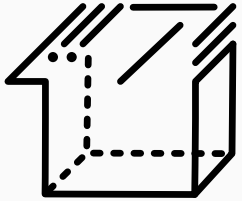
Towards Computational UIP in Cubical Agda

or: making proof assistants nicer to use by **filling squares**!



Yee-Jian Tan (j.w.w. Andreas Nuyts, Dominique Devriese)

DRADS 2026
16 March 2026

Why Proof Assistants Matter?

Proof assistants (e.g. Agda  , Rocq  , Lean ) verify mathematical proofs.

Slogan: “Proposition as Types, Proofs as Programs” 📕

Why Proof Assistants Matter?

Proof assistants (e.g. Agda  , Rocq  , Lean ) verify mathematical proofs.

Slogan: “Proposition as Types, Proofs as Programs” 📕

One possible setup: program $P : I \to O$ under ***ANY*** input $i : I$ is $\mathrm{isNice} : O \to \mathrm{Prop}$:

$$\forall i : I, \mathrm{isNice}\ (P\ i)\ \mathrm{holds}.$$

Where isNice could be termination, memory safety, doesn’t access webcam...

Why Proof Assistants Matter?

Proof assistants (e.g. Agda, Rocq, Lean) verify mathematical proofs.

Slogan: “Proposition as Types, Proofs as Programs” 📕

One possible setup: program $P : I \to O$ under ***ANY*** input $i : I$ is $\mathrm{isNice} : O \to \mathrm{Prop}$:

$$\forall i : I, \mathrm{isNice}\ (P\ i)\ \mathrm{holds}.$$

Where isNice could be termination, memory safety, doesn’t access webcam...

🛡️ **CompCert** [Ler+16] verified C compiler: compiled code provably equivalent to source

Why Proof Assistants Matter?

Proof assistants (e.g. Agda  , Rocq  , Lean ) verify mathematical proofs.

Slogan: “Proposition as Types, Proofs as Programs” 📕

One possible setup: program $P : I \to O$ under ***ANY*** input $i : I$ is $\mathrm{isNice} : O \to \mathrm{Prop}$:

$$\forall i : I, \mathrm{isNice}\ (P\ i)\ \mathrm{holds}.$$

Where isNice could be termination, memory safety, doesn’t access webcam...

🛡️ **CompCert** [Ler+16] — verified C compiler: compiled code provably equivalent to source

🛡️ **VeLLVM** [BCZ25] — Rocq formalization of LLVM IR semantics. Verified optimization passes (mem2reg, memory safety).

Why Proof Assistants Matter?

Proof assistants (e.g. Agda  , Rocq  , Lean ) verify mathematical proofs.

Slogan: “Proposition as Types, Proofs as Programs” 📕

One possible setup: program $P : I \to O$ under ***ANY*** input $i : I$ is $\mathrm{isNice} : O \to \mathrm{Prop}$:

$$\forall i : I, \mathrm{isNice}\ (P\ i)\ \mathrm{holds}.$$

Where isNice could be termination, memory safety, doesn’t access webcam…

- 🛡️ **CompCert** [Ler+16] — verified C compiler: compiled code provably equivalent to source
- 🛡️ **VeLLVM** [BCZ25] — Rocq formalization of LLVM IR semantics. Verified optimization passes (mem2reg, memory safety).
- 🛡️ **RustCompCert** [Wu+26] — (ongoing, 2026) — end-to-end verified Rust compiler, including a verified borrow checker.

Equality as paths: a richer foundation

Homotopy Type Theory (HoTT): equalities as **paths** between points in a space [Uni13].

Equality as paths: a richer foundation

Homotopy Type Theory (HoTT): equalities as **paths** between points in a space [Uni13].

- **Functional extensionality** (funext): pointwise equal functions are equal:

$$(\forall x, f(x) = g(x)) \Rightarrow f = g$$

Equality as paths: a richer foundation

Homotopy Type Theory (HoTT): equalities as **paths** between points in a space [Uni13].

- **Functional extensionality** (funext): pointwise equal functions are equal:

$$(\forall x, f(x) = g(x)) \Rightarrow f = g$$

- **Higher Inductive Types** (HITs): types defined by their equalities

  e.g. integers as $\mathbb{Z} = \mathbb{N} \times \mathbb{N}/\sim$ where $-1 := (0,1) \sim (2,3)$

Equality as paths: a richer foundation

Homotopy Type Theory (HoTT): equalities as **paths** between points in a space [Uni13].

- **Functional extensionality** (funext): pointwise equal functions are equal:

$$(\forall x, f(x) = g(x)) \Rightarrow f = g$$

- **Higher Inductive Types** (HITs): types defined by their equalities

  e.g. integers as $\mathbb{Z} = \mathbb{N} \times \mathbb{N}/\sim$ where $-1 := (0,1) \sim (2,3)$

Assumes the **univalence axiom**: equivalent (isomorphic) types are equal. (If they behave exactly the same way, they must be equal). Buy 1 free 2!

Implemented in Agda as **Cubical Agda**.  Univalence derived from **Glue** types.

~~*Homotopy Type Theory*~~

Let's do **paths** and **squares** and **cubes**!  [1]

[1] [Coh+17]

Equality as paths: geometric interpretation

Type : a space

Terms : points in that space

Equality proof : a path between points

An example in natural numbers $\mathbb{N}$:

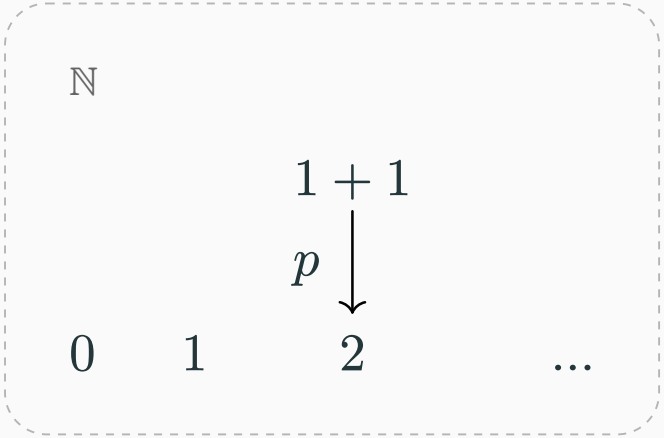
$\mathbb{N}$ is a space: $0, 1, 2, \ldots$ are its points.

Equality as paths: geometric interpretation

Type : a space

Terms : points in that space

Equality proof : a path between points



An example in natural numbers $\mathbb{N}$:

A proof that $1+1=2$ is a path $1+1 \xrightarrow{p} 2$ in $\mathbb{N}$.[1]

Reversing p gives you $2=1+1$ (symmetry).

[1] $\mathbb{N}$ is actually a **set** (h-set): all paths are refl, and $1+1$, 2, $2+0$ are the same point. Shown as distinct here for illustration only.

Equality as paths: geometric interpretation

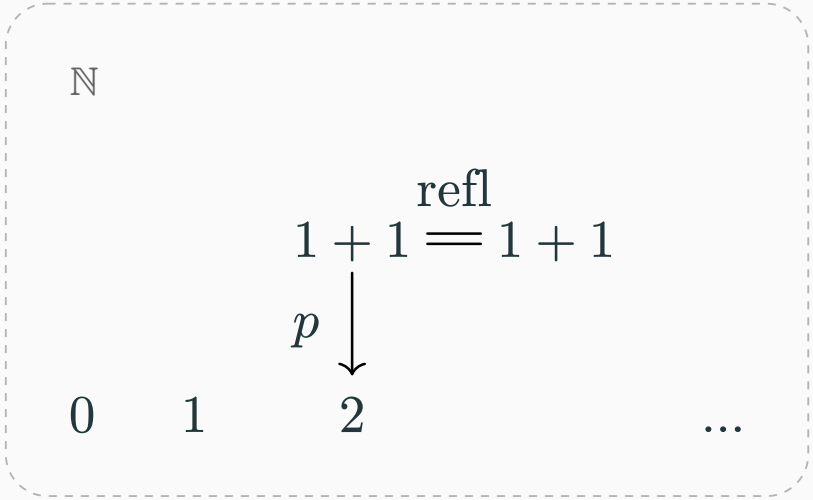
Type : a space

Terms : points in that space

Equality proof : a path between points

An example in natural numbers $\mathbb{N}$:

What about $1 + 1 = 1 + 1$? That is the constant path, $\mathbf{refl}$ (reflexivity).

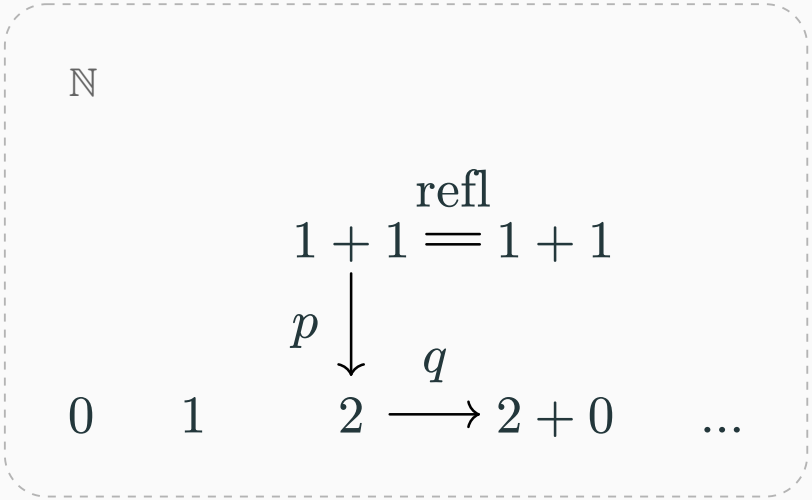


Equality as paths: geometric interpretation

Type : a space

Terms : points in that space

Equality proof : a path between points



An example in natural numbers $\mathbb{N}$:

In Cubical Agda, you can compose 3[1] paths:

- $p : 1 + 1 = 2$ and
- $q : 2 = 2 + 0$ and
- $\mathrm{refl} : 1 + 1 = 1 + 1$

give

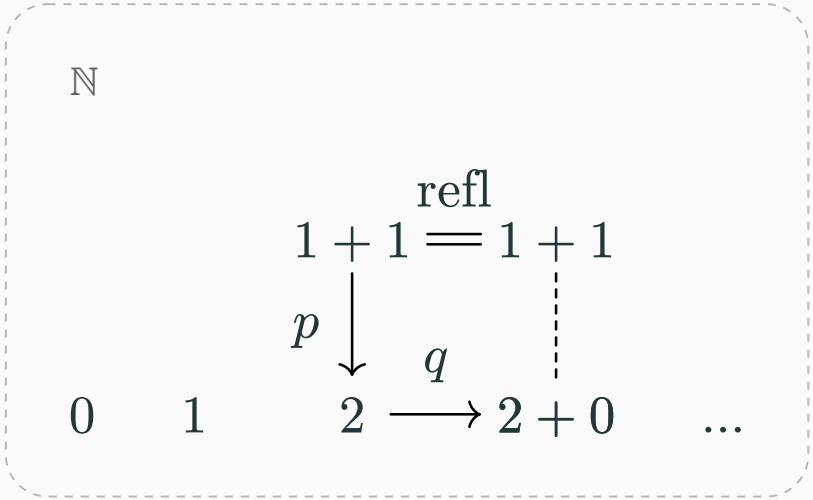
[1] Guess why it's called **Cubical**!

Equality as paths: geometric interpretation

Type : a space

Terms : points in that space

Equality proof : a path between points



An example in natural numbers $\mathbb{N}$:

In Cubical Agda, you can compose 3[1] paths:

- $p : 1 + 1 = 2$ and
- $q : 2 = 2 + 0$ and
- $\mathrm{refl} : 1 + 1 = 1 + 1$

give

$$q \cdot p \cdot \mathrm{refl} : 1 + 1 = 2 + 0.$$

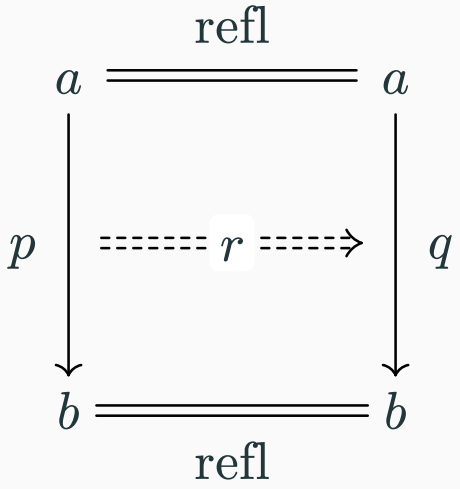
[1] Guess why it's called **Cubical**!

Equality as paths: geometric interpretation

Type : a space

Terms : points in that space

Equality proof : a path between points



Observation

For an arbritraty type A, two paths $p, q : a =_A b$ can themselves be **equal**: witnessed by a **square**.

Squares can be equal too: then it becomes... a **cube**!

And **cubes** can be equal too... (you get the idea)

UIP and Square-Filling

In general, two paths $p, q : a = b$ need not be equal, but many structures (e.g. $\mathbb{N}$, $\mathbb{Z}$) are like sets: paths are always unique. (Easier!)

Uniqueness of Identity Proofs (UIP) condition

"There is at most one proof of $x =_A y$ for any $x\ y : A$."

Incompatible with one key ingredient of HoTT: **univalence**!

Fortunately compatible with Cubical Agda without *Glue* types: the source of **univalence** in Cubical Agda. We implemented a `--cubical=no-glue` variant of Agda [Tan25].

Geometrically:

UIP and Square-Filling

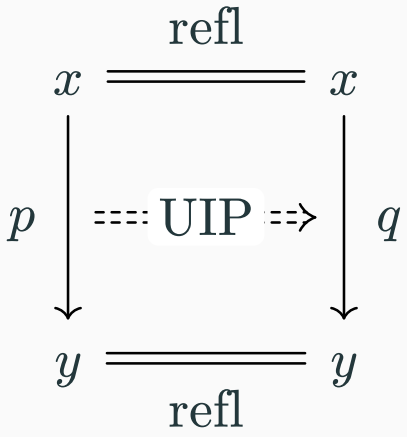
In general, two paths $p, q : a = b$ need not be equal, but many structures (e.g. $\mathbb{N}$, $\mathbb{Z}$) are like sets: paths are always unique. (Easier!)

Uniqueness of Identity Proofs (UIP) condition

"There is at most one proof of $x =_A y$ for any $x\ y : A$."

Geometrically:

UIP: any square with two **reflexive** sides has a filling.



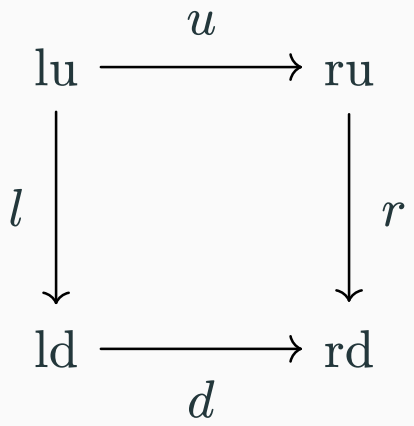
UIP and Square-Filling

In general, two paths $p, q : a = b$ need not be equal, but many structures (e.g. $\mathbb{N}$, $\mathbb{Z}$) are like sets: paths are always unique. (Easier!)

Geometrically:

UIP: any square with two **reflexive** sides has a filling.

Square-Filling (SqFill): relax the refl conditions: **any** square has a filling.



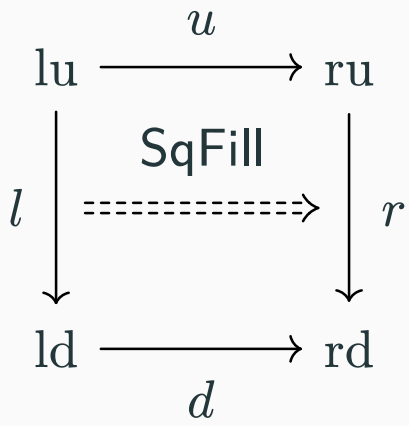
UIP and Square-Filling

In general, two paths $p, q : a = b$ need not be equal, but many structures (e.g. $\mathbb{N}$, $\mathbb{Z}$) are like sets: paths are always unique. (Easier!)

Geometrically:

UIP: any square with two **reflexive** sides has a filling.

Square-Filling (SqFill): relax the refl conditions: **any** square has a filling.



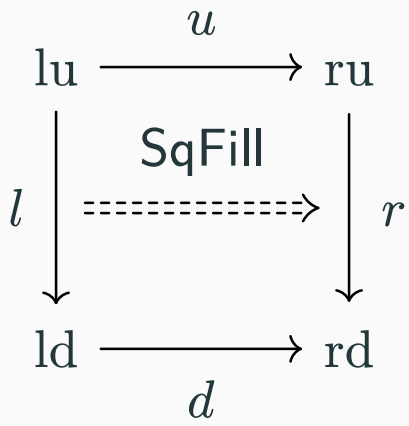
UIP and Square-Filling

In general, two paths $p, q : a = b$ need not be equal, but many structures (e.g. $\mathbb{N}$, $\mathbb{Z}$) are like sets: paths are always unique. (Easier!)

Geometrically:

UIP: any square with two **reflexive** sides has a filling.

Square-Filling (SqFill): relax the refl conditions: **any** square has a filling.



Surprisingly: UIP $\leftrightarrow$ SqFill are **equivalent**, but SqFill is easier to work with!

Genius idea 💡: just add in **SqFill** everywhere!

⚠️ Bad news: SqFill doesn't compute (naively)

Naïvely adding **SqFill** as a **postulate** blocks computation. If **SqFill** appears in a term:

... **SqFill** $(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool})$ ...

$\to_\beta$ ... **SqFill** $(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool}))$ ...

$\to_\beta$ ... **SqFill** $(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool}))$ ...

$\vdots$

Everything else around it evaluates, but it doesn't.

Like x in $42 + x$, can never return a value.

⚠️ Bad news: SqFill doesn't compute (naively)

Naïvely adding **SqFill** as a **postulate** blocks computation. If **SqFill** appears in a term:

... **SqFill** $(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool})$ ...

$\to_\beta$ ... **SqFill** $(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool}))$ ...

$\to_\beta$ ... **SqFill** $(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool}))$ ...

⋮

Everything else around it evaluates, but it doesn't.

Like x in $42 + x$, can never return a value.

Our fix: make SqFill **compute** by inspecting the type.

$$\textbf{SqFill}\ \underbrace{(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool}))}_{A \to B}$$

⚠️ Bad news: SqFill doesn't compute (naively)

Naïvely adding **SqFill** as a **postulate** blocks computation. If **SqFill** appears in a term:

... **SqFill** $(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool})$ ...

$\to_\beta$ ... **SqFill** $(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool}))$ ...

$\to_\beta$ ... **SqFill** $(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool}))$ ...

⋮

Everything else around it evaluates, but it doesn't.

Like x in $42 + x$, can never return a value.

Our fix: make SqFill **compute** by inspecting the type.

$$\textbf{SqFill}\ \underbrace{(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool}))}_{A \to B}$$

$$\to_\beta \mathrm{SqFill}_\Pi \left(\textbf{SqFill}\ \underbrace{(\mathbb{N} \times \mathrm{Bool})}_{A \times B} \right)$$

⚠️ Bad news: SqFill doesn't compute (naively)

Naïvely adding **SqFill** as a **postulate** blocks computation. If **SqFill** appears in a term:

... **SqFill** $(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool})$ ...

$\to_\beta$ ... **SqFill** $(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool}))$ ...

$\to_\beta$ ... **SqFill** $(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool}))$ ...

⋮

Everything else around it evaluates, but it doesn't.

Like x in $42 + x$, can never return a value.

Our fix: make SqFill **compute** by inspecting the type.

$$\textbf{SqFill}\ \underbrace{(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool}))}_{A \to B}$$

$$\to_\beta \mathrm{SqFill}_\Pi \left(\textbf{SqFill}\ \underbrace{(\mathbb{N} \times \mathrm{Bool})}_{A \times B} \right)$$

$$\to_\beta \mathrm{SqFill}_\Pi(\mathrm{SqFill}_\times(\textbf{SqFill}\ \mathbb{N})(\textbf{SqFill}\ \mathrm{Bool}))$$

⚠️ Bad news: SqFill doesn't compute (naively)

Naïvely adding **SqFill** as a **postulate** blocks computation. If **SqFill** appears in a term:

... **SqFill** $(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool})$ ...

$\to_\beta$ ... **SqFill** $(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool}))$ ...

$\to_\beta$ ... **SqFill** $(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool}))$ ...

⋮

Everything else around it evaluates, but it doesn't.

Like x in $42 + x$, can never return a value.

Our fix: make SqFill **compute** by inspecting the type.

$$\textbf{SqFill}\ \underbrace{(\mathbb{N} \to (\mathbb{N} \times \mathrm{Bool}))}_{A \to B}$$

$$\to_\beta \mathsf{SqFill}_\Pi \left(\textbf{SqFill}\ \underbrace{(\mathbb{N} \times \mathrm{Bool})}_{A \times B} \right)$$

$$\to_\beta \mathsf{SqFill}_\Pi(\mathsf{SqFill}_\times(\textbf{SqFill}\ \mathbb{N})(\textbf{SqFill}\ \mathrm{Bool}))$$

$$\to_\beta \mathsf{SqFill}_\Pi(\mathsf{SqFill}_\times\ \mathsf{SqFill}_\mathbb{N}\ \mathsf{SqFill}_{\mathrm{Bool}})$$

Proof by induction on the type: reduces to base cases, and computes away.

Live Demo: Cubical-UIP in Agda

Cubical Agda: SqFill as a stuck axiom:

```
-- postulated, never reduces
postulate
  prim^sqFill : ∀ (A : Type) → SqFill A

-- doesn't hold automatically!
test-computes : sqFill-block refl refl ≡ refl
test-computes = {! refl !} -- LHS does not reduce
```

Cubical-UIP: SqFill that computes:

```
-- dispatches by type, always reduces
primitive
  prim^sqFill : ∀ (A : Type) → SqFill A

-- computes!
test-computes : sqFill-block refl refl ≡ refl
test-computes = refl
```

 Live Demo with **Cubical-UIP**!

Behind the scenes: how SqFill computes

When Agda sees SqFill T, it **inspects the type** T and applies a proof for the outermost type former:

Behind the scenes: how SqFill computes

When Agda sees SqFill T, it **inspects the type** T and applies a proof for the outermost type former:

$$\textbf{SqFill}\ (A \times B) \quad \to_\beta \quad \mathsf{SqFill}_\times \underbrace{(\textbf{SqFill}\ A)}_{\text{recurse}} \underbrace{(\textbf{SqFill}\ B)}_{\text{recurse}}$$

$$\textbf{SqFill}\ (A \to B) \quad \to_\beta \quad \mathsf{SqFill}_\Pi \underbrace{(\textbf{SqFill}\ B)}_{\text{recurse}}$$

Behind the scenes: how SqFill computes

When Agda sees SqFill T, it **inspects the type** T and applies a proof for the outermost type former:

$$\textbf{SqFill}\ (A \times B) \quad \to_\beta \quad \mathrm{SqFill}_\times \underbrace{(\textbf{SqFill}\ A)}_{\text{recurse}} \underbrace{(\textbf{SqFill}\ B)}_{\text{recurse}}$$

$$\textbf{SqFill}\ (A \to B) \quad \to_\beta \quad \mathrm{SqFill}_\Pi \underbrace{(\textbf{SqFill}\ B)}_{\text{recurse}}$$

Base cases: types where equality is obviously unique, a proof is directly given.

$$\textbf{SqFill}\ \mathrm{Bool} \quad \to_\beta \quad \mathrm{SqFill}_{\mathrm{Bool}}$$

$$\textbf{SqFill}\ \mathbb{N} \quad \to_\beta \quad \mathrm{SqFill}_{\mathbb{N}}$$

Behind the scenes: how SqFill computes

When Agda sees **SqFill** T, it **inspects the type** T and applies a proof for the outermost type former:

$$\textbf{SqFill}\ (A \times B) \quad \to_\beta \quad \mathrm{SqFill}_\times \underbrace{(\textbf{SqFill}\ A)}_{\text{recurse}} \underbrace{(\textbf{SqFill}\ B)}_{\text{recurse}}$$

$$\textbf{SqFill}\ (A \to B) \quad \to_\beta \quad \mathrm{SqFill}_\Pi \underbrace{(\textbf{SqFill}\ B)}_{\text{recurse}}$$

Base cases: types where equality is obviously unique, a proof is directly given.

$$\textbf{SqFill}\ \mathrm{Bool} \quad \to_\beta \quad \mathrm{SqFill}_{\mathrm{Bool}}$$

$$\textbf{SqFill}\ \mathbb{N} \quad \to_\beta \quad \mathrm{SqFill}_{\mathbb{N}}$$

Proof by induction on the type: the type checker assembles the proof automatically.

Summary: SqFill for each type former

Full Agda proofs with beautiful diagrams [TND25] available.

Type former		**Difficulty**
$\Pi(a : A).B\ a$ (functions)[1]	:	Trivial: pointwise, no path manipulations needed[2]
$\Sigma(a : A).B\ a$ (dependent pairs)[3]	:	**Involved**: complex path manipulations[4]
$A \uplus B$ (coproducts)	:	Standard: encode, manipulate, decode
$\mathbb{N}$ (natural numbers)	:	Standard: encode, manipulate, decode
$a =_A b$ (path types)	:	Simple: a single path composition[5]
$\mathcal{U}$ (the universe)	:	**Open question**: see next slide

[1] Non-dependent ($A \to B$) is a special case: also trivial.

[2] No Kan operations (hcomp/transport) required.

[3] Non-dependent ($A \times B$) is a special case that's trivial.

[4] Requires transport-fill-align: transport along a line of types, fill a box, then align the result via hcomp.

[5] A single hcomp suffices.

Open question: what about the universe? 🪐

So far: SqFill computes for **concrete** type formers: Π, Σ, $A \uplus B$, $\mathbb{N}$, $a =_A b$.

Open question: what about the universe? 🪐

So far: SqFill computes for **concrete** type formers: Π, Σ, $A \uplus B$, $\mathbb{N}$, $a =_A b$.

What about the **universe** $\mathcal{U}$: the type of **all types**?

$$\text{SqFill } \mathcal{U} \quad \to_\beta \quad ???$$

Open question: what about the universe? 🪐

So far: SqFill computes for **concrete** type formers: Π, Σ, $A \uplus B$, $\mathbb{N}$, $a =_A b$.

What about the **universe** $\mathcal{U}$: the type of **all types**?

$$\text{SqFill } \mathcal{U} \quad \to_\beta \quad ???$$

Two approaches, both open:

- **Approach A**: treat $\mathcal{U}$ as an inductive type with one constructor per type former — inspect it the same way we inspect $A \times B$, $A \to B$, etc.[1]
- **Approach B** (speculative — merely an idea at this stage): SqFill becomes a new type former itself, with the so-called ***Weld*** types[2].

[1] Formally: an inductive-recursive (IR) universe with a typecase operator, in the style of Dybjer [Dyb00] / Dybjer-Setzer [DS99] / XTT [SAG22]. This makes the type theory anti-classical [CD18, Hur10], invalidating excluded middle — but univalence is already disabled.

[2] Based on Andreas Nuyts's presheaf semantics [NPD20]. Weld types generalise Glue types, key problem here is perhaps show Kan fibrancy.

Open question: what about the universe? 🪐

So far: SqFill computes for **concrete** type formers: Π, Σ, $A \uplus B$, $\mathbb{N}$, $a =_A b$.

What about the **universe** $\mathcal{U}$: the type of **all types**?

$$\text{SqFill } \mathcal{U} \quad \to_\beta \quad ???$$

Two approaches, both open:

- **Approach A**: treat $\mathcal{U}$ as an inductive type with one constructor per type former — inspect it the same way we inspect $A \times B$, $A \to B$, etc.[1]
- **Approach B** (speculative — merely an idea at this stage): SqFill becomes a new type former itself, with the so-called ***Weld*** types[2].

Ask us after the talk if you want to know more!

---

[1] Formally: an inductive-recursive (IR) universe with a typecase operator, in the style of Dybjer [Dyb00] / Dybjer-Setzer [DS99] / XTT [SAG22]. This makes the type theory anti-classical [CD18, Hur10], invalidating excluded middle — but univalence is already disabled.

[2] Based on Andreas Nuyts's presheaf semantics [NPD20]. Weld types generalise Glue types, key problem here is perhaps show Kan fibrancy.

Conclusion: Cubical Agda with funext, quotients, and UIP — and it computes

TL;DR: Want all of the following?

- **Funext**: pointwise equal functions are equal
- **Quotient types**: define types by their equalities
- **UIP**: at most one proof per equality, and it **computes**

Conclusion: Cubical Agda with funext, quotients, and UIP — and it computes

TL;DR: Want all of the following?

- **Funext**: pointwise equal functions are equal
- **Quotient types**: define types by their equalities
- **UIP**: at most one proof per equality, and it **computes**

You are in luck! We implemented these Agda variants:

✅ `--cubical=no-glue`: removes the one feature (*Glue*) incompatible with UIP

✅ `--cubical=uip`: SqFill computation rules for **Pi**, **Sigma**, **Nat**, **Bool**, **Unit**

Conclusion: Cubical Agda with funext, quotients, and UIP — and it computes

TL;DR: Want all of the following?

- **Funext**: pointwise equal functions are equal
- **Quotient types**: define types by their equalities
- **UIP**: at most one proof per equality, and it **computes**

You are in luck! We implemented these Agda variants:

✅ `--cubical=no-glue`: removes the one feature (*Glue*) incompatible with UIP

✅ `--cubical=uip`: SqFill computation rules for **Pi**, **Sigma**, **Nat**, **Bool**, **Unit**

Work in progress: 🚧

- Universe case: IR universe via typecases
- Path types and generic inductive types
- Canonicity and Normalisation
- Evaluate advantage in projects that needed Computational UIP [CNT26, HC21]

Thank you!

Questions?

https://github.com/yeejian-tan/agda/tree/cubical-uip

References

[BCZ25] Calvin Beck, Hanxi Chen, and Steve Zdancewic. 2025. Vellvm: Formalizing the Informal LLVM - (Experience Report). In *NASA Formal Methods - 17th International Symposium, NFM 2025, Williamsburg, VA, USA, June 11-13, 2025, Proceedings (Lecture Notes in Computer Science)*, 2025. Springer, 91–99. https://doi.org/10.1007/978-3-031-93706-4_6

[CNT26] 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)*, 2026. Association for Computing Machinery, Rennes, France, 201–215. https://doi.org/10.1145/3779031.3779090

[CD18] Jesper Cockx and Dominique Devriese. 2018. Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory. *J. Funct. Program.* 28, (2018), e12. https://doi.org/10.1017/S095679681800014X

[Coh+17] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2017. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. *FLAP* 4, 10 (2017), 3127–3170. Retrieved from http://collegepublications.co.uk/ifcolog/?00019

[DS99] Peter Dybjer and Anton Setzer. 1999. A Finite Axiomatization of Inductive-Recursive Definitions. In *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings (Lecture Notes in Computer Science)*, 1999. Springer, 129–146. https://doi.org/10.1007/3-540-48959-2_11

[Dyb00] Peter Dybjer. 2000. A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *J. Symb. Log.* 65, 2 (2000), 525–549. https://doi.org/10.2307/2586554

References

[HC21] Jason Z. S. Hu and Jacques Carette. 2021. Formalizing category theory in Agda. In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, 2021. ACM, 327–342. https://doi.org/10.1145/3437992.3439922

[Hur10] Chung Kil Hur. 2010. [Coq-Club] Agda with the excluded middle is inconsistent?. Retrieved from https://sympa.inria.fr/sympa/arc/coq-club/2010-01/msg00007.html

[Ler+16] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, January 2016. Toulouse, France. Retrieved from https://inria.hal.science/hal-01238879

[NPD20] Andreas Nuyts, Frank Piessens, and Dominique Devriese. 2020. Contributions to Multimode and Presheaf Type Theory.

[SAG22] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. 2022. A Cubical Language for Bishop Sets. *Log. Methods Comput. Sci.* 18, 1 (2022). https://doi.org/10.46298/LMCS-18(1:43)2022

[TND25] Yee-Jian Tan, Andreas Nuyts, and Dominique Devriese. 2025. Towards Computational UIP in Cubical Agda. Retrieved from https://arxiv.org/abs/2511.21209

[Tan25] Yee-Jian Tan. 2025. Cubical — Agda 2.9.0 Documentation : Cubical Agda without Glue. Retrieved from https://agda.readthedocs.io/en/latest/language/cubical.html#cubical-without-glue

[Uni13] The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study. Retrieved from https://homotopytypetheory.org/book

References

[Wu+26] Jinhua Wu, Yuting Wang, Liukun Yu, and Linglong Meng. 2026. RustCompCert: A Verified and Verifying Compiler for a Sequential Subset of Rust. Retrieved from https://arxiv.org/abs/2602.07455