}\,t)\,[\sigma]_T$

$\mathsf{tyOf}\,([\mathsf{id}]_t\,i) = [\mathsf{id}]_T\,i$

$\mathsf{tyOf}\,([\circ]_t\,i) = [\circ]_T\,i$

The substitution calculus as a QIIRT

data _ where

$\emptyset : \mathsf{Ctx}$

$\_,\_ : (\Gamma : \mathsf{Ctx})(A : \mathsf{Ty}\,\Gamma) \to \mathsf{Ctx}$

$\_[\_]_T : (A : \mathsf{Ty}\,\Delta)(\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \to \mathsf{Ty}\,\Gamma$

$\_[\_]_t : (t : \mathsf{Tm}\,\Delta)(\sigma : \mathsf{Sub}\,\Gamma\,\Delta) \to \mathsf{Tm}\,\Gamma$

$\emptyset : \mathsf{Sub}\,\Gamma\,\emptyset$

$\_,\_{:}[\_] : (\sigma : \mathsf{Sub}\,\Gamma\,\Delta)(t : \mathsf{Tm}\,\Gamma) \to$

$\quad \mathsf{tyOf}\,t \equiv A\,[\sigma]_T \to \mathsf{Sub}\,\Gamma\,(\Delta , A)$

What is different?

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

$[\mathsf{id}]_T : A \equiv A\,[\mathsf{id}]_T$

$[\mathsf{id}]_t : t \equiv t\,[\mathsf{id}]_t$

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

$,\circ : (q : \mathsf{tyOf}\,(t\,[\tau]_t) \equiv A\,[\sigma \circ \tau]_T)$

$\quad \to (\sigma , t : [\,pt\,]) \circ \tau \equiv (\sigma \circ \tau, t\,[\tau]_t : [\,qt\,])$

$\mathsf{tyOf}\,(\pi_2\,\{A = A\}\,\sigma) = A\,[\pi_1\,\sigma]_T$

data _ where

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

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

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

$\beta\pi_2 : (q : A\,[\pi_1\,(\sigma , t : [\,p\,])]_T \equiv \mathsf{tyOf}\,t)$

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

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

$\mathsf{tyOf}\,(t\,[\sigma]_t) = (\mathsf{tyOf}\,t)\,[\sigma]_T$

$\mathsf{tyOf}\,([\mathsf{id}]_t\,i) = [\mathsf{id}]_T\,i$

$\mathsf{tyOf}\,([\circ]_t\,i) = [\circ]_T\,i$

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 $(\mathsf{U}, \mathsf{El})$.

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 $(\mathsf{U}, \mathsf{El})$.

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.

$$\mathbb{B}[]_2 : \mathsf{tyOf}\,(\pi_2\,\{\Gamma, \mathbb{B}\}\,\mathsf{id}) \equiv \mathbb{B}\,[\tau]_T$$

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 $(\mathsf{U}, \mathsf{El})$.

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.

$$\mathbb{B}[]_2 : \mathsf{tyOf}\,(\pi_2\,\{\Gamma, \mathbb{B}\}\,\mathsf{id}) \equiv \mathbb{B}\,[\tau]_T$$

Since $\mathsf{Ty}\,\Gamma$ 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

$$\mathsf{recTyOf} : S.\mathit{tyOf}\ t \equiv B \to [\![\mathit{tyOf}]\!]\,(\mathsf{recTm}\ t) \equiv \mathsf{recTy}\ B$$

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

$$\mathsf{recTyOf} : S.\mathit{tyOf}\ t \equiv B \to [\![\mathit{tyOf}]\!]\,(\mathsf{recTm}\ t) \equiv \mathsf{recTy}\ B$$

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

$$
\begin{aligned}
[\![Ctx]\!] &= \mathsf{Type} \\
[\![Ty]\!]\,\Gamma &= \Gamma \to \mathsf{Type} \\
[\![Sub]\!]\,\Gamma\,\Delta &= \Gamma \to \Delta \\
[\![Tm]\!]\,\Gamma &= (\Sigma\, A : \Gamma \to \mathsf{Type})((\gamma : \Gamma) \to A\,\gamma) \\
[\![tyOf]\!]\,(A, t) &= A
\end{aligned}
$$

if we assume UIP so that $[\![Ty]\!]\,\Gamma$ is a set.

Constructing models

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

$$\begin{aligned}
[\![Ctx]\!] &= \mathsf{Type} \\
[\![Ty]\!]\,\Gamma &= \Gamma \to \mathsf{Type} \\
[\![Sub]\!]\,\Gamma\,\Delta &= \Gamma \to \Delta \\
[\![Tm]\!]\,\Gamma &= (\Sigma\, A : \Gamma \to \mathsf{Type})((\gamma : \Gamma) \to A\,\gamma) \\
[\![tyOf]\!]\,(A, t) &= A
\end{aligned}$$

if we assume UIP so that $[\![Ty]\!]\,\Gamma$ is a set. Similarly we can define the term model

$$\begin{aligned}
[\![Ctx]\!] &= \mathsf{Ctx} \\
[\![Ty]\!] &= \mathsf{Ty} \\
&\vdots
\end{aligned}$$

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 $\mathsf{tyOf} : \mathsf{Tm}\,\Gamma \to \mathsf{Ty}\,\Gamma$.

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.

Can we formalise type theory intrinsically without any compromise? Not yet.

Agda 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 $\mathsf{tyOf} : \mathsf{Tm}\,\Gamma \to \mathsf{Ty}\,\Gamma$.

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 co

Thank you!

Can we formalise type theory intrinsically without any compromise? Not yet.

Agda 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.
- 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.
- Dan Licata. 2011. “Running circles around (in) your proof assistant; or, quotients that compute”. `https://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant/`