The Quest for Confluence

Modular global confluence checking for type theory with rewrite rules

Jesper Cockx

Talk at the first Dutch CATS in Amsterdam
27 October 2021

My collaborators

Théo Winterhalter

Nicolas Tabareau

Jesper Cockx

Highlights of this talk

- Rewriting Type Theory (RTT): dependent type theory with **user-definable rewrite rules**
- A **modular** and **decidable** confluence criterion based on the triangle propert of parallel reduction
- An implementation of RTT and our confluence check as an extension to **Agda**
- A formal proof of confluence and subject reduction using **MetaCoq**

One step building on a long legacy

- Extensions of the Calculus of Constructions with rewrite rules [Barbamera et al 1997, Walukiewicz-Crzaszcz 2003, Blanqui 2005, ...]

One step building on a long legacy

- Extensions of the Calculus of Constructions with rewrite rules [Barbamera et al 1997, Walukiewicz-Crzaszcz 2003, Blanqui 2005, . . .]
- CoqMT(U), extending Coq with decidable first-order theory [Strub 2010, Barras et al 2011, . . .]

One step building on a long legacy

- Extensions of the Calculus of Constructions with rewrite rules [Barbamera et al 1997, Walukiewicz-Crzaszcz 2003, Blanqui 2005, ...]
- CoqMT(U), extending Coq with decidable first-order theory [Strub 2010, Barras et al 2011, ...]
- Dedukti, a logical framework based on rewrite rules [Cousineau and Dowek 2007, Boespflug et al 2012, Ferey and Jouannaud 2019, ...]

Dramatic arc of this talk

Part I Type theory unchained
(Everything is awesome!)

Part II Problems in the metatheory
(Everything is awful. . .)

Part III Global confluence checking
(Everything is ok again?)

Outline

1. Type Theory Unchained
2. Metatheory of RTT
3. Global confluence checking

Type theory as the foundation of modern proof assistants

Modern proof assistants (e.g. Coq & Agda) are based on Martin-Löf's **dependent type theory**.

- Lambda calculus at the core

Per Martin-Löf

Type theory as the foundation of modern proof assistants

Modern proof assistants (e.g. Coq & Agda) are based on Martin-Löf's **dependent type theory**.

- Lambda calculus at the core
- Dependent function space $(b : \mathbb{B}) \to \texttt{if}\ b\ \texttt{then}\ \mathbb{N}\ \texttt{else}\ \mathbb{B}$

Per Martin-Löf

Type theory as the foundation of modern proof assistants

Modern proof assistants (e.g. Coq & Agda) are based on Martin-Löf's **dependent type theory**.

- Lambda calculus at the core
- Dependent function space $(b : \mathbb{B}) \to \texttt{if } b \texttt{ then } \mathbb{N} \texttt{ else } \mathbb{B}$
- Universes: $\mathbb{B} : \texttt{Type}$, $\texttt{Type} : \texttt{Type}_1, \ldots$

Per Martin-Löf

Type theory as the foundation of modern proof assistants

Modern proof assistants (e.g. Coq & Agda) are based on Martin-Löf's **dependent type theory**.

- Lambda calculus at the core
- Dependent function space $(b : \mathbb{B}) \to \texttt{if}\ b\ \texttt{then}\ \mathbb{N}\ \texttt{else}\ \mathbb{B}$
- Universes: $\mathbb{B} : \texttt{Type}$, $\texttt{Type} : \texttt{Type}_1, \ldots$
- Identity type, inductive types, ...

Per Martin-Löf

The modular set-up of Martin-Löf Type Theory

Each type former is defined by four sets of rules:

Formation rule	$\mathbb{N} : \mathtt{Type}$
Introduction rules	$\mathtt{zero} : \mathbb{N}$ and $\mathtt{suc} : \mathbb{N} \to \mathbb{N}$
Elimination rule	$\mathtt{ind} : (P : \mathbb{N} \to \mathtt{Type})$ $\to P\ \mathtt{zero}$ $\to ((n : \mathbb{N}) \to P\ n \to P\ (\mathtt{suc}\ n))$ $\to (n : \mathbb{N}) \to P\ n$
Computation rules	$\mathtt{ind}\ P\ pz\ ps\ \mathtt{zero} \rightsquigarrow pz$ and $\mathtt{ind}\ P\ pz\ ps\ (\mathtt{suc}\ n) \rightsquigarrow ps\ n\ (\mathtt{ind}\ P\ pz\ ps\ n)$

The limitations of a proof assistant

In a proof assistant such as Agda & Coq, one cannot freely add new type formers.

Instead, one can define...

- inductive types that are *strictly positive*
- functions through *complete case splits*
- fixpoints that are *structurally recursive*

...but this is not always enough!

Two notions of equality in MLTT

Definitional equality	**Propositional equality**
$x = y$	$p : x \equiv_A y$
x and y have the *same normal form*	there is a *proof* that x and y are equal
$(\lambda x.x)\ 4 = 4$	$\texttt{refl} : (\lambda x.x)\ 4 \equiv_{\mathbb{N}} 4$
$x + y \neq y + x$	$\texttt{+-comm}\ x\ y : x + y \equiv_{\mathbb{N}} y + x$
fixed by the language	*can be extended with axioms*
checked automatically	*has to be applied manually*

Problem #1: Definitional equality is fragile

$$\_+\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$\texttt{zero} + y \quad = \; y$$
$$(\texttt{suc}\; x) + y \; = \; \texttt{suc}\; (x + y)$$

$$\texttt{comm} : (x\; y : \mathbb{N}) \to x + y \equiv_{\mathbb{N}} y + x$$
$$\texttt{comm}\; \texttt{zero}\; y \quad = \; \underline{\text{refl}}$$
$$\texttt{comm}\; (\texttt{suc}\; x)\; y \; = \; \boxed{\{\; \}0}$$

Agda protests: `y != y + zero of type` $\mathbb{N}$

Problem #2: Intensional equality is not extensible

```
postulate
   ℕ/2  : Type
   proj : ℕ → ℕ/2
   quot : (x y : ℕ) → x % 2 ≡ℕ y % 2 →
          proj x ≡ℕ/2 proj y
```

Problem #2: Intensional equality is not extensible

```
postulate
  ℕ/2  : Type
  proj : ℕ → ℕ/2
  quot : (x y : ℕ) → x % 2 ≡ℕ y % 2 →
         proj x ≡ℕ/2 proj y

  rec  : (f : ℕ → A) →
         (q : ∀x y → x % 2 ≡ℕ y % 2 → f x ≡A f y) →
         (x : ℕ/2) → A
```

The term `rec` f q (`proj` x) should evaluate to f x, but it is stuck!

A non-solution: equality reflection

Applying propositional equalities by hand is very verbose and error-prone.

Instead, we can consider adding the *equality reflection* rule:

$$\frac{\Gamma \vdash p : x \equiv_A y}{\Gamma \vdash x = y}$$

This solves the two problems by merging definitional and propositional equality.

However, it makes type checking *undecidable*.

Do we want equality to be decidable or extensible?

A non-solution: equality reflection

Applying propositional equalities by hand is very verbose and error-prone.

Instead, we can consider adding the *equality reflection* rule:

$$\frac{\Gamma \vdash p : x \equiv_A y}{\Gamma \vdash x = y}$$

This solves the two problems by merging definitional and propositional equality.

However, it makes type checking *undecidable*.

Do we want equality to be decidable or extensible? YES!

Rewrite rules to the rescue!

By adding **rewrite rules**, definitional equality becomes extensible while staying decidable.[1]

In a proof assistant with rewrite rules, we can...

1. Add computation rules to existing definitions:

$$\begin{aligned} x + \mathtt{zero} &\rightarrow x \\ x + (\mathtt{suc}\ y) &\rightarrow \mathtt{suc}\ (x + y) \end{aligned}$$

2. Postulate new primitives that compute:

$$\mathtt{rec}\ f\ q\ (\mathtt{proj}\ x) \rightarrow f\ x$$

[1] If we choose rewrite rules carefully.

Rewrite rules in practice

Demo time!

General shape of a rewrite rule

$$\underbrace{?x\ ?y\ ?z}_{\text{pattern variables}} \vdash f\ \underbrace{p_1\ \ldots\ p_n}_{\text{patterns}} \rightharpoonup t$$

General shape of a rewrite rule

$$\underbrace{?x\ ?y\ ?z}_{\text{pattern variables}} \vdash f\ \underbrace{p_1\ \ldots\ p_n}_{\text{patterns}} \rhd t$$

1. Pattern variables must be left-linear

General shape of a rewrite rule

$$\underbrace{?x\ ?y\ ?z}_{\text{pattern variables}} \vdash f\ \underbrace{p_1\ \ldots\ p_n}_{\text{patterns}} \rightharpoonup t$$

1. Pattern variables must be left-linear
2. f must be fresh (defined in same block)

General shape of a rewrite rule

$$\underbrace{?x\ ?y\ ?z}_{\text{pattern variables}} \vdash f\ \underbrace{p_1\ \ldots\ p_n}_{\text{patterns}} \rightharpoonup t$$

1. Pattern variables must be left-linear
2. f must be fresh (defined in same block)
3. No higher-order rules (for now)

Rewriting Type Theory (RTT) is Martin-Löf's type theory extended with user-defined rewrite rules of this shape.

Outline

1. Type Theory Unchained
2. Metatheory of RTT
3. Global confluence checking

Metatheory of MLTT 101

MLTT satisfies many ‘good’ properties:

Logical consistency
: There is no term u such that $\vdash u : \bot$

Decidable typechecking
: We can decide whether $\Gamma \vdash u : A$

Subject reduction
: If $\Gamma \vdash u : A$ and $u \rightsquigarrow v$ then $\Gamma \vdash v : A$

Do these properties still hold in a type theory with rewrite rules??

Logical consistency

Q: Doesn't a rewrite rule $0 \rightarrow 1$ breaks consistency?

Logical consistency

Q: Doesn't a rewrite rule $0 \rightharpoondown 1$ breaks consistency?

A: Yes, but this is no diffent from using `postulate`! We can regain soundness by requiring a **proof** for each rewrite rule.

Theorem (Consistency of RTT). If for each rewrite rule $l \rightharpoondown r$ we have a proof $\vdash e : l \equiv r$, then the system is consistent.

Soundness of type checking

Q: Doesn't a rewrite rule `loop` $\rightarrow$ `loop` break normalization, and hence decidable typechecking?

Soundness of type checking

Q: Doesn't a rewrite rule `loop` $\rightharpoonup$ `loop` break normalization, and hence decidable typechecking?

A: Yes it does, but the usual algorithm is still correct if it terminates!

Theorem (Soundness of typechecking for RTT). If type checking terminates successfully on input context Γ, term u, and type A, then $\Gamma \vdash u : A$.

Completeness of type checking

Q: What about completeness? If we have two rules $X \rightarrowtriangle \mathbb{N}$ and $X \rightarrowtriangle \mathbb{B}$ and $u : \mathbb{B}$, will type checking accept $u : X$?

A: No, for type checking to be complete we need **confluence** of reduction.

Theorem (Completeness of typechecking for RTT). *Assume that reduction with the given set of rewrite rules is confluent.* If $\Gamma \vdash u : A$, then type checking will not throw an error on input context Γ, term u, and type A.

Practical type checking

We say type checking is **practical** if it is sound and complete: when it terminates, it is correct.

- RTT with confluent reduction has practical type checking.
- Type theory with equality reflection does not.

The *only* thing that can go wrong is that the type checker loops because of a non-terminating set of rewrite rules.

Subject reduction

Q: Doesn't a rewrite rule `true` $\rightharpoonup$ 42 break subject reduction?

A: Yes it does, but we can restrict ourselves to *homogeneous* rewrite rules where both sides have the same type.

Theorem. If all rewrite rules are homogeneous, types are preserved during reduction.

Subject reduction

Q: Doesn't a rewrite rule `true` $\rightharpoonup$ 42 break subject reduction?

A: Yes it does, but we can restrict ourselves to *homogeneous* rewrite rules where both sides have the same type.

~~**Theorem.** If all rewrite rules are homogeneous, types are preserved during reduction.~~

THIS IS FALSE!!

Counterexamples to subject reduction

The rule $(\mathbb{N} \to \mathbb{N}) \rightharpoonup (\mathbb{N} \to \mathbb{B})$ breaks safety:

$$\texttt{zero'} : \mathbb{B}$$
$$\texttt{zero'} = (\lambda x.\, x : \mathbb{N} \to \mathbb{B})\ \texttt{zero}$$

$$\texttt{test} = \texttt{if zero' then } 42 \texttt{ else } 9000$$

The (non-confluent) rules $X \rightharpoonup (\mathbb{N} \to \mathbb{N})$ and $X \rightharpoonup (\mathbb{N} \to \mathbb{B})$ similarly break subject reduction.

Counterexamples to subject reduction

The rule $(\mathbb{N} \to \mathbb{N}) \rightarrowtriangle (\mathbb{N} \to \mathbb{B})$ breaks safety:

$$\texttt{zero'} : \mathbb{B}$$
$$\texttt{zero'} = (\lambda x.\, x : \mathbb{N} \to \mathbb{B})\ \texttt{zero}$$

$$\texttt{test} = \texttt{if zero' then } 42 \texttt{ else } 9000$$

The (non-confluent) rules $X \rightarrowtriangle (\mathbb{N} \to \mathbb{N})$ and $X \rightarrowtriangle (\mathbb{N} \to \mathbb{B})$ similarly break subject reduction.

Regaining subject reduction

To prove subject reduction, we require three properties:

- All rewrite rules are homogeneous
- Rewrite rules do not rewrite type constructors (such as $\rightarrow$)
- Reduction is *confluent*

Again confluence is required!

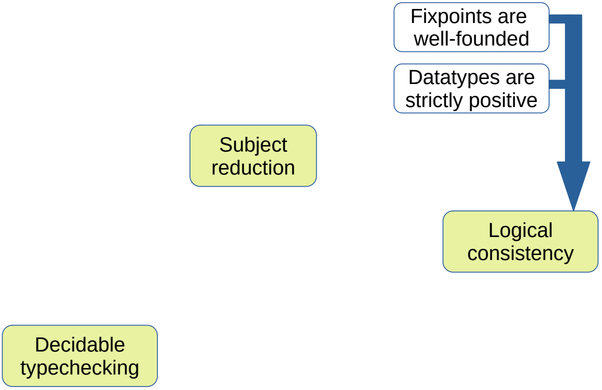
Metatheory of Type Theory

Subject reduction

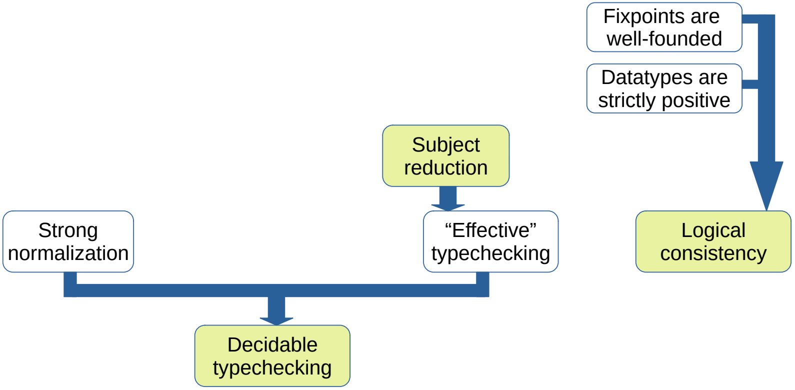
Logical consistency

Decidable typechecking

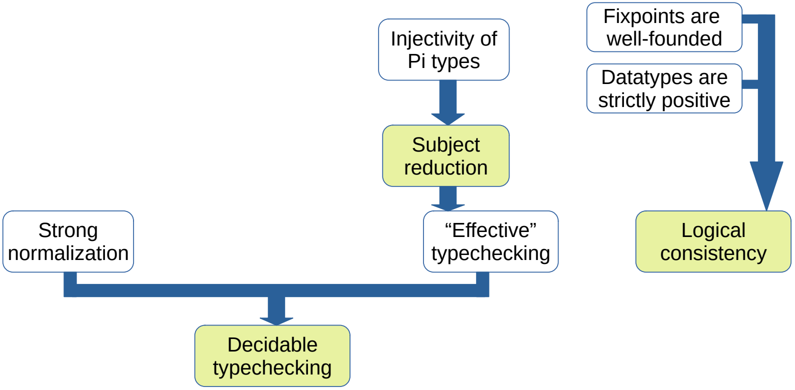
Metatheory of Type Theory



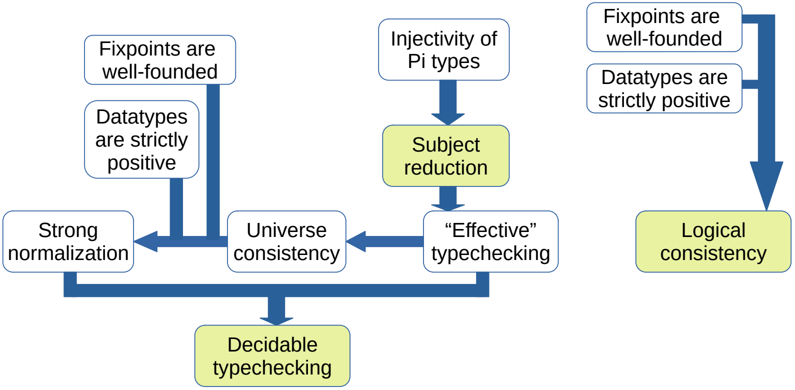
Metatheory of Type Theory



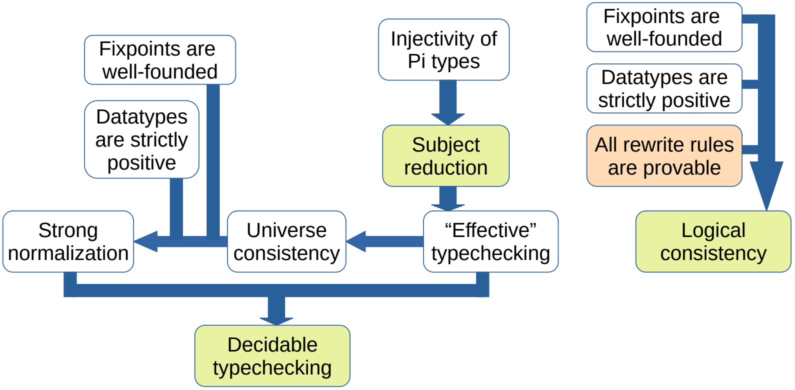
Metatheory of Type Theory



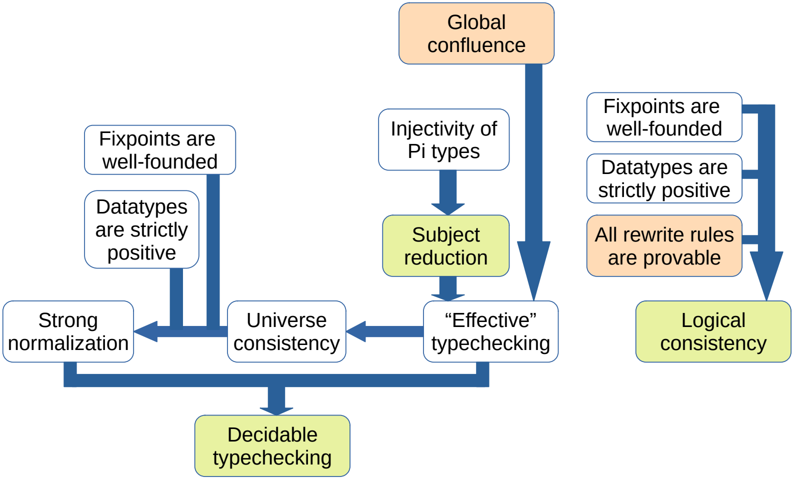
Metatheory of Type Theory



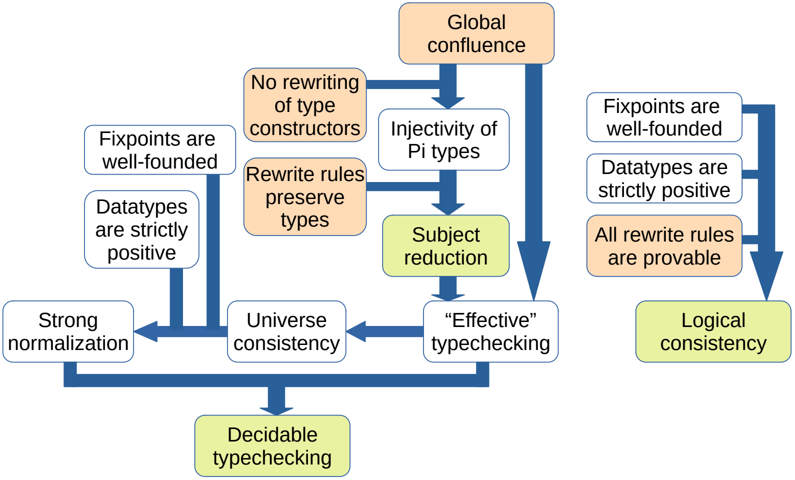
Metatheory of Rewriting Type Theory



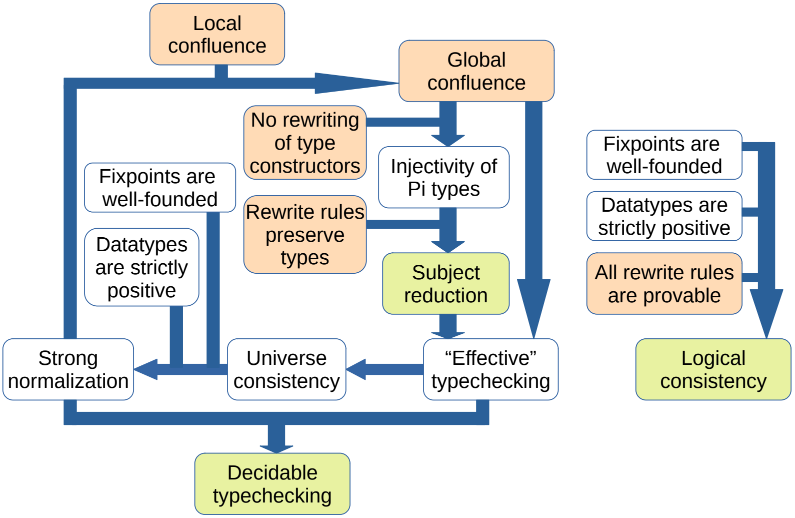
Metatheory of Rewriting Type Theory



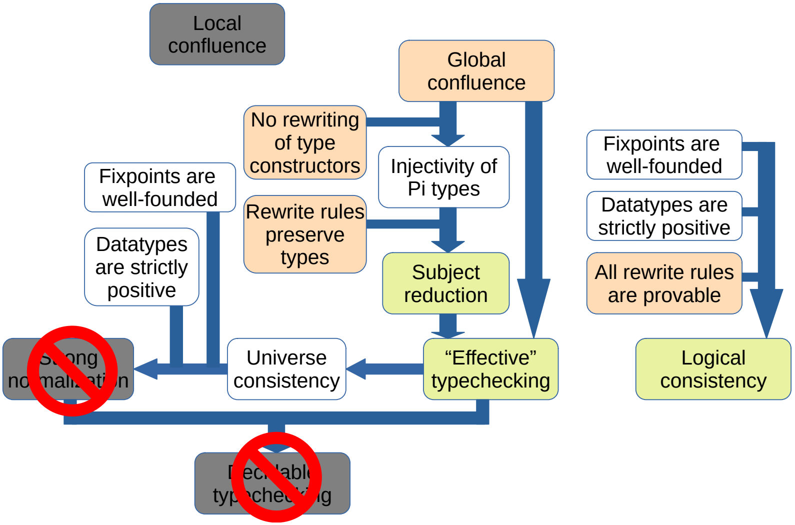
Metatheory of Rewriting Type Theory



Can you spot the problem?



Breaking the loop



Outline

1. Type Theory Unchained
2. Metatheory of RTT
3. Global confluence checking

Wanted: a confluence checker

To restore the metatheory of RTT, we need a confluence check that…

- … can deal with with all features of MLTT
- … accepts the examples we want to support
- … checks *global* confluence without assuming termination
- … is *modular* so we can check files separately and use external libraries without re-checking them

No quick off-the-shelf solution fits all of these…

Some inspiration from the masters

Tait and Martin-Löf gave a classic proof of confluence of untyped lambda calculus that relies on **parallel reduction**.

Parallel reduction ($\Rightarrow$) reduces all immediate redexes by one step:

$$(\texttt{suc}\ a) + ((\lambda x.\ x + b)\ 0) \;\Rightarrow\; \texttt{suc}\ (a + (0 + b))$$

The Tait-Martin-Löf criterion

Triangle property: each term t has an *optimal reduct* $\rho(t)$

$$\begin{tikzcd} t \arrow[d, Rrightarrow] \arrow[dr, Rrightarrow] & \\ u \arrow[r, Rrightarrow] & \rho(t) \end{tikzcd}$$

The triangle property implies global confluence:

$$\begin{tikzcd} & t \arrow[dl, Rrightarrow] \arrow[dr, Rrightarrow] \arrow[dd, Rrightarrow] & \\ u \arrow[dr, Rrightarrow] & & u' \arrow[dl, Rrightarrow] \\ & \rho(t) & \end{tikzcd}$$

Moreover, it can be checked **modularly**!

Checking the triangle property of rewrite rules in three steps

1. Pick an order on the rewrite rules
2. Check that lhs are *closed under unification*: if two lhs l_1 and l_2 have a most general unifier l, then l is the lhs of an *earlier* rewrite rule
3. For every rule $l \rightharpoonup r$ and every parallel step $l \Rrightarrow w$, check that $w \Rrightarrow r$

Checking the triangle property of rewrite rules in three steps

1. Pick an order on the rewrite rules
2. Check that lhs are *closed under unification*: if two lhs l_1 and l_2 have a most general unifier l, then l is the lhs of an *earlier* rewrite rule
3. For every rule $l \rightharpoonup r$ and every parallel step $l \Rrightarrow w$, check that $w \Rrightarrow r$

If step 2 fails we must add auxiliary rules, e.g.

$$(\mathtt{suc}\ x) + (\mathtt{suc}\ y) \rightharpoonup \mathtt{suc}\ (\mathtt{suc}\ (x + y))$$

The triangle criterion in practice

Demo time!

Can we do better?

The triangle criterion is not the most general.

However, its simplicity has some advantages as well:

- We have a formal proof of its correctness
- It is not too hard to implement
- It is predictable to the user
- When it fails, it is usually clear how to fix it

Open question: can we do better?

Conclusion

The tension between propositional and definitional equality is a big barrier to entry for modern proof assistants.

We make definitional equality *extensible* by adding rewrite rules to type theory:

- Improve computation of existing definitions
- Add new primitives that compute

Thanks to the triangle property, we can ensure they preserve type safety in a **modular** way.

All formalized in MetaCoq & implemented in Agda!

Want to learn more?

- Read the papers:
  - TYPES '19: *Type theory unchained* (`https://doi.org/10.4230/LIPIcs.TYPES.2019.2`)
  - POPL '21: *The taming of the rew* (`https://hal.archives-ouvertes.fr/hal-02901011`)
- Play with rewriting in Agda: `https://agda.readthedocs.io/en/v2.6.2/language/rewriting.html`
- Look at the formalization: `https://github.com/TheoWinterhalter/template-coq/`