Can We Formalise Type Theory Intrinsically without Any Compromise? A Case Study in Cubical Agda

Liang-Ting Chen
Academia Sinica
Taipei, Taiwan
liangtingchen@as.edu.tw

Fredrik Nordvall Forsberg
University of Strathclyde
Glasgow, United Kingdom
fredrik.nordvall-forsberg@strath.ac.uk

Tzu-Chun Tsai
Academia Sinica
Taipei, Taiwan
gene0905@icloud.com

Abstract

We present an intrinsic representation of type theory in the proof assistant Cubical Agda, inspired by Awodey's natural models of type theory. The initial natural model is defined as quotient inductive-inductive-recursive types, leading us to a syntax accepted by Cubical Agda without using any transports, postulates, or custom rewrite rules. We formalise some meta-properties such as the standard model, normalisation by evaluation for typed terms, and strictification constructions. Since our formalisation is carried out using Cubical Agda's native support for quotient inductive types, all our constructions compute at a reasonable speed. When we try to develop more sophisticated metatheory, however, the 'transport hell' problem reappears. Ultimately, it remains a considerable struggle to develop the metatheory of type theory using an intrinsic representation that lacks strict equations. The effort required is about the same whether or not the notion of natural model is used.

CCS Concepts: • **Theory of computation → Type theory**.

Keywords: Proof Assistants, Formalisation, Cubical Agda, Quotient Inductive-Inductive-Recursive Type

ACM Reference Format:
Liang-Ting Chen, Fredrik Nordvall Forsberg, and Tzu-Chun Tsai. 2026. Can We Formalise Type Theory Intrinsically without Any Compromise? A Case Study in Cubical Agda. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '26), January 12–13, 2026, Rennes, France.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3779031.3779090

1 Introduction

Internalising the syntax and semantics of type theory in type theory is a long-standing problem which stretches the limits of the theory [8, 16, 21, 24, 44]. There are both practical and theoretical reasons to pursue this problem. On the practical side, an internal representation of type theory is needed for mechanised metatheory and metaprogramming. On the theoretic side, if type theory is supposed to be a general constructive foundation of mathematics, then it should in particular be able to reason about its own syntax and semantics (up to inherent limitations due to Gödel's incompleteness theorems). In dependent type theory, types can depend on terms, which means that all of contexts, types and terms need to be defined simultaneously. This is one reason why formalising type theory in type theory is hard.

Early approaches to formalising type theory, e.g. McKinna and Pollack [45], dealt with untyped terms that were later refined by a typing relation or setoid equality, and thus had to prove a lot of congruence lemmas by hand [16, 21]. A breakthrough was achieved by Altenkirch and Kaposi [8], who showed that quotient inductive-inductive types (QIITs) [7] can be employed to significantly simplify the internal representation of well typed terms (or, more precisely, derivations), since equality constructors can be used to represent equations such as β- and η-equality. They took Dybjer's notion of a model of type theory in the form of a category with families [24], and translated it into a QIIT definition. In effect, this gives rise to the *initial* category with families, with the elimination principles of the QIIT giving a unique morphism of categories with families to any other model. This thus gives a both principled and practical way to formalise the syntax and equational theory of type theory in type theory; the feasibility of the approach was demonstrated by e.g. formalising normalisation by evaluation using this representation [9].

At the time of publication of Altenkirch and Kaposi [8], the proof assistant Agda did not allow equality constructors in data type declarations, so a workaround known as 'Licata's trick' [40] was used by postulating (equality) constructors and writing down the eliminator explicitly, which meant giving up features of the proof assistant such as dependent pattern matching. Cubical Agda, the cubical variant of Agda [55], is now equipped with a native support for QIITs, so it is natural to ask if we can use this support to formalise the intrinsic representation of type theory without the trick or any other compromise.

In this paper, we explore this question and see what the proof assistant provides and lacks to achieve this goal. We quickly find that their QIIT definitions break the strict positivity — a syntactic restriction imposed by Cubical Agda to

This work is licensed under a Creative Commons Attribution 4.0 International License.

CPP '26, Rennes, France
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2341-4/2026/01
https://doi.org/10.1145/3779031.3779090

ensure consistency. Moreover, their definition is cumbersome to work with, since the type of later constructors or even equations often only make sense because of earlier equations. In an intensional type theory, such as those implemented in proof assistants, this manifests itself in transport terms across equality proofs *inside* other terms, and leads to the so-called 'transport hell' — rather than just reasoning about the terms you actually want to study, you now also have to do a lot of reasoning about transports themselves and their algebraic properties. It turns out that we need an alternative way of representing type theory intrinsically without any transport hell, in order to make our formalisation of type theory more lightweight and accepted by Cubical Agda.

The framework of categories with families is only one of several (more or less) equivalent notions of models of type theory [30], and we were wondering if any of the other notions might offer any advantages. Bense [13] suggested that Awodey's notion of natural model [12] might be a good candidate. Indeed, in a natural model, the family of terms indexed by their types $\mathsf{Tm}_\Gamma : \mathsf{Ty}(\Gamma) \to \mathsf{Set}$ (as in a category with families) is replaced by a 'fibred' perspective where each term instead *has* a type, as picked out by a function $\mathsf{tyOf} : \mathsf{Tm}(\Gamma) \to \mathsf{Ty}(\Gamma)$. Terms and types are still indexed by contexts Γ, but since most 'type mismatches' arise from equations between types, not equations between contexts (indeed many formulations of type theory do not even have a notion of context equality), this should mean that many uses of transports can be avoided.

We test this hypothesis by formalising type theory in a form inspired by natural models. Cubical Agda is a particularly good fit for such a project, because not only does it support QIITs, it also supports inductive-recursive types [25], which are needed to simultaneously define the recursive tyOf function together with the inductively defined types $\mathsf{Tm}(\Gamma)$ and $\mathsf{Ty}(\Gamma)$. Indeed, it could be the lack of support for inductive-recursive definitions in many proof assistants which has held back formalisation attempts based on natural models so far.

While we manage to avoid transports occurring in its own syntax, the experiment is not an outright success. Indeed, we found that when developing more sophisticated metatheory, such as when defining a logical predicate model, the use of transports along equations often reappeared. Strictification [23, 36] — a recent technique for making certain equations hold up to definitional equality — partially helps, but does not completely eliminate the problem. Furthermore, we found that the use of natural models is less well supported in the Cubical Agda of today, compared to approaches based purely on QIITs. This is because we are more reliant on the computational behaviour of the recursively defined tyOf function, and this behaviour is only available in 'later' clauses, which leads to the need for hacks and tricks to work around this limitation. We discuss proof assistant features and their helpfulness further towards the end of the paper, after presenting our formalisation.

Contributions. We make the following contributions:

- We present an intrinsically well typed representation of the syntax of type theory, inspired by Awodey's natural models (Section 3).
- We derive elimination and recursion principles for the syntax (Section 3.7), and show how the standard model and the term model are constructed (Section 4.2).
- We discuss strictification constructions on models, and show that they also apply to our notion of natural models (Section 4.4).
- We develop normalisation by evaluation for the substitution calculus (Section 4.3) as a proof of concept: our development is carried out in Cubical Agda, which has a computational implementation of QIITs and principles such as function extensionality, so the resulting normaliser computes, and can potentially be extracted as a verified program.
- We discuss pros and cons of our approach compared to other approaches, and which features of a proof assistant and its metatheory would make future formalisation more feasible (Section 5).

Formalisation. Our formalisation [17] can be found at https://github.com/L-TChen/TTasQIIRT, or archived with DOI 10.5281/zenodo.17802827.

2 Setting and Metatheory

Our formalisation is carried out in Cubical Agda without the use of the Glue type and, in particular, the univalence principle, checked using the option `--cubical=no-glue` available in the forthcoming Agda 2.9.0. It also typechecks with Agda version 2.8.0. We explicitly set-truncate the types we define. Therefore, the term *set* is used interchangeably with type. For the sake of simplicity we occasionally postulate uniqueness of identity proofs (UIP) locally, which is of course inconsistent with univalence but in principle compatible with cubical type theory. We believe that the cubical type theory XTT [51], which enjoys definitional UIP without univalence, justifies this local assumption.

Cubical Agda implements cubical type theory, and one of the important concepts therein is the interval type `I` with two distinguished endpoints `i0` and `i1`. Propositional equality `x ≡ y` in a type `A` is given by *paths* in `A`, i.e., by a functions `I → A`. More generally, dependent paths `p : PathP P a b` are dependent functions `p : (i : I) → P i` sending `i0` to `a : P i0` and `i1` to `b : P i1`. Note that `P : I → Type` itself is a path in the universe `Type`, hence a witness that `P i0 ≡ P i1`, which the dependent path is *over*, so paths are special cases of `PathP` with `P` being a constant path. The constant path `refl`, defined as `λ i → a`, witnesses that equality is reflexive `a ≡ a`, and paths can be lifted to type families

in the sense that there is a transport operation subst : (P : A → Type) → x ≡ y → P x → P y. See the literature on cubical type theory for details [55].

Cubical Agda also allows paths to appear as the target of constructors in inductive definitions. That is, it implements higher inductive types [41]. When defining a function f : H → X by pattern matching out of a higher inductive type, f also needs to be defined on the path constructors: if e : s ≡ t is a path from s to t in H, then f e : f s ≡ f t should be a path from f s to f t in X. Agda's support for simultaneous definitions allows us to define quotient inductive-inductive types (QIITs) [7], where a type A : Type and a type family B : A → Type are defined inductively simultaneously. Agda even allows us to define quotient inductive-inductive-recursive types (QIIRTs), where A : Type and B : A → Type are defined inductively together with a recursive function f : A → C. We will make use of this feature to define types, terms and the tyOf function from terms to types simultaneously.

For the brevity of presentation, we have made arguments implicit for equality constructors, even though they are explicit in our formalisation. Similarly, we are ignoring universe levels, but they are all present in the formalisation.

3 Type Theory as Quotient Inductive Types

In this section, we explain why Altenkirch and Kaposi's representation [8] is hard to use in practice, and rejected by Cubical Agda due to transports in its definition. Then, we show how their representation can be transformed to a representation based on Awodey's natural models. This representation is accepted by Cubical Agda, since it is free from transports.

3.1 Type Theory as the Initial CwF Model

We briefly recall the QIIT representation given by Altenkirch and Kaposi. Each judgement therein is defined as an inductive type, each typing rule as a constructor, and each equality between types, terms, and substitutions as an *equality constructor*. The inhabitants of these types are valid derivations in type theory, because their validity is enforced by typing constraints. The four types of judgements in type theory are represented inductive-inductively as

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

For example, an inhabitant of Tm Γ A represents a derivation for a term of type A in context Γ. Rules are represented by constructors of these inductive types:

```
data _ where
  ∅     : Ctx
  _,_   : (Γ : Ctx) (A : Ty Γ) → Ctx
  _[_]T : (A : Ty Δ) (σ : Sub Γ Δ) → Ty Γ
  _[_]t : (t : Tm Δ A) (σ : Sub Γ Δ) → Tm Γ (A [ σ ]T)
  idS   : Sub Γ Γ
  _∘_   : Sub Δ Θ → Sub Γ Δ → Sub Γ Θ
  _,_   : (σ : Sub Γ Δ) (t : Tm Γ (A [ σ ]T))
    → Sub Γ (Δ , A)
  [∘]T  : A [ τ ]T [ σ ]T ≡ A [ τ ∘ σ ]T
  ...
```

The constructor ∅ represents the empty context, and Γ , A a context extension, while A [σ]T and t [σ]t represent substituted types and terms, respectively. Further, idS is the identity substitution, _∘_ the constructor for substitution composition, and the second _,_ the constructor for extending a substitution σ with a term t of type A [σ]T (making use of Agda's support for overloaded constructor names). The equality constructor [∘]T states that type substitution by τ followed by type substitution by σ is the same as a single substitution by the composition $\tau \circ \sigma$.

When formulating the corresponding rule for the interaction between _∘_ and _,_, we encounter a type mismatch that needs to be resolved by a transport (highlighted in green):

```
,∘ : (σ , t) ∘ τ ≡ (σ ∘ τ , subst (Tm Γ) [∘]T (t [ τ ]t))
```

The reason is that the type of t [τ]t is A [σ]T [τ]T rather than the required A [σ ∘ τ]T. However, since Tm is an argument to subst, the use of transport violates a syntactic restriction of Agda, namely its strict positivity check. In theory, transports are allowed in QIITs [35], but it is not clear to us how this syntactic restriction should be relaxed for higher inductive types supported by Cubical Agda to take into account other cubical primitives (such as hcomp).

In other words, transport hell is not only an obstacle for reasoning, but also breaks strict positivity in Cubical Agda when arising in inductive definitions themselves. The situation becomes worse once additional type formers are introduced — such as Π-types and the type El of elements [8] — since each brings further instances of this problem. Of course, one could bypass the strict positivity check, but doing so would undermine the general trustworthiness of formalisation. Another possibility is to fix the syntactic restriction for HIITs, but it is unclear what the conditions should be. Therefore, we seek an equivalent definition without transports first.

One option is to use *dependent paths* to encode equations over equations. For example, the fact that the identity term substitution really acts as an identity is introduced as an equality constructor [idS]t, defined over the highlighted equality constructor [idS]T : A ≡ A [idS]T for the identity type substitution:

```
[idS]t : PathP (λ i → Tm Γ ( [idS]T i )) t (t [ idS ]t)
```

This is in principle managable — indeed, Altenkirch, Kaposi and Xie [10] successfully use dependent paths and some encoding tricks to represent the groupoid syntax of type

theory in Cubical Agda— but the PathP type is not part of a standard presentation of type theory, but instead rather Cubical Agda specific (of course, dependent paths can also be encoded using transports, but then inheriting the problems described above). Dependent paths also quickly leads to equations over equations over yet more equations in their elimination rules. Hence, it is still preferable to avoid them if possible.

3.2 The 'Ford Transformation' and Index Elimination

To avoid transports in the definition itself, we note that the index A of Tm Γ A often needs an explicit proof for the typing constraint — for example, that the term t in the substitution (σ , t) has type A [σ]T — if this does not happen to hold strictly (i.e., up to judgemental equality), so enforcing this constraint in the index of Tm just shoots ourselves in the foot. Hence, we apply the 'Ford transformation' [43] ("You can have any index you want, as long as it is equal to the specified one") to move the constraint on its index to its argument as an equality proof (highlighted below):

```
_,_:[_] : (σ : Sub Γ Δ) (t : Tm Γ B) (pt : B ≡ A [ σ ]T)
  → Sub Γ (Δ , A)
```

The constructor ,∘ which had a transport in its type becomes

```
,∘ : ... (σ , t :[ pt ]) ∘ τ
     ≡ (σ ∘ τ , t [ τ ]t :[ cong _[ τ ]T pt · [∘]T ])
```

where _·_ is the transitivity of equality. Although transport is not needed this time, the use of cong and _·_ still prevent the definition from being seen as strictly positive. Similar to the Ford transformation, this problem can be overcome by asking for another equality proof, highlighted below, as an argument:

```
,∘ : ... (qt : B [ τ ]T ≡ A [ σ ∘ τ ]T)
  → (σ , t :[ pt ]) ∘ τ ≡ (σ ∘ τ) , t [ τ ]t :[ qt ]
```

As Sub is a set, the additional argument is essentially unique, so this updated constructor does not require any additional information but only defers the proof obligation.

Once the Ford transformation has been applied, the index B in Tm Γ B no longer plays the role of enforcing constraints. This opens the door to a simpler design: instead of carrying the index around, we can 'Ford' all Tm constructors uniformly and remove the index entirely. To preserve the necessary typing information, we simultaneously introduce an auxiliary function tyOf : Tm Γ → Ty Γ that records it explicitly. In the end, the constructor ,∘ becomes

```
,∘ : (σ : Sub Δ Θ) (t : Tm Δ) (τ : Sub Γ Δ)
  → (pt : tyOf t ≡ A [ σ ]T)
  → (qt : tyOf (t [ τ ]t) ≡ A [ σ ∘ τ ]T)
  → (σ , t :[ pt ]) ∘ τ ≡ (σ ∘ τ) , t [ τ ]t :[ qt ]
```

As a side effect, this approach also removes the need for dependent paths in the definition. Two terms can now be compared even when it is not known in advance whether their types are equal. For instance, the equality constructor for the identity substitution becomes

```
[idS]t : t ≡ t [ idS ]t
```

where the fact that t and t [idS]t share the same type follows from their term equality, rather than being a *requirement*.

3.3 Representing the Substitution Calculus using QIIRT

Building on the changes described in Section 3.2, we now spell out our version of the substitution calculus. The following types are defined simultaneously with a recursive function (changes compared to the QIIT version highlighted):

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

Similarly to the QIIT representation, constructors are introduced for rules and equalities as follows, where we highlight constructors that are different from their QIIT counterpart:

```
data _ where
  ∅      : Ctx
  _,_    : (Γ : Ctx) (A : Ty Γ) → Ctx
  _[_]T  : (A : Ty Δ) (σ : Sub Γ Δ) → Ty Γ
  _[_]t  : (t : Tm Δ) (σ : Sub Γ Δ) → Tm Γ
  ∅      : Sub Γ ∅
  _,_:[_] : (σ : Sub Γ Δ) (t : Tm Γ)
         → (pt : tyOf t ≡ A [ σ ]T) → Sub Γ (Δ , A)
  idS    : Sub Γ Γ
  _∘_    : Sub Δ Θ → Sub Γ Δ → Sub Γ Θ
  π1     : Sub Γ (Δ , A) → Sub Γ Δ
  π2     : Sub Γ (Δ , A) → Tm Γ
  idS∘_  : idS ∘ σ ≡ σ
  _∘idS  : σ ∘ idS ≡ σ
  assocS : (γ ∘ τ) ∘ σ ≡ γ ∘ (τ ∘ σ)
  [idS]T : A ≡ A [ idS ]T
  [idS]t : t ≡ t [ idS ]t
  [∘]T   : A [ τ ]T [ σ ]T ≡ A [ τ ∘ σ ]T
  [∘]t   : t [ τ ]t [ σ ]t ≡ t [ τ ∘ σ ]t
  ,∘     : (σ , t :[ pt ]) ∘ τ ≡ (σ ∘ τ , t [ τ ]t :[ qt ])
```

We write wk for the 'weakening' substitution wk = π1 idS : Sub (Γ , A) Γ, and vz for the most recently bound 'zeroth variable' vz = π2 idS : Tm (Γ , A).

The above definition works well, except that we have to interleave the function clauses of tyOf with constructors.

For example, we need define the function clause for $\pi_2\ \sigma$ before the η-law for substitution:

```
tyOf (π₂ {Γ} {Δ} {A} σ) = A [ π₁ σ ]T
data _ where
  ηπ : σ ≡ (π₁ σ , π₂ σ :[ refl ])
```

Otherwise, the proof obligation tyOf $(\pi_2\ \sigma) \equiv A\ [\ \pi_1\ \sigma\ ]_T$ on the right hand side of $\eta\pi$ cannot be fulfilled by refl. We proceed with other equality constructors:

```
data _ where
  ηΘ  : σ ≡ ΘS
  βπ₁ : π₁ (σ , t :[ p ]) ≡ σ
  βπ₂ : ... (q : A [ π₁ (σ , t :[ p ]) ]T ≡ tyOf t)
    → π₂ (σ , t :[ p ]) ≡ t
```

Note that $\beta\pi_2$ has an additional derivable equality proof `q`. This argument is needed as the coherence condition for

```
tyOf (βπ₂ ... q i) = q i
```

since again using any other function while defining inductive types breaks the strict positivity check. The remaining clauses are given as

```
tyOf (t [ σ ]t)   = (tyOf t) [ σ ]T
tyOf ([idS]t i)   = [idS]T i
tyOf ([∘]t i)     = [∘]T i
```

This definition is accepted by Cubical Agda[1] without any warnings or errors. Although Tm is only indexed by Γ : Ctx, the function tyOf ensures that every t : Tm Γ has a type. Hence, Tm only consists of valid derivations and is still an intrinsic representation of type theory.

Replacing the index A : Ty of Tm by a function tyOf : Tm Γ → Ty Γ aligns with Awodey's notion of natural model [12] where the collections of terms and types are represented as presheaves Tm, Ty : $\mathbb{C} \to$ **Set** over the category of contexts $\mathbb{C}$ and connected by a natural transformation Tm → Ty satisfying that each substitution into a non-empty context is equivalent to a pair of substitution and a term. That is, we have derived the intrinsic representation of type theory as a natural model using QIIRT in Cubical Agda. This situates our family of inductive types and their algebras within a well-studied categorical models for type theory.

3.4 Π-types

We extend our object type theory with dependent function types. First we define the lifting of a substitution by a type as the following abbreviation:

```
_↑_ : (σ : Sub Γ Δ) (A : Ty Δ)
    → Sub (Γ , A [ σ ]T) (Δ , A)
_↑_ {Γ} σ A = σ ∘ π₁ {Γ , A [ σ ]T} idS,
              π₂ (idS {Γ , A [ σ ]T}) :[ p ]
```

where p : tyOf $(\pi_2$ idS$) \equiv$ A $[\ \sigma \circ \pi_1$ idS $]_T$. We can use $[\circ]_T$ to define p, as tyOf $(\pi_2$ (idS {Γ , A $[\ \sigma\ ]_T$})) is equal to A $[\ \sigma\ ]_T$ $[\ \pi_1$ idS $]_T$ by definition.[2]

Other constructors are introduced following the 'Ford transformation', with differences compared to the usual QIIT presentation highlighted:

```
data _ where
  Π      : (A : Ty Γ) (B : Ty (Γ , A)) → Ty Γ
  app    : (t : Tm Γ) (B : Ty (Γ , A))
         → (pt : tyOf t ≡ Π A B) → Tm (Γ , A)
  abs    : (t : Tm (Γ , A)) → Tm Γ
  Π[]    : (Π A B) [ σ ]T ≡ Π (A [ σ ]T) (B [ σ ↑ A ]T)
  abs[]  : abs t [ σ ]t ≡ abs (t [ σ ↑ A ]t)
  Πβ     : app (abs t) (tyOf t) pt ≡ t
  Πη     : abs (app t B pt)        ≡ t
tyOf (app t B p)       = B
tyOf (abs {A = A} t)   = Π A (tyOf t)
tyOf (abs[] σ t i)     = Π[] σ (tyOf t) i
tyOf (Πβ t pt i)       = tyOf t
tyOf (Πη t pt i)       = pt (~ i)
```

Apart from the extra clauses of tyOf, the main change happens in the constructor app. The constraint that t is of type Π A B is enforced there, but every other constructor remains almost the same as their QIIT definition. We have formulated application and abstraction as an isomorphism between terms of type B in context Γ , A and terms of type Π A B in context Γ, but we can also derive ordinary application:

```
_:[_]$$_:[_] : (t : Tm Γ) → tyOf t ≡ Π A B
             → (s : Tm Γ) → tyOf s ≡ A
             → Tm Γ
t :[ p ]$$ s :[ q ] = app t _ p [ idS , s :[ q · [idS]T ] ]t
```

As an example, we can write the identity function at type A as id A = abs {A = A} vz, and the identity function at type Π A (A $[\ \mathsf{wk}\ ]_T$) applied to the identity function at type A is represented by

```
idid : (A : Ty Γ) → Tm Γ
idid A = id (Π A (A [ wk ])) :[ refl ]$$ id A :[ refl ]
```

and indeed using the equality constructors, we have a proof of tyOf (idid A) ≡ Π A (A $[\ \mathsf{wk}\ ]_T$) as expected.

[1] At the time of writing, Cubical Agda does not support interleaved mutual definitions, but it can be equivalently defined using forward declarations. We will discuss this idiom in Section 3.8.

[2] Yet, as interleaving function clauses with inductive types is also not supported, the strict equality tyOf $(\pi_2\ \sigma)$ = A $[\ \pi_1\ \sigma\ ]_T$ is not available at this point for p. We again use forward declarations to introduce the required equalities tyOf $(\pi_2\ \sigma) \equiv$ A $[\ \pi_1\ \sigma\ ]_T$ and tyOf $(\pi_2$ idS$) \equiv$ A $[\ \sigma \circ \pi_1$ idS $]_T$ to be defined later as refl and $[\circ]_T$, see Section 3.8 for further details.

3.5 The Type of Booleans

To introduce the inductive type of Booleans, we need to specialise the substitution lifting. Let us see its constructors (with differences highlighted) and explain why a specialisation is needed.

```
data _ where
  𝔹      : Ty Γ
  𝔹[]    : 𝔹 [ σ ]T ≡ 𝔹
  tt ff  : Tm Γ
  tt[]   : tt [ σ ]t ≡ tt
  ff[]   : ff [ σ ]t ≡ ff
tyOf tt   = 𝔹
tyOf ff   = 𝔹
tyOf (tt[] σ i) = 𝔹[] σ i
tyOf (ff[] σ i) = 𝔹[] σ i
data _ where
  elim𝔹 : (P : Ty (Γ , 𝔹))
    (t : Tm Γ) (pt : tyOf t ≡ P [ idS , tt :[ [idS]T ] ]T)
    (u : Tm Γ) (pu : tyOf u ≡ P [ idS , ff :[ [idS]T ] ]T)
    (b : Tm Γ) (pb : tyOf b ≡ 𝔹 [ idS ]) → Tm Γ
tyOf (elim𝔹 P u t pu pt b pb) = P [ idS , b :[ pb ] ]T
```

The only thing missing from the above definition is the substitution rule for elim𝔹: applying the substitution σ to 'elim𝔹 P t pt u pu b pb' is equal to applying a lifted substitution $\sigma \uparrow \mathbb{B}$ to P and σ to t, u, and b. However, P $[\,\sigma \uparrow \mathbb{B}\,]_T$ gives us a type in the context Δ , $\mathbb{B}$ $[\,\sigma\,]_T$ instead of Δ , $\mathbb{B}$, so we provide a lifting with a type Sub Γ Δ → Sub (Γ , 𝔹) (Δ , 𝔹) with a proof that tyOf (π_2 {Γ , 𝔹} idS) ≡ 𝔹 $[\,\tau\,]_T$. The proof, however, requires the transitivity of equalities, and Cubical Agda would see this as a strict positivity problem. Instead we introduce a *superfluous* equality constructor $\mathbb{B}[]_2$ to satisfy its proof obligation (highlighted):

```
data _ where
  𝔹[]₂ : tyOf (π₂ {Γ , 𝔹} idS) ≡ 𝔹 [ τ ]T
_↑𝔹 : (σ : Sub Γ Δ) → Sub (Γ , 𝔹) (Δ , 𝔹)
_↑𝔹 {Γ} {Δ} σ = σ ∘ π₁ {Γ , 𝔹} idS , π₂ idS :[ 𝔹[]₂ ]
```

Finally, we introduce the equality constructor for the interaction between elim𝔹 and substitution:

```
data _ where
  elim𝔹[] : ...
    (pt₂ : tyOf (t [ σ ]t) ≡ P [ σ ↑𝔹 ]T [ idS , tt :[ [idS]T ] ]T)
    (pu₂ : tyOf (u [ σ ]t) ≡ P [ σ ↑𝔹 ]T [ idS , ff :[ [idS]T ] ]T)
    (pb₂ : tyOf (b [ σ ]t) ≡ 𝔹 [ idS ]T)
    (q : P [ idS , b :[ pb ] ]T [ σ ]T
           ≡ P [ σ ∘ wk , vz :[ 𝔹[]₂ ] ]T
                [ idS , b [ σ ]t :[ pb₂ ] ]T)
    → (elim𝔹 P t pt u pu b pb) [ σ ]t
           ≡ elim𝔹 (P [ σ ↑𝔹 ]T) (t [ σ ]t) pt₂ (u [ σ ]t) pu₂
                   (b [ σ ]t) pb₂
tyOf (elim𝔹[] P u t pu pt b pb pt₂ pu₂ pb₂ q i) = q i
```

Note again that we also defer the coherence proof of tyOf for elim𝔹[] by introducing another argument q in elim𝔹 which can be removed when defining its elimination rule.

3.6 A Tarski Universe

Using the same idiom described previously, a Tarski universe of types is introduced to our type theory in the same vein. First we need U : Ty Γ as the type of codes, and a type family El of elements for a given code (differences compared to the usual presentation highlighted):

```
data _ where
  U    : Ty Γ
  U[]  : U [ σ ]T ≡ U
  El   : (u : Tm Γ) (pu : tyOf u ≡ U) → Ty Γ
  El[] : (q : tyOf (u [ τ ]t) ≡ U)
    → (El u pu) [ τ ]T ≡ El (u [ τ ]t) q
```

For the type 𝔹 of Booleans, its code 𝕓 is introduced with a type equality El𝕓 such that the elements of 𝕓 are exactly 𝔹:

```
data _ where
  𝕓    : Tm Γ
  𝕓[]  : 𝕓 [ σ ]t ≡ 𝕓
tyOf 𝕓        = U
tyOf (𝕓[] σ i) = U[] {σ = σ} i
data _ where
  El𝕓 : El {Γ} 𝕓 refl ≡ 𝔹
```

For the Π-type, we again need a specialised substitution lifting. This continues the pattern of introducing superfluous constructors to satisfy proof obligations (differences again highlighted).

```
data _ where
  El[]₂ : (u : Tm Δ) (pu : tyOf u ≡ U)
    → (pu₂ : tyOf (u [ σ ]t) ≡ U)
    → tyOf (π₂ {Γ , El (u [ σ ]t) pu₂ } idS)
     ≡ El u pu [ σ ∘ π₁ idS ]T
_↑El : (σ : Sub Γ Δ) {u : Tm Δ}
  {pu : tyOf u ≡ U} {pu' : tyOf (u [ σ ]t) ≡ U}
  → Sub (Γ , El (u [ σ ]t) pu') (Δ , El u pu)
(σ ↑El) {u} {pu} {pu'} =
  σ ∘ π₁ idS , π₂ idS :[ El[]₂ u pu pu' ]
```

Finally, we introduce the code π for Π and the type equality Elπ to complete our definition of type theory using QIIRT:

```
data _ where
  π   : (a : Tm Γ) (pa : tyOf a ≡ U)
      → (b : Tm (Γ , El a pa))
      → (pb : tyOf b ≡ U) → Tm Γ
```

```
  π[] :  (a : Tm Γ) (pa : tyOf a ≡ U)
      → (b : Tm (Γ , El a pa)) (pb : tyOf b ≡ U)
      → (pa' : tyOf (a [ σ ]t) ≡ U)
      → (pb' : tyOf (b [ σ ↑El ]t)
      → (π a pa b pb) [ σ ]t
      ≡ π (a [ σ ]t) pa' (b [ σ ↑El ]t) pb'
tyOf (π _ _ _ _) = U
data _ where
  Elπ : (a : Tm Γ) (pa : tyOf a ≡ U)
    → (b : Tm (Γ , El a pa)) (pb : tyOf b ≡ U)
    → El (π a pa b pb) refl ≡ Π (El a pa) (El b pb)
tyOf (π[] _ _ _ _ _ _ i) = U[] i
```

In the end, we emphasise that the introduction of superfluous equality proofs and constructors only makes sense because of set truncation. These additional arguments are essentially unique and thus do not add any new laws to type theory, but merely serve as devices to meet the syntactic restriction of strict positivity in the current implementation of Cubical Agda. After the definition, one can introduce 'smart' constructors that supply canonical constructions for the superfluous equality proofs, if one so wishes.

3.7 Recursion and Elimination Principles

We turn to recursion and elimination principles for our syntax. Our QIIRT definition yields an *initial model*. This means that for any other model (algebra) of our theory, there is a unique structure-preserving map from our syntax to that model. The recursion and elimination principles make this property concrete. Here, we only discuss the part for the substitution calculus, since other type formers are addressed similarly. For the interested reader, see our formalisation.

The signature for an algebra is packed in a record type SC (short for Substitution Calculus). Inductive types and the function tyOf are interpreted as indexed sets and a function between sets, respectively, with explicit set truncation assumptions. Constructors of our syntax correspond to function fields in this record, including equality constructors and clauses of tyOf.

```
record SC : Set where
  field
    Ctx    : Set
    Ty     : Ctx → Set
    Tm     : Ctx → Set
    Sub    : Ctx → Ctx → Set
    tyOf   : Tm Γ → Ty Γ
    ∅      : Ctx
    _,C_   : (Γ : Ctx) (A : Ty Γ) → Ctx
    _[_]T  : (A : Ty Δ) (σ : Sub Γ Δ) → Ty Γ
    _[_]t  : (t : Tm Δ) (σ : Sub Γ Δ) → Tm Γ
    idS∘_  : idS ∘ σ ≡ σ
    ...
    βπ₂    : π₂ (σ , t :[ p ]) ≡ t
    ...
    tyOf[] : tyOf (t [ σ ]t)          ≡ (tyOf t) [ σ ]T
    tyOfπ₂ : tyOf (π₂ {A = A} σ) ≡ A [ π₁ σ ]T
```

To distinguish syntactic constructors from the semantic methods in SC, we qualify the syntactic constructors with 'S.' in the following discussion.

One could think that it would be advantageous to leave out superfluous equality constructors as fields in the record, as their semantic counterparts can be defined within any given model using the other methods, and thus reduce the burden for users defining models. For example, the semantic interpretation of S.tyOfπ₂idS can be given as

```
tyOfπ₂idS : (σ : Sub Γ Δ) (A : Ty Δ)
  → tyOf (π₂ {A = A [ σ ]T} idS) ≡ A [ σ ∘ π₁ idS ]T
tyOfπ₂idS σ A = tyOfπ₂ idS · [∘]T _ _ _
```

However, some models might allow stricter and thus simpler implementations of such 'derived' constructors, which in turn might simplify later definitions[3], and so we *do* include them in the record. For derived equality constructors such as tyOfπ₂idS above, their implementation is essentially unique due to set truncation assumption, but for derived point constructors, we also ask for a proof that the constructor is equal to its the canonical interpretation.

The recursion principle consists of a family of functions that map syntax to their semantic counterparts:

```
recCtx : S.Ctx       → Ctx
recTy  : S.Ty Γ      → Ty (recCtx Γ)
recTm  : S.Tm Γ      → Tm (recCtx Γ)
recSub : S.Sub Γ Δ → Sub (recCtx Γ) (recCtx Δ)
```

We also need a function that translates proofs about syntactic equalities into semantic equalities:

```
recTyOf : S.tyOf t ≡ B → tyOf (recTm t) ≡ recTy B
```

The definition of these functions proceeds by pattern matching on the syntactic structure. Each clause is an application of the corresponding method from the SC record:

```
recCtx S.∅          = ∅
recCtx (Γ S., A)    = recCtx Γ ,C recTy A
recSub (σ S., t :[ pt ])
  = recSub σ , recTm t :[ recTyOf t pt ]
...
```

The most interesting case is perhaps recTyOf, which handles the translation of syntactic equations. For a given syntactic

[3] For example, we were struggling to finish the construction of the standard Set model including Π-types and a universe, described in Section 4.2, before we changed to a more specialised interpretation of some derived constructors. This made our remaining proof obligations definitionally true.

equality proof p : S.tyOf t ≡ B, we must construct a semantic equality proof. This is done by applying recTy to both sides of the syntactic equality to get recTy (S.tyOf t) ≡ recTy B, and then using the semantic counterpart of the tyOf clause to derive tyOf (recTm t) ≡ recTy (S.tyOf t). Taking S.π_2 as an example, we have:

```
recTyOf {B = B} (S.π₂ {A = A} σ) p =
  tyOf (recTm (S.π₂ σ))          ≡⟨⟩
  tyOf (π₂ (recSub σ))           ≡⟨ tyOfπ₂ (recSub σ) ⟩
  (recTy A) [ π₁ (recSub σ) ]T   ≡⟨⟩
  recTy (A S.[ S.π₁ σ ]T)        ≡⟨⟩
  recTy (S.tyOf (S.π₂ σ))        ≡⟨ cong recTy p ⟩
  recTy B                        ■
```

The coherence conditions for recTyOf over equality constructors are trivial because of set-truncation.

Unfortunately, the termination checker of Cubical Agda is especially brittle and does not see the above as obviously terminating, perhaps (among other things) because of the seemingly unrelated recursive call recTy B in the type of recTyOf. Since all recursive calls in clauses defining the recursive functions are on structurally smaller terms, we are confident that the definition is indeed terminating, and mark it as such using the `{-# TERMINATING #-}` pragma.

For the elimination principle, we consider the notion of displayed algebras over an SC-algebra M, as a record type SC• parametric in M, and later instantiate M to the term algebra, i.e. the syntax. Carriers of a displayed algebra as well as the semantics of tyOf are given below.

```
record SC• (M : SC) : Set where
  open SC M
  field
    Ctx•  : Ctx → Set
    Ty•   : Ctx• Γ → Ty Γ → Set
    Tm•   : Ctx• Γ → Tm Γ → Set
    Sub•  : Ctx• Γ → Ctx• Δ → Sub Γ Δ → Set
    tyOf• : Tm• Γ• t → Ty• Γ• (tyOf t)
```

As motives are indexed by their underlying model, we will have equations over equations of the underlying model. Inspired by the syntax of equational reasoning for displayed categories in the 1Lab [39], we introduce some convenient syntax for specialised dependent paths, e.g.,

```
_≡Tm[_]_ : Tm• Γ• t → t ≡ u → Tm• Γ• u → Set
_≡Tm[_]_ {Γ• = Γ•} t• e u• =
  PathP (λ i → Tm• Γ• (e i)) t• u•
```

The signature for SC•-algebras is similar to the one for SC-algebras, except that each displayed operation is indexed by their underlying operation (leading to equations over equations).

```
field
  ∅•      : Ctx• ∅
  _,•_    : Ctx• Γ → Ty• Γ• A → Ctx• (Γ ,C A)
  _[_]T•  : Ty• Δ• A → Sub• Γ• Δ• σ → Ty• Γ• (A [ σ ]T)
  _[_]t•  : Tm• Δ• t → Sub• Γ• Δ• σ → Tm• Γ• (t [ σ ]t)
  tyOf[]• : tyOf• (t• [ σ• ]t•)
              ≡Ty[ tyOf[] ] (tyOf• t• [ σ• ]T•)
  ...
  [idS]t• : t•                ≡Tm[ [idS]t ] t• [ idS• ]t•
  [∘]t•   : t• [ τ• ]t• [ σ• ]t• ≡Tm[ [∘]t ]  t• [ τ• ∘• σ• ]t•
```

Note that if $[\mathsf{idS}]_t$ in its QIIT definition is formulated with a dependent path, the equality proof in the middle of _≡Tm[_]_ has to be a dependent path. As a result, we would have to specify two underlying equations as

```
t• ≡Tm[ [idS]T ][ [idS]t ] t• [ idS• ]t•
```

and equational reasoning with them would involve three equations altogether. It is nice that we do not have to deal with this extra proof obligation in our formulation.

The elimination principle is stated similarly to the recursion principle but indexed over the term algebra (Section 4.1):

```
elimCtx   : (Γ : S.Ctx)   → Ctx• Γ
elimTy    : (A : S.Ty Γ)  → Ty• (elimCtx Γ) A
elimTm    : (t : S.Tm Γ) → Tm• (elimCtx Γ) t
elimSub   : (σ : S.Sub Γ Δ)
  → Sub• (elimCtx Γ) (elimCtx Δ) σ
elimTyOf : (t : S.Tm Γ) (p : S.tyOf t ≡ A)
  → tyOf• (elimTm t) ≡Ty[ p ] elimTy A
```

Just like for the recursors, we again mark the eliminators as terminating using a pragma.

For the coherence conditions in the definition of the eliminators, we may need additional reasoning steps instead of just using the semantics equation, so we use transitivity of dependent paths:

```
_·P_ : {x' : B x} {y' : B y} {z' : B z}
  → {p : x ≡ y} {q : y ≡ z}
  → PathP (λ i → B (p i)) x' y'
  → PathP (λ i → B (q i)) y' z'
  → PathP (λ i → B ((p · q) i)) x' z'
```

We also use set truncation to identify the highlighted p · q with the desired underlying equation in special-purpose equational reasoning combinators such as the following:

```
beginSub[_]_ : ({p} q : σ ≡ τ)
  → σ• ≡Sub[ p ] τ• → σ• ≡Sub[ q ] τ•
beginSub[_]_ {σ•} {τ•} {p} q p• =
  subst (λ r → σ• ≡Sub[ r ] τ•) (Sub-is-set p q) p•
```

For example, the coherence proof for $\eta\pi$ is given by

```
beginSub[ ηπ ]
  elimSub σ
```

```
    ≡Sub[ ηπ ]⟨ ηπ· (elimSub σ) ⟩
  π₁• (elimSub σ) ,
  π₂• (elimSub σ) :[ refl , tyOfπ₂• (elimSub σ) ]t•
    ≡Sub[ refl ]⟨ cong (λ z → ... , ... :[ refl , z ]t•)
                      (Ty··-is-set _ _) ⟩
  π₁• (elimSub σ) ,
  elimTm (π₂ σ) :[ refl , elimTyOf (π₂ σ) refl ]t•
    ■
```

with the steps $\sigma^\bullet$ ≡Sub[p]⟨ $\mathsf{p}^\bullet$ ⟩ $\tau^\bullet$ implemented using _·ᴘ_. These steps give us an equation over $\eta\pi \cdot (\mathsf{refl} \cdot \mathsf{refl})$, and beginSub[$\eta\pi$]_ gives us an easy way to correct this to an equation over $\eta\pi$ as desired instead.

3.8 Practical Workarounds for Mutual Definitions

So far, we have outlined how the recursion and elimination principles should be defined *ideally*. In practice, however, limitations (and occasional mysterious bugs) of the proof assistant require us to adopt certain workarounds in order to implement the intended definitions.

Mutually Interleaved QIITs. Constructors of QIITs currently can not be interleaved in Cubical Agda [3], even within an **interleaved mutual** block. The reason is that such a block is desugared into a collection of forward declarations for the **data** types, rather than for the constructors. In principle, all constructors belonging to the same family of QIITs should be declared within the same context [35]. However, due to this desugaring, equality constructors may end up depending on other constructors that are not yet in scope.

We work around this issue as follows: (i) we make forward declarations for the *entire definition* of the QIITs, including the constructors; (ii) we introduce each constructor but only refer to forward declarations when referring to other constructors; (iii) we define the forward declarations by their corresponding constructors; (iv) finally, we expose only the actual constructors, omitting the forward declarations. The following snippet illustrates this approach:

```
module S where
  data Ctx : Set
  ...
  ∅   : Ctx   -- note indentation, not a constructor!
  _,_ : (Γ : Ctx) (A : Ty Γ) → Ctx
  ...
  data Ctx where
    ∅'  : Ctx   -- this is a constructor
    _,'_ : (Γ : Ctx) (A : Ty Γ) → Ctx
  ...
  ∅     = ∅'  -- make definition for forward declarations
  _,_   = _,'_
  ...
open S public    -- expose actual constructors
  hiding (∅; _,_; ...)
  renaming (∅' to ∅; _,'_ to _,_; ...)
```

This translation from QIITs in theory to their actual definitions in Cubical Agda should be sufficient to define mutually interleaved QIITs. Indeed, pedagogical presentations of type theory typically introduces one type former at a time, together with its formation, introduction, elimination, and equality rules (see, e.g., Hofmann [30]), rather than presenting all type formers at once using a few large monolithic sets of rules.

Mutual Interleaved QIIRTs. Interleaving function clauses with inductive types is a different matter, since we cannot forward declare a function clause together with its computational behaviour.[4] However, because we have 'Forded' the typing constraints into equality proofs, what we actually need at the point of introducing constructors is only the existence of such an equality proof, not its computational content.

Our workaround is therefore as follows: (i) we declare the existence of the required equality proof before it is used, (ii) we define tyOf only after all datatype declarations have been given, and (iii) we provide the actual definition corresponding to the forward declaration. For instance, the equality constructor $\eta\pi$ asks for a proof of $\mathsf{tyOf}\ (\pi_2\ \sigma) \equiv \mathsf{A}\ [\pi_1\ \sigma]_\mathrm{T}$. In this case, we simply forward declare such a proof:

```
tyOfπ₂ : tyOf (π₂ σ) ≡ A [π₁ σ]T
ηπ     : σ ≡ (π₁ σ , π₂ σ :[ tyOfπ₂ ])
```

Then, once tyOf has been defined, we simply set $\mathsf{tyOf}\pi_2$ to refl:

```
tyOf (π₂' {Γ} {Δ} {A} {σ}) = A [π₁ {A = A} σ]T
...
tyOfπ₂ = refl
```

This translation is valid as long as the computational behaviour of the interleaved function clauses is not needed up to judgemental equality.

Mutually-Defined Functions. Since the constructors of QII(R)Ts can be mutually interleaved, their recursion and elimination principles also need to be given in the same vein. However, Agda does not allow us to interleave clauses of different functions directly. One workaround is to use forward declarations as a lifting of the entire clause and then perform the necessary coercions along the corresponding equality proofs by hand.

Another possibility is to define a single family of functions indexed by tags. For instance, the recursion principle can be implemented by introducing a datatype Tag with one constructor for each motive:

```
data Tag : Set where
  ctx ty sub tm tyof : Tag
```

[4] Custom rewrite rules are not allowed in Cubical Agda.

Then, we define the recursion principle uniformly as rec, with tyOfRec computing its type. Each actual function is introduced as a synonym for rec at the appropriate tag:

```
tyOfRec : Tag → Set
rec : (t : Tag) → tyOfRec t
tyOfRec ctx = S.Ctx → Ctx
rec ctx S.∅       = ∅
rec ctx (Γ S., A) = rec ctx Γ ,C rec ty A
...
recCtx = rec ctx
...
```

At the time of writing, however, this encoding cannot be fully carried out in Cubical Agda: some terms that should be strictly equal are not recognised as such during type checking. For example, in the following clause of rec:

```
rec sub (S._,_:[_] {Γ} {Δ} {A} σ t p) = _,_:[_]
  {rec ctx Γ} {rec ctx Δ} {rec ty A} (rec sub σ) (rec tm t)
  {! rec tyof t p !}
```

the subterm in the hole is accepted by Agda, but refining it results an error, as the terms rec ty (A S.[σ]T) and rec ty A [rec sub σ]T are not recognised as equal — even though the former was already defined to be the latter.

In our formalisation, we fall back on forward declarations along with coercions. We are still investigating the root cause of this behaviour, but it may point to a design flaw.

4 Metatheory

Having defined type theory as QIIRTs, we now turn to models of type theory as well as constructions of new models from existing ones.

4.1 Term Model

The term model is a self-interpretation of syntax, allowing displayed models to be instantiated over it and ensuring that the elimination rule computes. The definition is routine: each field is interpreted by the corresponding constructor, except that additional equality constraints (such as the one in $\beta\pi_2$ highlighted below) are replaced by actual proofs:

```
Term : SC
Term = record
  {∅       = S.∅
  ;tyOf[]  = refl
  ...
  ;βπ₂     = λ {Γ} {Δ} {A} σ t p
    → S.βπ₂ σ t p (cong (A [_]T) (S.βπ₁ σ t p) · sym p) }
```

Other type formers are given similarly. The recursion and elimination principles justify that the term model is indeed the initial model.

4.2 Standard Model

In the standard model, contexts are interpreted as sets in Cubical Agda, types as sets indexed by a context Γ, substitutions as functions between these sets, and terms as *pairs* (A, t) consisting of an interpreted type A : Γ → Set together with a dependent function t : (γ : Γ) → A γ. The interpretation of tyOf is simply the first component A. In other words, terms are interpreted as a type A, together with a 'section' t : (γ : Γ) → A γ of that type, as usual:

```
std : SC
std .Ctx       = Set
std .Ty Γ      = Γ → Set
std .Sub Γ Δ   = Γ → Δ
std .Tm Γ      = Σ[ A ∈ (Γ → Set) ] ((γ : Γ) → A γ)
std .tyOf (A , t) = A
```

The main construction of the model is the same as in the standard model of type theory using QIITs [9, Section 4], except that the typing constraint p in σ , t :[p] is 'Forded'. As a result, the value t γ below must be transported along p, as highlighted below:

```
std .∅           = Unit
std ._,C_ Γ A    = Σ Γ A
std ._[_]T A σ γ = A (σ γ)
std ._[_]t (A , t) σ = (λ γ → A (σ γ)) , (λ γ → t (σ γ))
std .tyOf[]      = refl
...
std ._,_:[_] σ (A , t) p γ
  = (σ γ , transport (λ i → p i γ) (t γ) )
```

To extend the standard model for the universe U, we define a Tarski universe of codes and its interpretation

```
data UU : Set
T : UU → Set
data UU where
  bool : UU
  pi : (a : UU) → (T a → UU) → UU
T : UU → Set
T bool     = Bool
T (pi a b) = (x : T a) → T (b x)
```

Each of the constructs in Section 3.6 can now be interpreted:

```
std .U _          = UU
std .U[]          = refl
std .El (A , u) pu γ = T (subst (λ A → A γ) pu (u γ))
std .𝔹            = (λ _ → UU) , λ _ → bool
std .π (A , a) pa (B , b) pb = (λ _ → UU) , λ γ → pi
  (transport (λ i → pa i γ) (a γ))
  (λ a → transport (λ i → pb i (γ , a)) (b (γ , a)))
...
```

Coherence conditions are then verified using standard properties of transport. We have formalised the standard model for type theory with Π-types, Booleans, and a Tarski universe.

The main effort in the formalisation arises from the lack of *regularity* [51]: there is a path transportRefl between the transport along reflexivity and the identity but they are not strictly equal. For instance, the coherence condition for Π[] is given as

```
stdPi .Π[] {Γ} {Δ} {A} σ B i γ =
  (a : A (σ γ)) → B (σ γ , transportRefl³ a) (~ i))
```

where transportRefl³ amounts to using transportRefl three times. If regularity were available, this would collapse to the trivial reflexivity proof.

4.3 Normalisation by Evaluation, and the Logical Predicate Interpretation

We implement normalisation by evaluation (NbE) for the substitution calculus. Following the approach for type theory [9], we define inductive-recursively both the type of variables (with their embedding into terms) and the type of renamings (with their embedding into substitutions). The implementation is straightforward, so we omit the details here. In the end, this yields a normalisation function that produces, for every term, a de Bruijn variable and computes:

```
normalise : (t : Tm Γ) → Σ[ tⁿ ∈ NeTm Γ ] t ≡ ⌜ tⁿ ⌝
```

Compared to NbE for the substitution calculus using QIITs, our formalisation is simpler: no transports appear at all, because variables and terms are not indexed by their types.

The picture is very different for the logical predicate interpretation. Although NbE works cleanly, the logical predicate interpretation — often considered a benchmark challenge [1] for language formalisation — remains at least as difficult as in the QIIT-based setting, even for the substitution calculus.

To see why, recall that the motives for Ctx and Ty in the logical predicate interpretation are given by

```
record Ctx^P (Γ : Ctx) : Set where
  field
    ctx^P : Ctx
    wk^P  : Sub ctx^P Γ

Ty^P : Ctx^P Γ → Ty Γ → Set
Ty^P Γ^P A = Ty (ctx^P Γ^P , A [ wk^P Γ^P ]T)
```

Here the typing constraint (ctx^P Γ^P , A [wk^P Γ^P]$_T$) already shares the familiar shape of Tm Γ A, but with an additional complication: the index explicitly demands a type substitution. Since the QIIRT representation only provides equality constructors for type substitutions, the development quickly results in repeated and tedious use of transports.

In short, NbE benefits directly from removing typing indices and avoids transports altogether, whereas the logical predicate interpretation still inherits the need for coercions with type substitutions. We did not bother to finish all cases of the logical predicate interpretation, as Altenkirch and Kaposi [8] have already shown that such tedious use of transports is possible (but impractical) in theory.

4.4 Strictification

Instead, we turn our attention to *strictification* [23, 36]: given a model of type theory, certain equality constructors can be made strict (i.e., made to hold judgementally) to form a new model. A familiar analogy is the transition from lists to difference lists, where a list is represented by a list-appending function and list concatenation is function composition, which is associative judgementally.

In the same spirit, we may attempt to 'strictify' the category part of the substitution calculus using the Yoneda embedding, so that the unit laws and associativity law hold strictly, *modulo* the property of naturality. Given any SC-algebra, we define a presheaf interpretation of Sub by

```
record Sub^y (Γ Δ : Ctx) : Set where
  constructor _,_
  field
    y   : ∀ {Θ} → Sub Θ Γ → Sub Θ Δ
    nat : (τ : Sub Ξ Θ) (δ : Sub Θ Γ)
        → {δ' : Sub Ξ Γ} → δ ∘ τ ≡ δ' → y δ ∘ τ ≡ y δ'
```

We are thankful to one of our anonymous reviewers for pointing out that introducing δ' and a proof of $\delta \circ \tau \equiv \delta$' rather than working with $\delta \circ \tau$ directly gives a little bit more slack, which gives us strict unit and associativity laws:

```
Yoneda .idS           = (λ γ → γ) , (λ τ δ r → r)
Yoneda ._∘_ (σ , pσ) (σ' , pσ')
  = (λ γ → σ (σ' γ)) , (λ τ δ r → pσ τ (σ' δ) (pσ' τ δ r))
Yoneda .idS∘_ _       = refl
Yoneda ._∘idS _       = refl
Yoneda .assocS _ _ _  = refl
```

To further strictify the laws for type substitutions for the substitution calculus, we can adapt the local universe construction [23, 42] for M : SC. Types under the context Γ in the local universe construction are interpreted as a triple consisting of a context V as the local universe, a type under the context V, and a substitution from Γ to V:

```
record Ty³ (Γ : Ctx) : Set where
  constructor ty³
  field
    V : Ctx
    E : Ty V
    ⌜_⌝ : Sub Γ V
```

Type substitution is defined by 'delaying' the substitution, i.e., by viewing the substitution ⌜ A ⌝ as an accumulator parameter. Then, we can show that the functor laws for type

substitution boil down to the right unit and the associativity laws for substitutions:

```
(M !) .[idS]T {Γ} {A}  =
  cong (ty³ (A .V) (E A)) (sym (⌜ A ⌝ ∘idS))
(M !) .[∘]T A σ τ      =
  cong (ty³ (A .V) (E A)) (assocS σ τ ⌜ A ⌝)
```

If M is strictified by the Yoneda embedding, then the laws for identity substitution and substitution composition in M ! will be strict. Hence, combining both techniques, we can construct models with strict category laws and substitution functor laws. Nevertheless, strictification does *not* resolve our difficulties with the logical predicate interpretation. For example, the above strictification will not make the substitution rule for Π-types strict, and thus transports are still needed.

5 Discussion and Conclusion

It is well known that type theory in type theory is possible in theory, but in practice its formalisation often requires giving up some of the support and safety checks provided by proof assistants. From one point of view, our work addresses the following question: is there any existing type-theoretic proof assistant that can formalise the intrinsic representation of type theory using QIITs reliably, without compromise? Based on our experimental formalisation in Cubical Agda, our answer is regrettably: not yet.

Comparison with Intrinsic Approaches. Previous intrinsic formalisations [8, 9, 32, 35] based on the CwF semantics of type theory have resorted to using postulated constructors and custom rewrite rules to manually define QIITs and their eliminators. However, this comes at a cost: the proof assistant no longer performs strict positivity, coverage, or termination checks for functions defined from quotient inductive types, nor does it supports dependent pattern matching, program extraction, and interactive theorem proving. The loss of coverage check for inductive types is mitigated by using hand-crafted eliminators (see Section 3.7), since the coverage check is also in effect for record types.

Ehrhard [18, 26] presented a variation of the substitution calculus, which, as Altenkirch, Kaposi and Xie [10] point out, is well suited for formalisation in Cubical Agda, since all use of transports can be replaced by the use of dependent paths. Our approach leads to no transports appearing in the syntax and also avoids the use of dependent paths at all. However, the same transports (and the same equations for them) seem to have a tendency to come back in concrete model constructions, as explained in Section 4.

We remark that the lack of transports is an advantage for avoiding strict positivity issues in the current implementation of Cubical Agda. By using native features such as higher inductive types, rather than postulates in ordinary Agda, we do get computational interpretations. For example, our implementation of normalization by evaluation in Section 4.3 can actually normalise terms and could be potentially extracted as Haskell programs with the cubical information explicitly erased using Agda's `--erased-cubical` feature.

For other notions of models of type theory, such as locally Cartesian closed categories [50], (split) comprehension categories [31] or contextual categories [15], it is not yet clear if initial models can be defined as QIITs or QIIRTs. Brunerie and de Boer [14, 22] work with contextual categories in order to prove the Initiality Conjecture [52] for Martin-Löf Type Theory, but uses an extrinsic representation of terms.

Strictification. Kaposi and Pujet [36] have shown how strictification techniques can simplify proofs, but this is an orthogonal issue. Even though strictification turns most of the equality constructors about substitution to strict equalities, this does not help with transports appearing in the QIIT definition of terms and the resulting strict positivity issues, as strictification can only be applied *after* the inductive types are defined. In Section 4.4, we have sketched how also our notion of models can be strictified using a simpler idea. However, a proper strictification may require a different metatheory than 'just' the support of QII(R)Ts (see below).

A Proper Syntax for HII(R)Ts. The syntax for declaring higher inductive-inductive types and QIITs in Cubical Agda falls short of their theoretical capabilities [34, 35]. As shown in Section 3.1, the legitimate definition of type theory based on CwF with transports violates the syntactic restriction of strict positivity in Cubical Agda. Even though transport hell is better avoided in practice, this is not a justification for excluding otherwise valid definitions. The alternative definition, based on natural models, does not violate strict positivity but still requires forward declarations (Section 3.8) to overcome the inconvenience of syntax.

A Theory of QIIRTs. We work around the problem by defining type theory using QIIRTs, but this raises another question: what is the theory of QIIRTs, anyway?

Different forms of QIITs and inductive-recursive types (IRTs) have been used to develop type theory in type theory. Large and infinitary IRTs are used for the standard model of universes, while small inductive-recursive types [29] have been used for implementing normalisation by evaluation [9]. Meanwhile QIITs and QIIRTs can be used for intrinsic representations of type theory. However, note that in our representation, the target of tyOf is the inductive type Ty being defined simultaneously, which is not a form of definition supported by current theories of large or small induction-recursion. Nevertheless, we expect that the encoding of small inductive-recursive types as indexed inductive types still applies to this 'self-referring' induction-recursion, as we have derived our representation by 'undoing' the encoding.

A framework to encompass all variants will be convenient for proof assistant implementation. Such a scheme might possibly be developed by extending the type theory of QIITs [35, 38] with an additional construct to distinguish strict and weak equalities and a coverage condition for function clauses.

The Support of Interleaved Mutual Definitions. Another challenge concerns interleaved mutual definitions. Since constructors of QIITs may be mutually interleaved, the elaboration from dependent pattern matching to eliminators need to take this into account. Our workaround, using forward declarations to lift function clauses to fix the dependency, sacrifices their computational behaviour. Another option is to use 'Xie's Trick' [33] and reduce the mutual definition to a two-sorted QIIT with a base sort of tags. Furthermore, our definitions appear to reach the limits of the termination checker: even seemingly harmless functions when defining the recursion principle fail to be seen as terminating.

Strict Propositions, and Observational Type Theory. The recent work on strictified syntax [36] addresses transport hell using observational type theory (OTT) [47–49], with strict propositions in the metatheory playing a central role. While Agda does support strict propositions [28], this feature was not designed to work with Cubical Agda [4]. To formalise the metatheory of type theory using QIITs with as few transports as possible, it seems inevitable to use a different metatheory rather than the off-the-shelf metatheory offered by Cubical Agda. In particular, regularity and definitional UIP — supported by OTT (see Altenkirch, Boulier, Kaposi and Tabareau [6] and Pujet and Tabareau [49] for discussion of regularity) and by its cubical variant XTT [51] — would simplify our standard model in Section 4.2 considerably.

The use of QIITs in OTT [36] in Agda requires the user themselves to implement the coercion rules for inductive types [49] as well as their elimination principles. Quotient inductive types are not supported in the implementation of OTT in Rocq [46] and its theory is still being developed [27].

The option `--cubical=no-glue` in the forthcoming Agda 2.9.0 [5] disables the Glue type in cubical mode and, in principle, yields a cubical type theory compatible with UIP [2]. Since Cubical Agda already provides support for HIITs (with the earlier caveats), realising a type theory with QIITs, strict propositions, and computational UIP, as a variant of Cubical Agda may now be within reach [53, 54].

The Ford Transformation and Definitional UIP. The Ford transformation is known to work well with definitional UIP [11]. So far, we have only 'Forded' the `Tm` constructors, but what if every constructor were Forded, with indices removed entirely? The resulting representation of type theory would remain intrinsic, but all typing constraints would appear as equality proofs, which are definitionally proof-irrelevant in XTT or OTT. This could provide a way to escape transport hell without relying on strictification.

Of course, explicitly transforming these typing constraints to equality proofs would still be tedious and error-prone, so an elaboration from QIITs to their Forded presentation using QIIRTs would be useful. The connection between QIITs and its Forded definition with the index eliminated echoes the notion of reornament [19, 20, 37], so this translation itself may be of interest in general.

Conclusion. There are still gaps between a pen-and-paper type theory and a fully formalised type theory in a proof assistant, on both the theoretical and practical sides. Without further advances in the technology of proof assistants, formalising type theory intrinsically within a proof assistant remains a difficult task. We hope that the lessons learned here can help the design of future proof assistants, so that one day we may implement a proof assistant within a proof assistant without (too much) sweat and tears.

Acknowledgments

We appreciate the comments from the anonymous reviewers, which improved the results of the paper and the formalisation in several ways. We are also grateful to Shu-Hung You for his comments on an early draft. The work is supported by the National Science and Technology Council of Taiwan under grant NSTC 114-2222-E-001-001-MY3, and the Engineering and Physical Sciences Research Council (EP/Y000455/2).

Data-Availability Statement

All data underpinning this publication are openly available from Zenodo [17] with DOI 10.5281/zenodo.17802827.

References

[1] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, and Steven Schäfer. 2019. POPLMark reloaded: Mechanizing proofs by logical relations. *Journal of Functional Programming* 29 (2019), e19. doi:10.1017/S0956796819000170

[2] Agda issue. 2019. A variant of Cubical Agda that is consistent with UIP. https://github.com/agda/agda/issues/3750

[3] Agda issue. 2021. Interleaved mutual and equality constructors. https://github.com/agda/agda/issues/5362

[4] Agda issue. 2022. Fix #5816: skip generating cubical clauses for Prop stuff. https://github.com/agda/agda/pull/5897

[5] Agda issue. 2025. Implement a `-cubical=no-glue` option. https://github.com/agda/agda/pull/7861

[6] Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, and Nicolas Tabareau. 2019. Setoid type theory—A syntactic translation. In *Mathematics of Program Construction (MPC) (Lecture Notes in Computer Science, Vol. 11825)*, Graham Hutton (Ed.). Springer, Cham, 155–196. doi:10.1007/978-3-030-33636-3_7

[7] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. 2018. Quotient inductive-inductive types. In *Foundations of Software Science and Computation Structures (FoSSaCS) (Lecture Notes in Computer Science, Vol. 10803)*, Christel Baier and Ugo Dal Lago (Eds.). Springer, Cham, 293–310. doi:10.1007/978-3-319-89366-2_16

[8] Thorsten Altenkirch and Ambrus Kaposi. 2016. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

Languages (POPL '16). Association for Computing Machinery, New York, NY, USA, 18–29. doi:10.1145/2837614.2837638

[9] Thorsten Altenkirch and Ambrus Kaposi. 2017. Normalisation by evaluation for type theory, in type theory. *Logical Methods in Computer Science* 13, 4 (Oct. 2017). doi:10.23638/LMCS-13(4:1)2017

[10] Thorsten Altenkirch, Ambrus Kaposi, and Szumi Xie. 2026. The groupoid-syntax of type theory is a set. In *Computer Science Logic (CSL'26)*. https://arxiv.org/abs/2509.14988 To appear.

[11] Thorsten Altenkirch and Conor McBride. 2006. Towards observational type theory. (2006). http://www.strictlypositive.org/ott.pdf

[12] Steve Awodey. 2018. Natural models of homotopy type theory. *Mathematical Structures in Computer Science* 28, 2 (2018), 241–286. doi:10.1017/S0960129516000268

[13] Viktor Bense, Ambrus Kaposi, and Szumi Xie. 2024. Strict syntax of type theory via alpha-normalisation. (2024). https://vipwww.itu.dk/research/types2024/slides/ViktorBense-Strictsyntaxoftypetheoryviaalpha-normalisation.pdf Presented at TYPES'24.

[14] Guillaume Brunerie, Menno de Boer, Peter LeFanu Lumsdaine, and Anders Mörtberg. 2019. A formalization of the initiality conjecture in Agda. (2019). https://guillaumebrunerie.github.io/pdf/initiality.pdf Slides of a talk at the Homotopy Type Theory 2019 Conference.

[15] John Cartmell. 1986. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic* 32 (1986), 209–243. doi:10.1016/0168-0072(86)90053-9

[16] James Chapman. 2009. Type theory should eat itself. *Electronic Notes in Theoretical Computer Science* 228 (2009), 21–36. doi:10.1016/j.entcs.2008.12.114

[17] Liang-Ting Chen, Fredrik Nordvall Forsberg, and Tzu-Chun Tsai. 2025. *Formalisation underlying "Can We Formalise Type Theory Intrinsically without Any Compromise? A Case Study in Cubical Agda"*. doi:10.5281/zenodo.17802827

[18] Thierry Coquand. 2020. Generalised algebraic presentation of type theory. (2020). https://www.cse.chalmers.se/~coquand/cwf2.pdf

[19] Pierre-Évariste Dagand. 2017. The essence of ornaments. *Journal of Functional Programming* 27, e9 (2017). doi:10.1017/S0956796816000356

[20] Pierre-Évariste Dagand and Conor McBride. 2014. Transporting functions across ornaments. *Journal of Functional Programming* 24, 2–3 (2014), 316–383. doi:10.1017/S0956796814000069

[21] Nils Anders Danielsson. 2007. A formalisation of a dependently typed language as an inductive-recursive family. In *Types for Proofs and Programs (TYPES 2006) (Lecture Notes in Computer Science, Vol. 4502)*, Thorsten Altenkirch and Conor McBride (Eds.). Springer, Berlin, Heidelberg. doi:10.1007/978-3-540-74464-1_7

[22] Menno de Boer. 2020. *A Proof and Formalization of the Initiality Conjecture of Dependent Type Theory*. Licentiate thesis. Department of Mathematics, Stockholm University. https://urn.kb.se/resolve?urn=urn%3Anbn%3Ase%3Asu%3Adiva-181640

[23] István Donkó and Ambrus Kaposi. 2022. Internal strict propositions using point-free equations. In *27th International Conference on Types for Proofs and Programs (TYPES 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 239)*, Henning Basold, Jesper Cockx, and Silvia Ghilezan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 6:1–6:21. doi:10.4230/LIPIcs.TYPES.2021.6

[24] Peter Dybjer. 1996. Internal type theory. In *Types for Proofs and Programs. TYPES 1995*, Stefano Berardi and Mario Coppo (Eds.). Lecture Notes in Computer Science, Vol. 1158. Springer, Berlin, Heidelberg, 120–134. doi:10.1007/3-540-61780-9_66

[25] Peter Dybjer and Anton Setzer. 1999. A finite axiomatization of inductive-recursive definitions. In *Typed Lambda Calculi and Applications (Lecture Notes in Computer Science, Vol. 1581)*, Jean-Yves Girard (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 129–146. doi:10.1007/3-540-48959-2_11

[26] Thomas Ehrhard. 1988. *Une sémantique catégorique des types dépendents*. Ph. D. Dissertation. Université Paris VII.

[27] Thiago Felicissimo and Nicolas Tabareau. 2025. Towards quotient inductive types in observational type theory. (2025). https://msp.cis.strath.ac.uk/types2025/abstracts/TYPES2025_paper85.pdf Presented at TYPES'25.

[28] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional proof-irrelevance without K. *Proceedings of the ACM on Programming Languages* 3, POPL (2019). doi:10.1145/3290316

[29] Peter Hancock, Conor McBride, Neil Ghani, Lorenzo Malatesta, and Thorsten Altenkirch. 2013. Small induction recursion. In *Typed Lambda Calculi and Applications (Lecture Notes in Computer Science, Vol. 7941)*, Masahito Hasegawa (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 156–172. doi:10.1007/978-3-642-38946-7_13

[30] Martin Hofmann. 1997. *Extensional Constructs in Intensional Type Theory*. Springer London. doi:10.1007/978-1-4471-0963-1

[31] Bart Jacobs. 1993. Comprehension categories and the semantics of type dependency. *Theoretical Computer Science* 107, 2 (1993), 169–207. doi:10.1016/0304-3975(93)90169-T

[32] Ambrus Kaposi. 2017. *Type theory in a type theory with quotient inductive types*. Ph. D. Dissertation. University of Nottingham. https://eprints.nottingham.ac.uk/41385/

[33] Ambrus Kaposi. 2019. Re: Separate definition of constructors? Post to the Agda mailing list, https://web.archive.org/web/20241004151846/https://lists.chalmers.se/pipermail/agda/2019/011176.html.

[34] Ambrus Kaposi and András Kovács. 2020. Signatures and induction principles for higher inductive-inductive types. *Logical Methods in Computer Science* 16, 1 (2020), 10:1–10:30. doi:10.23638/LMCS-16(1:10)2020

[35] Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. 2019. Constructing quotient inductive-inductive types. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 2:1–2:24. doi:10.1145/3290315

[36] Ambrus Kaposi and Loïc Pujet. 2025. Type theory in type theory using a strictified syntax. *Proceedings of the ACM on Programming Languages* 9, ICFP (2025). doi:10.1145/3747535

[37] Hsiang-Shang Ko and Jeremy Gibbons. 2016. Programming with ornaments. *Journal of Functional Programming* 27 (2016), e2. doi:10.1017/S0956796816000307

[38] András Kovács and Ambrus Kaposi. 2020. Large and infinitary quotient inductive-inductive types. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '20)*. Association for Computing Machinery, New York, NY, USA, 648–661. doi:10.1145/3373718.3394770

[39] Amélia Liao, Daniel Sölch, Naïm Favier, and Reed Mullanix. 2022. Displayed categories. https://1lab.dev/Cat.Displayed.Base.html.

[40] 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/

[41] Peter LeFanu Lumsdaine and Michael Shulman. 2020. Semantics of higher inductive types. *Mathematical Proceedings of the Cambridge Philosophical Society* 169, 1 (2020), 159–208. doi:10.1017/S030500411900015X

[42] Peter LeFanu Lumsdaine and Michael A. Warren. 2015. The local universes model: An overlooked coherence construction for dependent type theories. *ACM Transactions on Computational Logic* 16, 3 (2015). doi:10.1145/2754931

[43] Conor McBride. 1999. *Dependently Typed Functional Programs and their Proofs*. Ph. D. Dissertation. University of Edinburgh. https://era.ed.ac.uk/bitstream/id/600/ECS-LFCS-00-419.pdf

[44] Conor McBride. 2010. Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming (WGP' 10, 12)*, Bruno C. d. S. Oliveira and Marcin Zalewski (Eds.). Baltimore, Maryland, USA, 1–12. doi:10.1145/1863495.1863497

[45] James McKinna and Robert Pollack. 1999. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning* 23 (1999), 373–409. doi:10.1023/A:1006294005493

[46] Loïc Pujet. 2024. Observational Coq. GitHub repository. https://github.com/loic-p/observational-coq

[47] Loïc Pujet and Nicolas Tabareau. 2022. Observational equality: now for good. *Proceedings of the ACM on Programming Languages* 6, POPL (2022). doi:10.1145/3498693

[48] Loïc Pujet and Nicolas Tabareau. 2023. Impredicative observational equality. *Proceedings of the ACM on Programming Languages* 7, POPL (2023). doi:10.1145/3571739

[49] Loïc Pujet and Nicolas Tabareau. 2024. Observational equality meets CIC. In *Programming Languages and Systems (ESOP) (Lecture Notes in Computer Science, Vol. 14576)*, Stephanie Weirich (Ed.). Springer, Cham, 275–301. doi:10.1007/978-3-031-57262-3_12

[50] R. A. G. Seely. 1984. Locally cartesian closed categories and type theory. *Mathematical Proceedings of the Cambridge Philosophical Society* 95, 1 (1984), 33–48. doi:10.1017/S0305004100061284

[51] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. 2022. A cubical language for Bishop sets. *Logical Methods in Computer Science* 18, 1 (2022). doi:10.46298/lmcs-18(1:43)2022

[52] Thomas Streicher. 1991. *Semantics of type theory: correctness, completeness, and independence results.* Birkhauser Boston Inc., USA. doi:10.1007/978-1-4612-0433-6

[53] Yee-Jian Tan. 2025. *Towards Computational UIP in Cubical Agda.* Master's thesis. Ecole polytechnique; Inria - Paris.

[54] Yee-Jian Tan, Andreas Nuyts, and Dominique Devriese. 2025. Towards Computational UIP in Cubical Agda. https://arxiv.org/abs/2511.21209

[55] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2021. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *Journal of Functional Programming* 31, e8 (2021). doi:10.1017/S0956796821000034

Received 2025-09-13; accepted 2025-11-13