

Definitional Proof Irrelevance Made Accessible

Thiago Felicissimo, Yann Leray, Loïc Pujet, Nicolas Tabareau, Éric Tanter & Théo Winterhalter

14 April 2026 @ Liberabaci meeting

INRIA

Prop = Type?

Prop = Type?

Propositions

Prop = Type?

Propositions

Types

Prop = Type?

Propositions \neq Types

As in First-Order Logic (FOL), Higher-Order Logic (HOL), ISABELLE

Prop = Type?

Propositions

Types

Prop = Type?

Propositions = Types

As in Martin-Löf Type Theory (MLTT)

Prop = Type?

Propositions

Types

Prop = Type?

Propositions \subset Types

As in ROCQ, AGDA, LEAN: we have type universes `Type` and `Prop`

Prop = Type?

Propositions \subset Types

As in ROCQ, AGDA, LEAN: we have type universes `Type` and `Prop`

Motto Distinguish *relevant* data of type $A : \text{Type}$ from *irrelevant* proofs of propositions $P : \text{Prop}$

If $P : \text{Prop}$ and $p, q : P$ we might want $p = q$, but we always want `true` \neq `false`

Ensuring compatibility of Prop with irrelevance

Constructing *relevant* data in **Type** from *irrelevant* proofs in **Prop** can be dangerous:

$$\left(\begin{array}{l} \lambda p. \text{ match } p \text{ with} \\ \quad | \text{ inl } a \rightarrow \text{ true} \\ \quad | \text{ inr } b \rightarrow \text{ false} \end{array} \right) : A \vee B \rightarrow \text{ Bool}$$

Ensuring compatibility of Prop with irrelevance

Constructing *relevant* data in `Type` from *irrelevant* proofs in `Prop` can be dangerous:

$$\left(\begin{array}{l} \lambda p. \text{ match } p \text{ with} \\ \quad | \text{ inl } a \rightarrow \text{ true} \\ \quad | \text{ inr } b \rightarrow \text{ false} \end{array} \right) : A \vee B \rightarrow \text{ Bool}$$

If `inl a = inr b` then `true = false`, yet we can prove that `true ≠ false`...

Ensuring compatibility of Prop with irrelevance

Constructing *relevant* data in **Type** from *irrelevant* proofs in **Prop** can be dangerous:

$$\left(\begin{array}{l} \lambda p. \text{ match } p \text{ with} \\ \quad | \text{ inl } a \rightarrow \text{ true} \\ \quad | \text{ inr } b \rightarrow \text{ false} \end{array} \right) : A \vee B \rightarrow \text{ Bool}$$

If $\text{inl } a = \text{inr } b$ then $\text{true} = \text{false}$, yet we can prove that $\text{true} \neq \text{false}$...

In Rocq, *subsingleton criterion* allows to construct data in **Type** from three propositions:

Ensuring compatibility of Prop with irrelevance

Constructing *relevant* data in **Type** from *irrelevant* proofs in **Prop** can be dangerous:

$$\left(\begin{array}{l} \lambda p. \text{ match } p \text{ with} \\ \quad | \text{ inl } a \rightarrow \text{ true} \\ \quad | \text{ inr } b \rightarrow \text{ false} \end{array} \right) : A \vee B \rightarrow \text{ Bool}$$

If $\text{inl } a = \text{inr } b$ then $\text{true} = \text{false}$, yet we can prove that $\text{true} \neq \text{false}$...

In Rocq, *subsingleton criterion* allows to construct data in **Type** from three propositions:

1. the empty type **False**

Ensuring compatibility of Prop with irrelevance

Constructing *relevant* data in **Type** from *irrelevant* proofs in **Prop** can be dangerous:

$$\left(\begin{array}{l} \lambda p. \text{ match } p \text{ with} \\ \quad | \text{ inl } a \rightarrow \text{ true} \\ \quad | \text{ inr } b \rightarrow \text{ false} \end{array} \right) : A \vee B \rightarrow \text{ Bool}$$

If $\text{inl } a = \text{inr } b$ then $\text{true} = \text{false}$, yet we can prove that $\text{true} \neq \text{false}$...

In Rocq, *subsingleton criterion* allows to construct data in **Type** from three propositions:

1. the empty type **False**
2. the equality type =

Ensuring compatibility of Prop with irrelevance

Constructing *relevant* data in **Type** from *irrelevant* proofs in **Prop** can be dangerous:

$$\left(\begin{array}{l} \lambda p. \text{ match } p \text{ with} \\ \quad | \text{ inl } a \rightarrow \text{ true} \\ \quad | \text{ inr } b \rightarrow \text{ false} \end{array} \right) : A \vee B \rightarrow \text{Bool}$$

If $\text{inl } a = \text{inr } b$ then $\text{true} = \text{false}$, yet we can prove that $\text{true} \neq \text{false}$...

In Rocq, *subsingleton criterion* allows to construct data in **Type** from three propositions:

1. the empty type **False**
2. the equality type $=$
3. the accessibility predicate **Acc**

Ensuring compatibility of Prop with irrelevance

Constructing *relevant* data in **Type** from *irrelevant* proofs in **Prop** can be dangerous:

$$\left(\begin{array}{l} \lambda p. \text{ match } p \text{ with} \\ \quad | \text{ inl } a \rightarrow \text{ true} \\ \quad | \text{ inr } b \rightarrow \text{ false} \end{array} \right) : A \vee B \rightarrow \text{ Bool}$$

If $\text{inl } a = \text{inr } b$ then $\text{true} = \text{false}$, yet we can prove that $\text{true} \neq \text{false}$...

In Rocq, *subsingleton criterion* allows to construct data in **Type** from three propositions:

1. the empty type **False**
2. the equality type $=$
3. the accessibility predicate **Acc**

Ensures consistency of **Prop** with proof-irrel : $\forall (P : \text{Prop}) (p q : P). p = q$

Ensuring compatibility of Prop with irrelevance

Constructing *relevant* data in **Type** from *irrelevant* proofs in **Prop** can be dangerous:

$$\left(\begin{array}{l} \lambda p. \text{ match } p \text{ with} \\ \quad | \text{ inl } a \rightarrow \text{ true} \\ \quad | \text{ inr } b \rightarrow \text{ false} \end{array} \right) : A \vee B \rightarrow \text{ Bool}$$

If $\text{inl } a = \text{inr } b$ then $\text{true} = \text{false}$, yet we can prove that $\text{true} \neq \text{false}$...

In Rocq, *subsingleton criterion* allows to construct data in **Type** from three propositions:

1. the empty type **False**
2. the equality type $=$
3. the accessibility predicate **Acc**

Ensures consistency of **Prop** with proof-irrel : $\forall (P : \text{Prop}) (p q : P). p = q$

Enables *extraction* into common programming languages (eg OCaml)

Accessibility: Positive well-foundedness

$$\frac{R : A \rightarrow A \rightarrow \text{Prop} \quad a : A}{\text{Acc } R \ a : \text{Prop}}$$

Accessibility: Positive well-foundedness

$$\frac{R : A \rightarrow A \rightarrow \text{Prop} \quad a : A}{\text{Acc } R \ a : \text{Prop}}$$

$$\frac{p : \forall (b : A). R \ b \ a \rightarrow \text{Acc } R \ b}{\text{acc-in } p : \text{Acc } R \ a}$$

Accessibility: Positive well-foundedness

$$\frac{R : A \rightarrow A \rightarrow \text{Prop} \quad a : A}{\text{Acc } R a : \text{Prop}}$$

$$\frac{p : \forall (b : A). R b a \rightarrow \text{Acc } R b}{\text{acc-in } p : \text{Acc } R a}$$

$$\frac{P : A \rightarrow \text{Type} \quad p : \forall (a : A) (rec : \forall (b : A). R b a \rightarrow P b). P a}{\text{acc-el } P p : \forall (a : A). \text{Acc } R a \rightarrow P a}$$

Accessibility: Positive well-foundedness

$$\frac{R : A \rightarrow A \rightarrow \text{Prop} \quad a : A}{\text{Acc } R a : \text{Prop}} \qquad \frac{p : \forall (b : A). R b a \rightarrow \text{Acc } R b}{\text{acc-in } p : \text{Acc } R a}$$
$$\frac{P : A \rightarrow \text{Type} \quad p : \forall (a : A) (rec : \forall (b : A). R b a \rightarrow P b). P a}{\text{acc-el } P p : \forall (a : A). \text{Acc } R a \rightarrow P a}$$

Used in proof assistants to elaborate definitions by well-founded recursion:

```
Equations? gcd (x y : nat) : nat by wf (x + y) lt :=
gcd 0 x := x ;      gcd x 0 := x ;
gcd x y with gt_eq_gt_dec x y := {
| inleft (left _) := gcd x (y - x) ;
| inleft (right refl) := x ;
| inright _ := gcd (x - y) y }.
Proof. all: lia. Defined.
```

Propositional irrelevance versus definitional irrelevance

Propositional proof irrelevance If $P : \text{Prop}$ and $p, q : P$, then there is $e : p = q$

Any two $(n, p), (n, q)$ of type $\Sigma(x : \text{Nat}). x > 0$ are always propositionally equal

But might not be *convertible*: the user needs to manually transport terms

Propositional irrelevance versus definitional irrelevance

Propositional proof irrelevance If $P : \text{Prop}$ and $p, q : P$, then there is $e : p = q$

Any two $(n, p), (n, q)$ of type $\Sigma(x : \text{Nat}). x > 0$ are always propositionally equal

But might not be *convertible*: the user needs to manually transport terms

Definitional proof irrelevance If $P : \text{Prop}$ and $p, q : P$, then $p \equiv q$ (p and q are *convertible*)

Now, $(n, p), (n, q)$ are always interchangeable, closer to mathematical practice

Integrated into type-theoretic proof assistants (AGDA, ROCQ & LEAN) in universe **SProp**

Propositional irrelevance versus definitional irrelevance

Propositional proof irrelevance If $P : \text{Prop}$ and $p, q : P$, then there is $e : p = q$

Any two $(n, p), (n, q)$ of type $\Sigma(x : \text{Nat}). x > 0$ are always propositionally equal

But might not be *convertible*: the user needs to manually transport terms

Definitional proof irrelevance If $P : \text{Prop}$ and $p, q : P$, then $p \equiv q$ (p and q are *convertible*)

Now, $(n, p), (n, q)$ are always interchangeable, closer to mathematical practice

Integrated into type-theoretic proof assistants (AGDA, ROCQ & LEAN) in universe $S\text{Prop}$

Aside from helping users, crucial for many type-theoretic constructions:

- setoid model of Altenkirch
- strict presheaves of Pédrot

Failure of the subsingleton criterion

Unfortunately, subsingleton criterion interacts badly with $SProp$...

Failure of the subsingleton criterion

Unfortunately, subsingleton criterion interacts badly with `SProp`...

Eliminating from `SProp` to `Type` with

Failure of the subsingleton criterion

Unfortunately, subsingleton criterion interacts badly with $SProp$...

Eliminating from $SProp$ to $Type$ with

- $False$: Still ok

Failure of the subsingleton criterion

Unfortunately, subsingleton criterion interacts badly with $SProp$...

Eliminating from $SProp$ to $Type$ with

- **False**: Still ok
- Equality (=): In presence of *function extensionality*, breaks *canonicity*, stating that

$\vdash t : \mathbf{Nat}$ implies $\vdash t \equiv S^n(0) : \mathbf{Nat}$ for some n

Failure of the subsingleton criterion

Unfortunately, subsingleton criterion interacts badly with $SProp$...

Eliminating from $SProp$ to $Type$ with

- **False**: Still ok
- Equality (=): In presence of *function extensionality*, breaks *canonicity*, stating that

$\vdash t : \mathbf{Nat}$ implies $\vdash t \equiv S^n(0) : \mathbf{Nat}$ for some n

Moreover, in presence of impredicativity, breaks proof normalization

Failure of the subsingleton criterion

Unfortunately, subsingleton criterion interacts badly with $SProp$...

Eliminating from $SProp$ to $Type$ with

- **False**: Still ok
- Equality (=): In presence of *function extensionality*, breaks *canonicity*, stating that

$\vdash t : \mathbf{Nat}$ implies $\vdash t \equiv S^n(0) : \mathbf{Nat}$ for some n

Moreover, in presence of impredicativity, breaks proof normalization

- **Acc**: Breaks decidability of conversion/type checking (= proof checking)

Failure of the subsingleton criterion

Unfortunately, subsingleton criterion interacts badly with $SProp$...

Eliminating from $SProp$ to $Type$ with

- **False**: Still ok
- Equality (=): In presence of *function extensionality*, breaks *canonicity*, stating that

$\vdash t : \mathbf{Nat}$ implies $\vdash t \equiv S^n(0) : \mathbf{Nat}$ for some n

Moreover, in presence of impredicativity, breaks proof normalization

- **Acc**: Breaks decidability of conversion/type checking (= proof checking)

Inconsistent contexts can lie about accessibility proofs

$x : \mathbf{False} \vdash p : \mathbf{Acc} (\lambda xy. \mathbf{True}) a$

Failure of the subsingleton criterion

Unfortunately, subsingleton criterion interacts badly with $SProp$...

Eliminating from $SProp$ to $Type$ with

- **False**: Still ok
- Equality (=): In presence of *function extensionality*, breaks *canonicity*, stating that

$\vdash t : \mathbf{Nat}$ implies $\vdash t \equiv S^n(0) : \mathbf{Nat}$ for some n

Moreover, in presence of impredicativity, breaks proof normalization

- **Acc**: Breaks decidability of conversion/type checking (= proof checking)

Inconsistent contexts can lie about accessibility proofs

$x : \mathbf{False} \vdash p : \mathbf{Acc} (\lambda xy. \mathbf{True}) a$

From $p : \mathbf{Acc} R a$ we can construct **acc-in** $(\text{acc-inv } p) : \mathbf{Acc} R a$

Failure of the subsingleton criterion

Unfortunately, subsingleton criterion interacts badly with $SProp$...

Eliminating from $SProp$ to $Type$ with

- **False**: Still ok
- Equality (=): In presence of *function extensionality*, breaks *canonicity*, stating that

$\vdash t : \mathbf{Nat}$ implies $\vdash t \equiv S^n(0) : \mathbf{Nat}$ for some n

Moreover, in presence of impredicativity, breaks proof normalization

- **Acc**: Breaks decidability of conversion/type checking (= proof checking)

Inconsistent contexts can lie about accessibility proofs

$x : \mathbf{False} \vdash p : \mathbf{Acc} (\lambda xy. \mathbf{True}) a$

From $p : \mathbf{Acc} R a$ we can construct **acc-in** $(\text{acc-inv } p) : \mathbf{Acc} R a$

Definitional proof irrelevant implies they are convertible

So any **acc-el** can be evaluated, even if termination argument can be bogus

Dealing with **SProp** in proof assistants

AGDA, Rocq: Only **False** can be eliminated into **Type**

Preserves good properties of the theory, but highly limits applicability

In practice, users still need to resort to **Prop**...

Dealing with **SProp** in proof assistants

AGDA, Rocq: Only **False** can be eliminated into **Type**

Preserves good properties of the theory, but highly limits applicability

In practice, users still need to resort to **Prop**...

LEAN: Implements full subsingleton criterion, and lives with its consequences

No canonicity, because (1) elimination of equality into **Type**, and (2) Hilbert's ε operator

Undecidable type-checking, implementation less stable

Since v4.19.0, undecidability hidden by elaboration when defining functions by well-founded rec.

Dealing with **SProp** in proof assistants

AGDA, Rocq: Only **False** can be eliminated into **Type**

Preserves good properties of the theory, but highly limits applicability

In practice, users still need to resort to **Prop**...

LEAN: Implements full subsingleton criterion, and lives with its consequences

No canonicity, because (1) elimination of equality into **Type**, and (2) Hilbert's ε operator

Undecidable type-checking, implementation less stable

Since v4.19.0, undecidability hidden by elaboration when defining functions by well-founded rec.

Our goal A design combining **SProp** with equality and **Acc**, but preserving good properties

Reconciling = with **SProp**, with *observational equality*

As remarked by Pujet & Tabareau, problems of **SProp** with = solved by McBride's & Altenkirch's *observational equality*

Reconciling = with **SProp**, with *observational equality*

As remarked by Pujet & Tabareau, problems of **SProp** with = solved by McBride's & Altenkirch's *observational equality*

Instead of usual **J** eliminator, eliminated using a *cast* operator:

$$\frac{A, B : \text{Type} \quad p : A = B \quad a : A}{\text{cast}_p^{A \rightsquigarrow B}(a) : B}$$

Reconciling = with **SProp**, with *observational equality*

As remarked by Pujet & Tabareau, problems of **SProp** with = solved by McBride's & Altenkirch's *observational equality*

Instead of usual **J** eliminator, eliminated using a *cast* operator:

$$\frac{A, B : \text{Type} \quad p : A = B \quad a : A}{\text{cast}_p^{A \rightsquigarrow B}(a) : B}$$

Crucial property Unlike **J**, **cast** computes by case analysis on types:

$$\text{cast}_p^{(A \times B) \rightsquigarrow (A' \times B')} t \longrightarrow \langle \text{cast}_{p.1}^{A \rightsquigarrow A'}(\pi_1 t), \text{cast}_{p.2}^{B \rightsquigarrow B'}(\pi_2 t) \rangle$$

Reconciling = with **SProp**, with *observational equality*

As remarked by Pujet & Tabareau, problems of **SProp** with = solved by McBride's & Altenkirch's *observational equality*

Instead of usual **J** eliminator, eliminated using a *cast* operator:

$$\frac{A, B : \text{Type} \quad p : A = B \quad a : A}{\text{cast}_p^{A \rightsquigarrow B}(a) : B}$$

Crucial property Unlike **J**, **cast** computes by case analysis on types:

$$\text{cast}_p^{(A \times B) \rightsquigarrow (A' \times B')} t \longrightarrow \langle \text{cast}_{p.1}^{A \rightsquigarrow A'}(\pi_1 t), \text{cast}_{p.2}^{B \rightsquigarrow B'}(\pi_2 t) \rangle$$

Rules *never look inside equality proofs*, hence axioms (funext, propext, etc) cannot block reduction!

Reconciling = with **SProp**, with *observational equality*

As remarked by Pujet & Tabareau, problems of **SProp** with = solved by McBride's & Altenkirch's *observational equality*

Instead of usual **J** eliminator, eliminated using a *cast* operator:

$$\frac{A, B : \text{Type} \quad p : A = B \quad a : A}{\text{cast}_p^{A \rightsquigarrow B}(a) : B}$$

Crucial property Unlike **J**, **cast** computes by case analysis on types:

$$\text{cast}_p^{(A \times B) \rightsquigarrow (A' \times B')} t \longrightarrow \langle \text{cast}_{p.1}^{A \rightsquigarrow A'}(\pi_1 t), \text{cast}_{p.2}^{B \rightsquigarrow B'}(\pi_2 t) \rangle$$

Rules *never look inside equality proofs*, hence axioms (funext, propext, etc) cannot block reduction!

Combined with **SProp** yields CC^{obs} , well-behaved theory for set-level mathematics

Enjoys canonicity, decidability of typing and consistency, as shown by Pujet & Tabareau

Usual **J** eliminator can be simulated using **cast** and **transp** (= **J** restricted to **SProp**)

This work: Reconciling Acc with SProp

We propose a design combining two theories that extend CC^{obs} with Acc:

This work: Reconciling Acc with SProp

We propose a design combining two theories that extend CC^{obs} with **Acc**:

$$\mathcal{T}_{\text{Acc}}^=$$

Eliminator **acc-el** computes up to =

This work: Reconciling Acc with SProp

We propose a design combining two theories that extend CC^{obs} with **Acc**:

$$\mathcal{T}_{\text{Acc}}^=$$

Eliminator **acc-el** computes up to =

Fact $\mathcal{T}_{\text{Acc}}^=$ has decidable type-checking

Extends CC^{obs} with just constants

This work: Reconciling Acc with SProp

We propose a design combining two theories that extend CC^{obs} with **Acc**:

$$\mathcal{T}_{\text{Acc}}^=$$

Eliminator **acc-el** computes up to =

Fact $\mathcal{T}_{\text{Acc}}^=$ has decidable type-checking

Extends CC^{obs} with just constants

But usual form of canonicity fails...

$\text{gcd } 8 \ 2 = 2$ does not follow from conversion

This work: Reconciling Acc with SProp

We propose a design combining two theories that extend CC^{obs} with **Acc**:

$$\mathcal{T}_{\text{Acc}}^= \quad \subset \quad \mathcal{T}_{\text{Acc}}^{\equiv}$$

Eliminator **acc-el** computes up to =

Fact $\mathcal{T}_{\text{Acc}}^=$ has decidable type-checking

Extends CC^{obs} with just constants

But usual form of canonicity fails...

$\text{gcd } 8 \ 2 = 2$ does not follow from conversion

Eliminator **acc-el** computes definitionally
Hence conversion/typing are undecidable

This work: Reconciling Acc with SProp

We propose a design combining two theories that extend CC^{obs} with **Acc**:

$$\mathcal{T}_{\text{Acc}}^= \quad \subset \quad \mathcal{T}_{\text{Acc}}^{\equiv}$$

Eliminator **acc-el** computes up to =

Fact $\mathcal{T}_{\text{Acc}}^=$ has decidable type-checking

Extends CC^{obs} with just constants

But usual form of canonicity fails...

Eliminator **acc-el** computes definitionally
Hence conversion/typing are undecidable

Thm $\mathcal{T}_{\text{Acc}}^{\equiv}$ satisfies canonicity

$\text{gcd } 8 \ 2 = 2$ follows from conversion

This work: Reconciling Acc with SProp

We propose a design combining two theories that extend CC^{obs} with **Acc**:



Eliminator **acc-el** computes up to =

Fact $\mathcal{T}_{\text{Acc}}^=$ has decidable type-checking

Extends CC^{obs} with just constants

But usual form of canonicity fails...

Eliminator **acc-el** computes definitionally
Hence conversion/typing are undecidable

Thm $\mathcal{T}_{\text{Acc}}^{\equiv}$ satisfies canonicity

$\text{gcd } 8 \ 2 = 2$ follows from conversion

Thm $\mathcal{T}_{\text{Acc}}^{\equiv}$ is conservative over $\mathcal{T}_{\text{Acc}}^=$:

If $\vdash^= P : \mathbf{SProp}$ and $\vdash^{\equiv} p : P$ then $\vdash^= q : P$ for some q

This work: Reconciling Acc with SProp

We propose a design combining two theories that extend CC^{obs} with **Acc**:



Eliminator **acc-el** computes up to =

Fact $\mathcal{T}_{\text{Acc}}^=$ has decidable type-checking

Extends CC^{obs} with just constants

Cor $\mathcal{T}_{\text{Acc}}^=$ enjoys *propositional* canonicity:

If $\vdash t : \mathbf{Nat}$ then $\vdash e : t = \mathbf{S}^n(\mathbf{0})$ for some e, n

Eliminator **acc-el** computes definitionally
Hence conversion/typing are undecidable

Thm $\mathcal{T}_{\text{Acc}}^{\equiv}$ satisfies canonicity

$\text{gcd } 8 \ 2 = 2$ follows from conversion

Thm $\mathcal{T}_{\text{Acc}}^{\equiv}$ is conservative over $\mathcal{T}_{\text{Acc}}^=$:

If $\vdash^= P : \mathbf{SProp}$ and $\vdash^{\equiv} p : P$ then $\vdash^= q : P$ for some q

This work: Reconciling Acc with SProp

We propose a design combining two theories that extend CC^{obs} with **Acc**:



Eliminator **acc-el** computes up to =

Fact $\mathcal{T}_{\text{Acc}}^=$ has decidable type-checking

Extends CC^{obs} with just constants

Cor $\mathcal{T}_{\text{Acc}}^=$ enjoys *propositional* canonicity:

If $\vdash t : \mathbf{Nat}$ then $\vdash e : t = \mathbf{S}^n(0)$ for some e, n

Eliminator **acc-el** computes definitionally
Hence conversion/typing are undecidable

Thm $\mathcal{T}_{\text{Acc}}^{\equiv}$ satisfies canonicity

$\text{gcd } 8 \ 2 = 2$ follows from conversion

Thm $\mathcal{T}_{\text{Acc}}^{\equiv}$ is conservative over $\mathcal{T}_{\text{Acc}}^=$:

If $\vdash^= P : \mathbf{SProp}$ and $\vdash^{\equiv} p : P$ then $\vdash^= q : P$ for some q

Thm $\mathcal{T}_{\text{Acc}}^=$ and $\mathcal{T}_{\text{Acc}}^{\equiv}$ admit set-theoretic models, in particular they are consistent

This work: Reconciling Acc with SProp

We propose a design combining two theories that extend CC^{obs} with **Acc**:



Eliminator **acc-el** computes up to =

Fact $\mathcal{T}_{\text{Acc}}^=$ has decidable type-checking

Extends CC^{obs} with just constants

Cor $\mathcal{T}_{\text{Acc}}^=$ enjoys *propositional* canonicity:

If $\vdash t : \mathbf{Nat}$ then $\vdash e : t = \mathbf{S}^n(0)$ for some e, n

Eliminator **acc-el** computes definitionally
Hence conversion/typing are undecidable

Thm $\mathcal{T}_{\text{Acc}}^{\equiv}$ satisfies canonicity

$\text{gcd } 8 \ 2 = 2$ follows from conversion

Thm $\mathcal{T}_{\text{Acc}}^{\equiv}$ is conservative over $\mathcal{T}_{\text{Acc}}^=$:

If $\vdash^= P : \mathbf{SProp}$ and $\vdash^{\equiv} p : P$ then $\vdash^= q : P$ for some q

Thm $\mathcal{T}_{\text{Acc}}^=$ and $\mathcal{T}_{\text{Acc}}^{\equiv}$ admit set-theoretic models, in particular they are consistent

Results formalized in Rocq

The design in practice

$\mathcal{T}_{\text{Acc}}^=$ implemented on top of observational fork of RocQ of Pujet, Leray & Tabareau

Proof mode switch to $\mathcal{T}_{\text{Acc}}^{\equiv}$ to ease proofs, justified by conservativity theorem

The design in practice

$\mathcal{T}_{\text{Acc}}^=$ implemented on top of observational fork of RocQ of Pujet, Leray & Tabareau

Proof mode switch to $\mathcal{T}_{\text{Acc}}^{\equiv}$ to ease proofs, justified by conservativity theorem

In $\mathcal{T}_{\text{Acc}}^=$:

```
Lemma gcd_test : (gcd (2 ^ N) 2 <? 5) ~ true.
```

```
Proof.
```

```
  auto_Acc_unfold; reflexivity.
```

```
Qed.
```

The design in practice

$\mathcal{T}_{\text{Acc}}^=$ implemented on top of observational fork of RocQ of Pujet, Leray & Tabareau

Proof mode switch to $\mathcal{T}_{\text{Acc}}^{\equiv}$ to ease proofs, justified by conservativity theorem

In $\mathcal{T}_{\text{Acc}}^=$:

```
Lemma gcd_test : (gcd (2 ^ N) 2 <? 5) ~ true.
```

```
Proof.
```

```
  auto_Acc_unfold; reflexivity.
```

```
Qed.
```

In $\mathcal{T}_{\text{Acc}}^{\equiv}$:

```
#[rewrite_rules(Acc_el_def)]
```

```
Lemma gcd_test_def (gcd (2 ^ N) 2 <? 5) ~ true.
```

```
Proof.
```

```
  reflexivity.
```

```
Qed.
```

The design in practice

$\mathcal{T}_{\text{Acc}}^=$ implemented on top of observational fork of Rocq of Pujet, Leray & Tabareau

Proof mode switch to $\mathcal{T}_{\text{Acc}}^{\equiv}$ to ease proofs, justified by conservativity theorem

In $\mathcal{T}_{\text{Acc}}^=$:

```
Lemma gcd_test : (gcd (2 ^ N) 2 <? 5) ~ true.
```

```
Proof.
```

```
  auto_Acc_unfold; reflexivity.
```

```
Qed.
```

In $\mathcal{T}_{\text{Acc}}^{\equiv}$:

```
#[rewrite_rules(Acc_el_def)]
```

```
Lemma gcd_test_def (gcd (2 ^ N) 2 <? 5) ~ true.
```

```
Proof.
```

```
  reflexivity.
```

```
Qed.
```

For $N = 10$, takes 0.004 s

The design in practice

$\mathcal{T}_{\text{Acc}}^=$ implemented on top of observational fork of RocQ of Pujet, Leray & Tabareau

Proof mode switch to $\mathcal{T}_{\text{Acc}}^{\equiv}$ to ease proofs, justified by conservativity theorem

In $\mathcal{T}_{\text{Acc}}^=$:

```
Lemma gcd_test : (gcd (2 ^ N) 2 <? 5) ~ true.
```

```
Proof.
```

```
  auto_Acc_unfold; reflexivity.
```

```
Qed.
```

For $N = 10$, timeouts after 10 min...

In $\mathcal{T}_{\text{Acc}}^{\equiv}$:

```
#[rewrite_rules(Acc_el_def)]
```

```
Lemma gcd_test_def (gcd (2 ^ N) 2 <? 5) ~ true.
```

```
Proof.
```

```
  reflexivity.
```

```
Qed.
```

For $N = 10$, takes 0.004 s

The design in practice

$\mathcal{T}_{\text{Acc}}^=$ implemented on top of observational fork of RocQ of Pujet, Leray & Tabareau

Proof mode switch to $\mathcal{T}_{\text{Acc}}^{\equiv}$ to ease proofs, justified by conservativity theorem

In $\mathcal{T}_{\text{Acc}}^=$:

```
Lemma gcd_test : (gcd (2 ^ N) 2 <? 5) ~ true.
```

```
Proof.
```

```
  auto_Acc_unfold; reflexivity.
```

```
Qed.
```

For $N = 10$, timeouts after 10 min...

In $\mathcal{T}_{\text{Acc}}^{\equiv}$:

```
#[rewrite_rules(Acc_el_def)]
```

```
Lemma gcd_test_def (gcd (2 ^ N) 2 <? 5) ~ true.
```

```
Proof.
```

```
  reflexivity.
```

```
Qed.
```

For $N = 10$, takes 0.004 s

Crucial remark No need to implement costly translation from $\mathcal{T}_{\text{Acc}}^{\equiv}$ to $\mathcal{T}_{\text{Acc}}^=$:

Proofs are irrelevant, so can be treated as opaque!

The design in practice

$\mathcal{T}_{\text{Acc}}^=$ implemented on top of observational fork of Rocq of Pujet, Leray & Tabareau

Proof mode switch to $\mathcal{T}_{\text{Acc}}^{\equiv}$ to ease proofs, justified by conservativity theorem

In $\mathcal{T}_{\text{Acc}}^=$:

```
Lemma gcd_test : (gcd (2 ^ N) 2 <? 5) ~ true.
```

```
Proof.
```

```
  auto_Acc_unfold; reflexivity.
```

```
Qed.
```

For $N = 10$, timeouts after 10 min...

In $\mathcal{T}_{\text{Acc}}^{\equiv}$:

```
#[rewrite_rules(Acc_el_def)]
```

```
Lemma gcd_test_def (gcd (2 ^ N) 2 <? 5) ~ true.
```

```
Proof.
```

```
  reflexivity.
```

```
Qed.
```

For $N = 10$, takes 0.004 s

Crucial remark No need to implement costly translation from $\mathcal{T}_{\text{Acc}}^{\equiv}$ to $\mathcal{T}_{\text{Acc}}^=$:

Proofs are irrelevant, so can be treated as opaque!

Expected to be integrated in main version of Rocq in the future

Conclusion

We propose a design combining two theories with **SProp**, observational equality and **Acc**:



Eliminator **acc-el** computes up to =

- Has decidable type-checking
- Enjoys propositional canonicity

Eliminator **acc-el** computes definitionally

- Enjoys canonicity
- Conservative over $\mathcal{T}_{\text{Acc}}^=$

Both $\mathcal{T}_{\text{Acc}}^=$ and $\mathcal{T}_{\text{Acc}}^{\equiv}$ admit set-theoretic models, and are in particular consistent

Results have been formalized in Rocq: <https://github.com/thiagofelicissimo/acc-in-sprop>

$\mathcal{T}_{\text{Acc}}^=$ implemented on top of observational Rocq

Proof mode switch to $\mathcal{T}_{\text{Acc}}^{\equiv}$ allows proofs which are easier, and much faster to check

All details can be found in preprint: <https://hal.science/hal-05474391>

Conclusion

We propose a design combining two theories with **SProp**, observational equality and **Acc**:



Eliminator **acc-el** computes up to =

- Has decidable type-checking
- Enjoys propositional canonicity

Eliminator **acc-el** computes definitionally

- Enjoys canonicity
- Conservative over $\mathcal{T}_{\text{Acc}}^=$

Both $\mathcal{T}_{\text{Acc}}^=$ and $\mathcal{T}_{\text{Acc}}^{\equiv}$ admit set-theoretic models, and are in particular consistent

Results have been formalized in Rocq: <https://github.com/thiagofelicissimo/acc-in-sprop>

$\mathcal{T}_{\text{Acc}}^=$ implemented on top of observational Rocq

Proof mode switch to $\mathcal{T}_{\text{Acc}}^{\equiv}$ allows proofs which are easier, and much faster to check

All details can be found in preprint: <https://hal.science/hal-05474391>

Thank you for your attention!