HOR 2025

Controlling computation in type theory

Théo Winterhalter

joint work with

Jesper Cockx Gaëtan Gilbert Yann Leray Nicolas Tabareau

Main foundations of interactive theorem provers

Higher order logic

Isabelle/HOL

HOL

Main foundations of interactive theorem provers

Higher order logic

Isabelle/HOL

HOL

Dependent type theory

Agda

Lean

Rocq (formerly Coq)

Main foundations of interactive theorem provers

Higher order logic

Isabelle/HOL

HOL

💎 Rewriting 💎

Dependent type theory

Agda

Lean

Rocq (formerly Coq)

Main foundations of interactive theorem provers

Higher order logic

Isabelle/HOL

HOL

💎 Rewriting 💎

Dependent type theory

Agda

Lean

Rocq (formerly Coq)

Computation in type theory

```
refl : 5 + 3 = 10 - 2
```

both sides compute to 8

Computation in type theory

```
refl : 5 + 3 = 10 - 2
```

both sides compute to 8

Types are considered up to computation

```
matrix (0 + x) (2 * y) ≡ matrix x (y + y)
```

Computation in type theory

```
refl : 5 + 3 = 10 - 2
```

both sides compute to 8

Types are considered up to computation

```
matrix (0 + x) (2 * y) ≡ matrix x (y + y)
```

definitional equality

Computation in type theory

```
refl : 5 + 3 = 10 - 2
```

both sides compute to 8

propositional equality
(a data type)

Types are considered up to computation

```
matrix (0 + x) (2 * y) ≡ matrix x (y + y)
```

definitional equality

Computation in type theory

```
refl : 5 + 3 = 10 - 2
```

both sides compute to 8

propositional equality
(a data type)

Types are considered up to computation

```
matrix (0 + x) (2 * y) ≡ matrix x (y + y)
```

definitional equality

```
_|_ : matrix n m → matrix n p → matrix n (m + p)
```

Computation in type theory

```
refl : 5 + 3 ≡ 10 - 2
```

propositional equality
(a data type)

both sides compute to 8

Types are considered up to computation

```
matrix (0 + x) (2 * y) ≡ matrix x (y + y)
```

definitional equality

```
_|_ : matrix n m → matrix n p → matrix n (m + p)
```

Assuming `A : matrix 2 4` `B : matrix 2 3`

```
A | B : matrix 2 7
```

Computation in type theory

```
refl : 5 + 3 = 10 - 2
```

both sides compute to 8

propositional equality
(a data type)

Types are considered up to computation

```
matrix (0 + x) (2 * y) ≡ matrix x (y + y)
```

definitional equality

```
_|_ : matrix n m → matrix n p → matrix n (m + p)
```

$$A = \begin{bmatrix} 0 & 1 & 0 & 2 \\ 1 & 0 & 1 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 5 & 1 & 4 \\ 1 & 1 & 2 \end{bmatrix}$$

Assuming `A : matrix 2 4` `B : matrix 2 3`

```
A | B : matrix 2 7
```

$$A \mid B = \begin{bmatrix} 0 & 1 & 0 & 2 & 5 & 1 & 4 \\ 1 & 0 & 1 & 0 & 1 & 1 & 2 \end{bmatrix}$$

Proofs by computation ✨

Example: equalities in a monoid

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

Proofs by computation ✨

Example: equalities in a monoid

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

How to prove the following?

```
(x · (y · (z · ε))) · (ε · z) = (ε · (x · y)) · (z · z)
```

Proofs by computation ✨

Example: equalities in a monoid

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

How to prove the following? `(x · (y · (z · ε))) · (ε · z) = (ε · (x · y)) · (z · z)`

```
rewrite neu_l
```

Proofs by computation ✨

Example: equalities in a monoid

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

How to prove the following?

```
(x · (y · (z · ε))) · (ε · z) = (ε · (x · y)) · (z · z)
```

`rewrite neu_l`

```
(x · (y · (z · ε))) · z = (ε · (x · y)) · (z · z)
```

Proofs by computation ✨

Example: equalities in a monoid

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

How to prove the following?

```
(x · (y · (z · ε))) · (ε · z) = (ε · (x · y)) · (z · z)
```

`rewrite neu_l`

```
(x · (y · (z · ε))) · z = (ε · (x · y)) · (z · z)
```

`rewrite neu_l, neu_r`

Proofs by computation ✨

Example: equalities in a monoid

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

How to prove the following?

```
(x · (y · (z · ε))) · (ε · z) = (ε · (x · y)) · (z · z)
```

`rewrite neu_l`

```
(x · (y · (z · ε))) · z = (ε · (x · y)) · (z · z)
```

`rewrite neu_l, neu_r`

```
(x · (y · z)) · z = (x · y) · (z · z)
```

Proofs by computation ✨

Example: equalities in a monoid

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

How to prove the following?

```
(x · (y · (z · ε))) · (ε · z) = (ε · (x · y)) · (z · z)
```

`rewrite neu_l`

```
(x · (y · (z · ε))) · z = (ε · (x · y)) · (z · z)
```

`rewrite neu_l, neu_r`

```
(x · (y · z)) · z = (x · y) · (z · z)
```

`rewrite !assoc`

```
((x · y) · z) · z = ((x · y) · z) · z
```

Proofs by computation ✨

Example: equalities in a monoid

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

How to prove the following?	`(x · (y · (z · ε))) · (ε · z) = (ε · (x · y)) · (z · z)`
`rewrite neu_l`	`(x · (y · (z · ε))) · z = (ε · (x · y)) · (z · z)`
`rewrite neu_l, neu_r`	`(x · (y · z)) · z = (x · y) · (z · z)`
`rewrite !assoc`	`((x · y) · z) · z = ((x · y) · z) · z`
`reflexivity`	🎉

Proofs by computation ✨

Example: equalities in a monoid

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

How to prove the following?	`(x · (y · (z · ε))) · (ε · z) = (ε · (x · y)) · (z · z)`
`rewrite neu_l`	`(x · (y · (z · ε))) · z = (ε · (x · y)) · (z · z)`
`rewrite neu_l, neu_r`	`(x · (y · z)) · z = (x · y) · (z · z)`
`rewrite !assoc`	`((x · y) · z) · z = ((x · y) · z) · z`
`reflexivity`	🎉 Yields a proof combining neu_l, neu_r and assoc with congruence of equality

Proofs by computation ✨

Example: equalities in a monoid

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

How to prove the following?	`(x · (y · (z · ε))) · (ε · z) = (ε · (x · y)) · (z · z)`
`rewrite neu_l`	`(x · (y · (z · ε))) · z = (ε · (x · y)) · (z · z)`
`rewrite neu_l, neu_r`	`(x · (y · z)) · z = (x · y) · (z · z)`
`rewrite !assoc`	`((x · y) · z) · z = ((x · y) · z) · z`
`reflexivity`	🎉 Yields a proof combining neu_l, neu_r and assoc with congruence of equality

Tedious
but feels like we can easily
compute a normal form

💡 as a list of atoms

Proofs by computation ✨

Example: equalities in a monoid

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

Proofs by computation ✨

Example: equalities in a monoid

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

```
neu : Mexpr
_*_ : Mexpr → Mexpr → Mexpr
`_ : M → Mexpr
```

Syntax of monoid expressions

Proofs by computation ✨

Example: equalities in a monoid

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

`norm`

normalisation procedure

```
neu : Mexpr
_*_ : Mexpr → Mexpr → Mexpr
`_ : M → Mexpr
```

Syntax of monoid expressions

Proofs by computation ✨

Example: equalities in a monoid

```
eval : Mexpr → M
eval neu := ε
eval (u ⋆ v) := eval u · eval v
eval (` u) := u
```

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

`norm`

normalisation procedure

```
neu : Mexpr
_⋆_ : Mexpr → Mexpr → Mexpr
`_ : M → Mexpr
```

Syntax of monoid expressions

`eval`

Proofs by computation ✨

Example: equalities in a monoid

```
eval : Mexpr → M
eval neu := ε
eval (u ⋆ v) := eval u · eval v
eval (` u) := u
```

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

`norm`

normalisation procedure

```
neu : Mexpr
_⋆_ : Mexpr → Mexpr → Mexpr
`_ : M → Mexpr
```

Syntax of monoid expressions

`eval`

```
lemma norm_sound : ∀ u v. norm u = norm v → eval u = eval v
```

Proofs by computation ✨

Example: equalities in a monoid

```
eval : Mexpr → M
eval neu := ε
eval (u ⋆ v) := eval u · eval v
eval (` u) := u
```

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

norm

normalisation procedure

```
neu : Mexpr
_⋆_ : Mexpr → Mexpr → Mexpr
`_ : M → Mexpr
```

Syntax of monoid expressions

eval

```
lemma norm_sound : ∀ u v. norm u = norm v → eval u = eval v
```

```
(x · (y · (z · ε))) · (ε · z) = (ε · (x · y)) · (z · z)
```

Proofs by computation ✨

Example: equalities in a monoid

norm

normalisation procedure

```
eval : Mexpr → M
eval neu := ε
eval (u ⋆ v) := eval u · eval v
eval (` u) := u
```

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

```
neu : Mexpr
_⋆_ : Mexpr → Mexpr → Mexpr
`_ : M → Mexpr
```

Syntax of monoid expressions

eval

```
lemma norm_sound : ∀ u v. norm u = norm v → eval u = eval v
```

```
(x · (y · (z · ε))) · (ε · z) = (ε · (x · y)) · (z · z)
```

computation

```
eval (`x ⋆ (`y ⋆ (`z ⋆ neu))) ⋆ (neu ⋆ `z) = eval (neu ⋆ (`x ⋆ `y)) ⋆ (`z ⋆ `z))
```

Proofs by computation ✨

Example: equalities in a monoid

norm

normalisation procedure

```
eval : Mexpr → M
eval neu := ε
eval (u ⋆ v) := eval u · eval v
eval (` u) := u
```

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

```
neu : Mexpr
_⋆_ : Mexpr → Mexpr → Mexpr
`_ : M → Mexpr
```

Syntax of monoid expressions

eval

```
lemma norm_sound : ∀ u v. norm u = norm v → eval u = eval v
```

```
(x · (y · (z · ε))) · (ε · z) = (ε · (x · y)) · (z · z)
```

computation

```
eval (`x ⋆ (`y ⋆ (`z ⋆ neu))) ⋆ (neu ⋆ `z) = eval (neu ⋆ (`x ⋆ `y)) ⋆ (`z ⋆ `z))
```

apply norm_sound

```
norm (`x ⋆ (`y ⋆ (`z ⋆ neu))) ⋆ (neu ⋆ `z) = norm (neu ⋆ (`x ⋆ `y)) ⋆ (`z ⋆ `z))
```

Proofs by computation ✨

Example: equalities in a monoid

norm

normalisation procedure

```
eval : Mexpr → M
eval neu := ε
eval (u ⋆ v) := eval u · eval v
eval (` u) := u
```

```
ε : M
_·_ : M → M → M
neu_l : ∀ x. ε · x = x
neu_r : ∀ x. x · ε = x
assoc : ∀ x y z. x · (y · z) = (x · y) · z
```

```
neu : Mexpr
_⋆_ : Mexpr → Mexpr → Mexpr
`_ : M → Mexpr
```

Syntax of monoid expressions

eval

```
lemma norm_sound : ∀ u v. norm u = norm v → eval u = eval v
```

```
(x · (y · (z · ε))) · (ε · z) = (ε · (x · y)) · (z · z)
```

computation

```
eval (`x ⋆ (`y ⋆ (`z ⋆ neu))) ⋆ (neu ⋆ `z) = eval (neu ⋆ (`x ⋆ `y)) ⋆ (`z ⋆ `z))
```

apply norm_sound

```
norm (`x ⋆ (`y ⋆ (`z ⋆ neu))) ⋆ (neu ⋆ `z) = norm (neu ⋆ (`x ⋆ `y)) ⋆ (`z ⋆ `z))
```

computation

```
`x ⋆ (`y ⋆ (`z ⋆ `z)) = `x ⋆ (`y ⋆ (`z ⋆ `z))
```

Proofs by computation ✨

Example: equalities in a monoid (proof comparison)

```
rewrite neu_l
rewrite neu_l, neu_r
rewrite !assoc
reflexivity
```

```
apply norm_sound
reflexivity
```

Proofs by computation ✨

Example: equalities in a monoid (proof comparison)

```
rewrite neu_l
rewrite neu_l, neu_r
rewrite !assoc
reflexivity
```

```
rew (λ X. (x · (y · (z · ε))) · X = (ε · (x · y)) · (z · z)) (neu_l z) (
  rew (λ X. (x · (y · (z · ε))) · z = X · (z · z)) (neu_r (x · y)) (
    rew (λ X. (x · (y · X)) · z = (x · y) · (z · z)) (neu_l z) (
      rew (λ X. X · z = (x · y) · (z · z)) (assoc x y z) (
        rew (λ X. ((x · y) · z) · z = X) (assoc (x · y) z z) refl
      )
    )
  )
)
```

```
apply norm_sound
reflexivity
```

Proofs by computation ✨

Example: equalities in a monoid (proof comparison)

```
rewrite neu_l
rewrite neu_l, neu_r
rewrite !assoc
reflexivity
```

```
rew (λ X. (x · (y · (z · ε))) · X = (ε · (x · y)) · (z · z)) (neu_l z) (
  rew (λ X. (x · (y · (z · ε))) · z = X · (z · z)) (neu_r (x · y)) (
    rew (λ X. (x · (y · X)) · z = (x · y) · (z · z)) (neu_l z) (
      rew (λ X. X · z = (x · y) · (z · z)) (assoc x y z) (
        rew (λ X. ((x · y) · z) · z = X) (assoc (x · y) z z) refl
      )
    )
  )
)
```

doesn't scale 😵‍💫

```
apply norm_sound
reflexivity
```

Proofs by computation ✨

Example: equalities in a monoid (proof comparison)

```
rewrite neu_l
rewrite neu_l, neu_r
rewrite !assoc
reflexivity
```

```
rew (λ X. (x · (y · (z · ε))) · X = (ε · (x · y)) · (z · z)) (neu_l z) (
  rew (λ X. (x · (y · (z · ε))) · z = X · (z · z)) (neu_r (x · y)) (
    rew (λ X. (x · (y · X)) · z = (x · y) · (z · z)) (neu_l z) (
      rew (λ X. X · z = (x · y) · (z · z)) (assoc x y z) (
        rew (λ X. ((x · y) · z) · z = X) (assoc (x · y) z z) refl
      )
    )
  )
)
```

doesn't scale 😵‍💫

```
apply norm_sound
reflexivity
```

```
norm_sound
  (`x * (`y * (`z * neu))) * (neu * `z)
  (neu * (`x * `y)) * (`z * `z))
  refl
```

Proofs by computation ✨

Example: equalities in a monoid (proof comparison)

```
rewrite neu_l
rewrite neu_l, neu_r
rewrite !assoc
reflexivity
```

```
rew (λ X. (x · (y · (z · ε))) · X = (ε · (x · y)) · (z · z)) (neu_l z) (
  rew (λ X. (x · (y · (z · ε))) · z = X · (z · z)) (neu_r (x · y)) (
    rew (λ X. (x · (y · X)) · z = (x · y) · (z · z)) (neu_l z) (
      rew (λ X. X · z = (x · y) · (z · z)) (assoc x y z) (
        rew (λ X. ((x · y) · z) · z = X) (assoc (x · y) z z) refl
      )
    )
  )
)
```

doesn't scale 😵‍💫

```
apply norm_sound
reflexivity
```

```
norm_sound
  (`x ⋆ (`y ⋆ (`z ⋆ neu))) ⋆ (neu ⋆ `z)
  (neu ⋆ (`x ⋆ `y)) ⋆ (`z ⋆ `z))
  refl
```

Proof by computation is much shorter and efficient!

(In such a case you can even show completeness of the approach within the ITP itself)

Proofs by computation ✨

Real-life examples

Proofs by computation ✨

Real-life examples

Using reflection to build efficient and certified decision procedures
Boutin
1997

Proving equalities in a commutative ring done right in Coq
Grégoire, Mahboubi
2005

Interpret ring expressions as multivariate polynomials
much more involved and useful than monoids

Proofs by computation ✨

Real-life examples

Using reflection to build efficient and certified decision procedures
Boutin
1997

Interpret ring expressions as multivariate polynomials
much more involved and useful than monoids

Proving equalities in a commutative ring done right in Coq
Grégoire, Mahboubi
2005

Accelerating verified-compiler development with a verified rewrite engine
Gross, Erbsen, Philipoom, Poddar-Agrawal, Chlipala
2022

Applying the methodology to get a faster
rewrite tactic

Proofs by computation ✨

Real-life examples

Using reflection to build efficient and certified decision procedures
Boutin
1997

Proving equalities in a commutative ring done right in Coq
Grégoire, Mahboubi
2005

Interpret ring expressions as multivariate polynomials
much more involved and useful than monoids

Accelerating verified-compiler development with a verified rewrite engine
Gross, Erbsen, Philipoom, Poddar-Agrawal, Chlipala
2022

Applying the methodology to get a faster
rewrite tactic

Ongoing experiments with Autosubst
a tool to generate boilerplate about substitution for languages with binders

30′ → 6′
rewrite computation

Mathis Bouverot-Dupuis is working on a new Autosubst-like tool based on computation
(jww Kenji Maillard, Kathrin Stark)

Outline

Controlling computation in type theory

Desirable properties and how to preserve them

Go local!

Limits of computation

(by default)

```
lemma comm : ∀ n m. n + m = m + n
```

Limits of computation

(by default)

```
lemma comm : ∀ n m. n + m = m + n
```

```
induction n
```

Limits of computation

(by default)

```
lemma comm : ∀ n m. n + m = m + n
```

```
induction n
```

```
0 + m = m + 0
```

1

Limits of computation

(by default)

```
lemma comm : ∀ n m. n + m = m + n
```

induction n

```
0 + m = m + 0
```

1

```
ih : ∀ m. n + m = m + n
―――――――――――――――――――――――
S n + m = m + S n
```

2

Limits of computation

(by default)

```
lemma comm : ∀ n m. n + m = m + n
```

```
induction n
```

```
0 + m = m + 0
```

1

```
ih : ∀ m. n + m = m + n
─────────────────────────
S n + m = m + S n
```

2

Limits of computation

(by default)

```
lemma comm : ∀ n m. n + m = m + n
```

induction n

```
0 + m = m + 0
```

1

```
ih : ∀ m. n + m = m + n
-----------------------
S n + m = m + S n
```

2

simpl

```
m = m + 0
```

Limits of computation

(by default)

```
lemma comm : ∀ n m. n + m = m + n
```

induction n

```
0 + m = m + 0
```

1

```
ih : ∀ m. n + m = m + n
S n + m = m + S n
```

2

simpl

```
m = m + 0
```

computation is stuck 😔

Limits of computation

(by default)

```
lemma comm : ∀ n m. n + m = m + n
```

```
induction n    0 + m = m + 0                    1
```

```
ih : ∀ m. n + m = m + n
-----------------------
S n + m = m + S n                               2
```

```
simpl          m = m + 0
```

computation is stuck 😔

```
0 + m ≡ m
S n + m ≡ S (n + m)
```

tension between
definitional and propositional
equality

```
n + 0 = n
n + S m = S (n + m)
```

Limits of computation

(by default)

```
lemma comm : ∀ n m. n + m = m + n
```

```
ih : ∀ m. n + m = m + n
───────────────────────
S n + m = m + S n
```

induction n

```
0 + m = m + 0
```

simpl

```
m = m + 0
```

computation is stuck 😔

```
0 + m ≡ m
S n + m ≡ S (n + m)
```

tension between definitional and propositional equality

```
n + 0 = n
n + S m = S (n + m)
```

in vanilla Agda LEAN ROCQ we are forced to choose

Limits of computation

(by default)

```
lemma comm : ∀ n m. n + m = m + n
```

induction n

```
0 + m = m + 0
```

1

```
ih : ∀ m. n + m = m + n
───────────────────────
S n + m = m + S n
```

2

simpl

```
m = m + 0
```

computation is stuck 😔

```
0 + m ≡ m
S n + m ≡ S (n + m)
```

tension between
definitional and propositional
equality

```
n + 0 = n
n + S m = S (n + m)
```

in vanilla
Agda LEAN ROCQ
we are forced to choose

```
n + 0 ≡ n
n + S m ≡ S (n + m)
```

with custom computation, one can get the best of both worlds

Controlling and extending computation in ITPs

(a very partial history, focusing on implementation)

Definitions by rewriting in the calculus of constructions.

Blanqui

2005

Controlling and extending computation in ITPs

(a very partial history, focusing on implementation)

Definitions by rewriting in the calculus of constructions.
Blanqui
2005

Coq modulo theory
Strub
2010

ROCQ
prototypes

Controlling and extending computation in ITPs

(a very partial history, focusing on implementation)

Definitions by rewriting in the calculus of constructions.
Blanqui
2005

Coq modulo theory
Strub
2010

Dedukti: a logical framework based on the λΠ-calculus modulo theory
Assaf, Burel, Cauderlier, Delahaye, Dowek, Dubois, Gilbert, Halmagrand, Hermant, Saillard
2016

ROCQ
prototypes

Controlling and extending computation in ITPs

(a very partial history, focusing on implementation)

Definitions by rewriting in the calculus of constructions.
Blanqui
2005

Coq modulo theory
Strub
2010

ROCQ
prototypes

Dedukti: a logical framework based on the λΠ-calculus modulo theory
Assaf, Burel, Cauderlier, Delahaye, Dowek, Dubois, Gilbert, Halmagrand, Hermant, Saillard
2016

Sprinkles of extensionality for your vanilla type theory
Cockx, Abel
2016

Agda

Controlling and extending computation in ITPs

(a very partial history, focusing on implementation)

Definitions by rewriting in the calculus of constructions.
Blanqui
2005

Coq modulo theory
Strub
2010

prototypes

Dedukti: a logical framework based on the λΠ-calculus modulo theory
Assaf, Burel, Cauderlier, Delahaye, Dowek, Dubois, Gilbert, Halmagrand, Hermant, Saillard
2016

Sprinkles of extensionality for your vanilla type theory
Cockx, Abel
2016

The Rewster: type-preserving rewrite rules for the Coq proof assistant
Leray, Gilbert, Tabareau, Winterhalter
2024

Examples of custom computation rules

Quotients

```
_//_ : ∀ A (R : A → A → Prop). Type
```

Examples of custom computation rules

Quotients

```
_//_ : ∀ A (R : A → A → Prop). Type
proj : A → A // R
```

Examples of custom computation rules

Quotients

```
_//_ : ∀ A (R : A → A → Prop). Type
proj : A → A // R
quot : (x y : A) → R x y → proj x = proj y
```

Examples of custom computation rules

Quotients

```
_//_ : ∀ A (R : A → A → Prop). Type
proj : A → A // R
quot : (x y : A) → R x y → proj x = proj y
rec  : ∀ (f : A → B).
```

Examples of custom computation rules

Quotients

```
_//_ : ∀ A (R : A → A → Prop). Type
proj : A → A // R
quot : (x y : A) → R x y → proj x = proj y
rec  : ∀ (f : A → B).
         (∀ (x y : A). R x y → f x = f y) →
```

Examples of custom computation rules

Quotients

```
_//_ : ∀ A (R : A → A → Prop). Type
proj : A → A // R
quot : (x y : A) → R x y → proj x = proj y
rec  : ∀ (f : A → B).
         (∀ (x y : A). R x y → f x = f y) →
         A // R → B
```

Examples of custom computation rules

Quotients

```
_//_ : ∀ A (R : A → A → Prop). Type
proj : A → A // R
quot : (x y : A) → R x y → proj x = proj y
rec  : ∀ (f : A → B).
         (∀ (x y : A). R x y → f x = f y) →
         A // R → B
```

Examples of custom computation rules

Quotients

```
_//_ : ∀ A (R : A → A → Prop). Type
proj : A → A // R
quot : (x y : A) → R x y → proj x = proj y
rec  : ∀ (f : A → B).
         (∀ (x y : A). R x y → f x = f y) →
         A // R → B

rec f q (proj x) ⇢ f x
```

Examples of custom computation rules

Quotients

```
_//_ : ∀ A (R : A → A → Prop). Type
proj : A → A // R
quot : (x y : A) → R x y → proj x = proj y
rec  : ∀ (f : A → B).
         (∀ (x y : A). R x y → f x = f y) →
         A // R → B

rec f q (proj x) ⇝ f x
```

```
set A := list A // permutation
set_min (s : set ℕ) : ℕ :=
  rec list_min list_min_perm s
```

Examples of custom computation rules

Quotients

```
_//_ : ∀ A (R : A → A → Prop). Type
proj : A → A // R
quot : (x y : A) → R x y → proj x = proj y
rec  : ∀ (f : A → B).
         (∀ (x y : A). R x y → f x = f y) →
         A // R → B

rec f q (proj x) ⇾ f x
```

```
set A := list A // permutation
set_min (s : set ℕ) : ℕ :=
  rec list_min list_min_perm s
```

```
set_min (proj [ 12 ; 10 ; 4 ; 9 ]) ≡ 4
```

Examples of custom computation rules

Quotients

```
_//_ : ∀ A (R : A → A → Prop). Type
proj : A → A // R
quot : (x y : A) → R x y → proj x = proj y
rec  : ∀ (f : A → B).
         (∀ (x y : A). R x y → f x = f y) →
         A // R → B

rec f q (proj x) ⇝ f x
```

```
set A := list A // permutation
set_min (s : set ℕ) : ℕ :=
  rec list_min list_min_perm s
```

```
set_min (proj [ 12 ; 10 ; 4 ; 9 ]) ≡ 4
```

```
set_min (proj (2 :: 1 :: s)) ≡ min 1 (list_min s)
```

better than going through sorted lists

Examples of custom computation rules

Quotients

```
_//_ : ∀ A (R : A → A → Prop). Type
proj : A → A // R
quot : (x y : A) → R x y → proj x = proj y
rec  : ∀ (f : A → B).
         (∀ (x y : A). R x y → f x = f y) →
         A // R → B

rec f q (proj x) ⇝ f x
```

```
set A := list A // permutation
set_min (s : set ℕ) : ℕ :=
  rec list_min list_min_perm s
```

```
set_min (proj [ 12 ; 10 ; 4 ; 9 ]) ≡ 4
```

```
set_min (proj (2 :: 1 :: s)) ≡ min 1 (list_min s)
```

better than going through sorted lists

Exceptions

```
raise : ∀ {A}. A

raise (A := ∀ B. C) ⇝ λ (x : B). raise (A := C)
if raise then t else f ⇝ raise
…
```

Examples of custom computation rules

Quotients

```
_//_ : ∀ A (R : A → A → Prop). Type
proj : A → A // R
quot : (x y : A) → R x y → proj x = proj y
rec  : ∀ (f : A → B).
         (∀ (x y : A). R x y → f x = f y) →
         A // R → B

rec f q (proj x) ↦ f x
```

```
set A := list A // permutation
set_min (s : set ℕ) : ℕ :=
  rec list_min list_min_perm s
```

```
set_min (proj [ 12 ; 10 ; 4 ; 9 ]) ≡ 4
```

```
set_min (proj (2 :: 1 :: s)) ≡ min 1 (list_min s)
```

better than going through sorted lists

Exceptions

```
raise : ∀ {A}. A

raise (A := ∀ B. C) ↦ λ (x : B). raise (A := C)
if raise then t else f ↦ raise
…
```

```
def nth_exn {A} (l : list A) n : A :=
  match l, n with
  | x :: l, 0 ⇒ x
  | x :: l, S n ⇒ nth_exn l n
  | _, _ ⇒ raise
  end
```

no need for the usual error handling with monads (or `option`)

Examples of custom computation rules

Quotients

```
_//_ : ∀ A (R : A → A → Prop). Type
proj : A → A // R
quot : (x y : A) → R x y → proj x = proj y
rec  : ∀ (f : A → B).
         (∀ (x y : A). R x y → f x = f y) →
         A // R → B

rec f q (proj x) ↦ f x
```

```
set A := list A // permutation
set_min (s : set ℕ) : ℕ :=
  rec list_min list_min_perm s
```

```
set_min (proj [ 12 ; 10 ; 4 ; 9 ]) ≡ 4
```

```
set_min (proj (2 :: 1 :: s)) ≡ min 1 (list_min s)
```

better than going through sorted lists

Exceptions

```
raise : ∀ {A}. A

raise (A := ∀ B. C) ↦ λ (x : B). raise (A := C)
if raise then t else f ↦ raise
…
```

```
def nth_exn {A} (l : list A) n : A :=
  match l, n with
  | x :: l, 0 ⇒ x
  | x :: l, S n ⇒ nth_exn l n
  | _, _ ⇒ raise
  end
```

no need for the usual error handling with monads (or `option`)

```
lemma hehe : 0 = 1 :=
  raise
```

☣ the logic is now inconsistent ☣

Equality reflection

Equality reflection

$$\frac{\Gamma \vdash p : u =_A v}{\Gamma \vdash u \equiv v}$$

Maximal extensibility

Equality reflection

Equality reflection

$$\frac{\Gamma \vdash p : u =_A v}{\Gamma \vdash u \equiv v}$$

Undecidable type checking

need to rely on heuristics *eg* SMT solvers in F*

so no longer really computation

Maximal extensibility

Equality reflection

Equality reflection

$$\frac{\Gamma \vdash p : u =_A v}{\Gamma \vdash u \equiv v}$$

Conservative over ITT + UIP + funext

Extensional concepts in intensional type theory
Hofmann
1995

Undecidable type checking
need to rely on heuristics *eg* SMT solvers in F*
so no longer really computation

Equality reflection

Equality reflection

$$\frac{\Gamma \vdash p : u =_A v}{\Gamma \vdash u \equiv v}$$

Undecidable type checking

need to rely on heuristics *eg* SMT solvers in F*

so no longer really computation

Conservative over ITT + UIP + funext

Extensional concepts in intensional type theory

Hofmann

1995

Effective translation to ITT

Extensionality in the calculus of constructions

Oury

2005

Eliminating reflection from type theory

Winterhalter, Sozeau, Tabareau

2019

Terribly inefficient!

Key properties of dependent type theory

(satisfied by Agda and ROCQ)

Consistency

There is no closed proof of $\bot$, ie there is no term `t` such that

$$\vdash \mathtt{t} : \bot$$

Key properties of dependent type theory

(satisfied by Agda and ROCQ)

Consistency

There is no closed proof of $\bot$, ie there is no term t such that

$$\vdash t : \bot$$

prerequisite to be used as a **logic**

Key properties of dependent type theory

(satisfied by Agda and ROCQ)

Consistency

There is no closed proof of $\bot$, ie there is no term `t` such that

$$\vdash t : \bot$$

prerequisite to be used as a **logic**

Decidability of type checking

Given a context Γ, a term `t` and a type `A`,
we can decide whether there is a derivation of

$$\Gamma \vdash t : A$$

Key properties of dependent type theory

(satisfied by Agda and ROCQ)

Consistency

There is no closed proof of $\bot$, ie there is no term t such that

$$\vdash t : \bot$$

prerequisite to be used as a **logic**

Decidability of type checking

Given a context Γ, a term t and a type A,
we can decide whether there is a derivation of

$$\Gamma \vdash t : A$$

key is deciding **conversion** ($\equiv$)

Key properties of dependent type theory

(satisfied by Agda and ROCQ)

Consistency

There is no closed proof of $\bot$, ie there is no term t such that

$$\vdash t : \bot$$

prerequisite to be used as a **logic**

Decidability of type checking

Given a context Γ, a term t and a type A,
we can decide whether there is a derivation of

$$\Gamma \vdash t : A$$

key is deciding **conversion** ($\equiv$)

Subject reduction (or type safety)

Computation may not change the type of an expression

If $\Gamma \vdash u : A$ and $u \to v$ then $\Gamma \vdash v : A$

Key properties of dependent type theory

(satisfied by Agda and ROCQ)

Consistency

There is no closed proof of $\bot$, ie there is no term t such that

$$\vdash t : \bot$$

prerequisite to be used as a **logic**

Decidability of type checking

Given a context Γ, a term t and a type A,
we can decide whether there is a derivation of

$$\Gamma \vdash t : A$$

key is deciding **conversion** ($\equiv$)

Subject reduction (or type safety)

Computation may not change the type of an expression

If $\Gamma \vdash u : A$ and $u \underset{\text{computation relation}}{\underbrace{\to}} v$ then $\Gamma \vdash v : A$

Key properties of dependent type theory

(satisfied by Agda and ROCQ)

Consistency

There is no closed proof of $\bot$, ie there is no term t such that

$$\vdash t : \bot$$

prerequisite to be used as a **logic**

Decidability of type checking

Given a context Γ, a term t and a type A,
we can decide whether there is a derivation of

$$\Gamma \vdash t : A$$

key is deciding **conversion** ($\equiv$)

Subject reduction (or type safety)

Computation may not change the type of an expression

If $\Gamma \vdash u : A$ and $u \to v$ then $\Gamma \vdash v : A$

computation relation

close in spirit to

"Well-typed programs cannot *go wrong*"

– Robin Milner

Deciding conversion

(or definitional equality)

First approximation of conversion:

reflexive, symmetric, transitive closure of computation

Deciding conversion

(or definitional equality)

First approximation of conversion:

reflexive, symmetric, transitive closure of computation

```
(λx. 0 + x) (S 0)
```

Deciding conversion

(or definitional equality)

First approximation of conversion:
reflexive, symmetric, transitive closure of computation

```
(λx. 0 + x) (S 0)  ⟶  0 + S 0
```

Deciding conversion

(or definitional equality)

First approximation of conversion:
reflexive, symmetric, transitive closure of computation

```
(λx. 0 + x) (S 0)  ⟶  0 + S 0  ⟶  S 0
```

Deciding conversion

(or definitional equality)

First approximation of conversion:

reflexive, symmetric, transitive closure of computation

```
                      ↗  (λx. x) (S 0)

(λx. 0 + x) (S 0)  ⟶  0 + S 0  ⟶  S 0
```

Deciding conversion

(or definitional equality)

First approximation of conversion:

reflexive, symmetric, transitive closure of computation

$$\begin{tikzcd}
& \texttt{(λx. x) (S 0)} \arrow[dr] & \\
\texttt{(λx. 0 + x) (S 0)} \arrow[ur] \arrow[r] & \texttt{0 + S 0} \arrow[r] & \texttt{S 0}
\end{tikzcd}$$

Deciding conversion

(or definitional equality)

First approximation of conversion:

reflexive, symmetric, transitive closure of computation

$$\begin{tikzcd}
 & \texttt{(\lambda x. x) (S 0)} \arrow[dr] & & \texttt{S (0 + 0)} \arrow[dl] \\
\texttt{(\lambda x. 0 + x) (S 0)} \arrow[ur] \arrow[r] & \texttt{0 + S 0} \arrow[r] & \texttt{S 0} &
\end{tikzcd}$$

Deciding conversion

(or definitional equality)

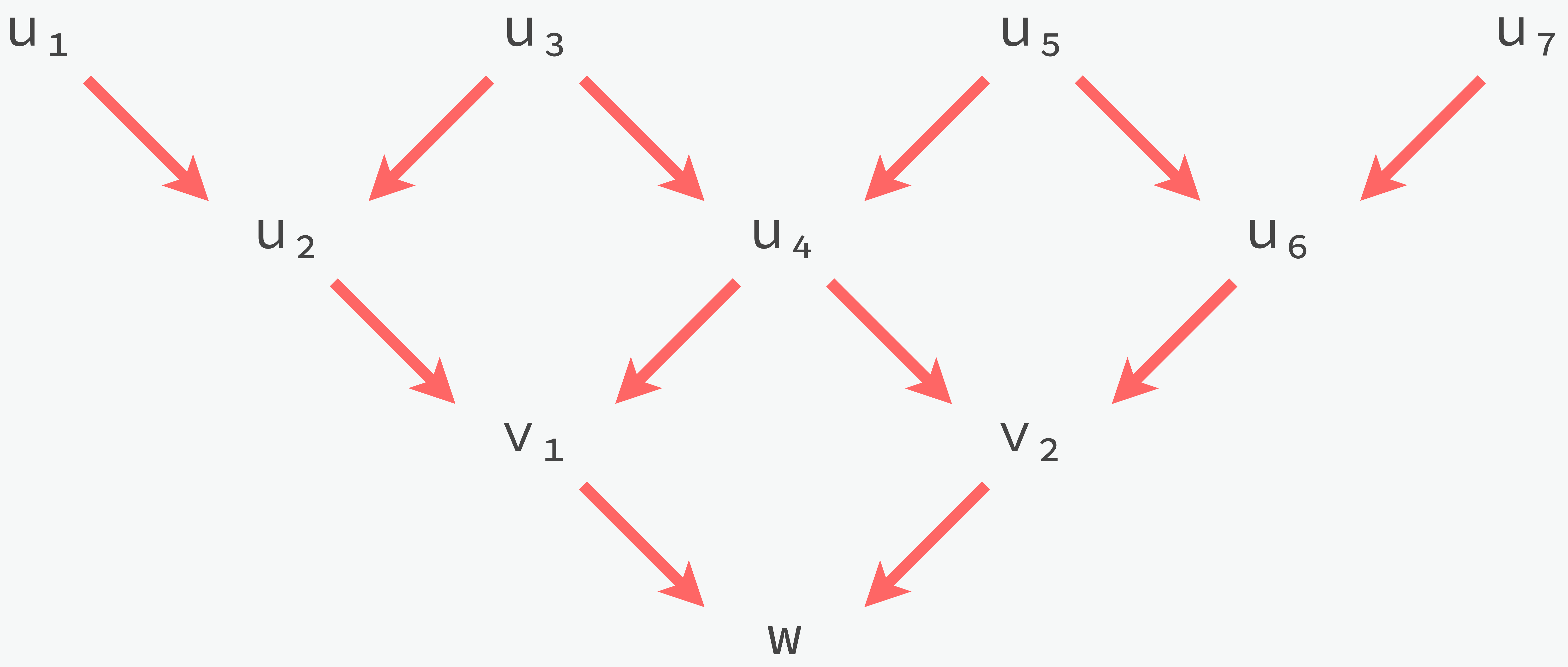
First approximation of conversion:
reflexive, symmetric, transitive closure of computation

$$\begin{tikzcd}
 & \mathtt{(\lambda x.\ x)\ (S\ 0)} \arrow[dr] & & \mathtt{S\ (0\ +\ 0)} \arrow[dl] \\
\mathtt{(\lambda x.\ 0\ +\ x)\ (S\ 0)} \arrow[ur] \arrow[r] & \mathtt{0\ +\ S\ 0} \arrow[r] & \mathtt{S\ 0} &
\end{tikzcd}$$

Two properties we *usually* want: confluence and termination

Confluence

If $u \twoheadrightarrow^* v$ and $u \twoheadrightarrow^* w$ then there exists z, such that $v \twoheadrightarrow^* z$ and $w \twoheadrightarrow^* z$



Only computation is necessary to decide conversion!

Termination

(two notions)

Weak normalisation

If $\Gamma \vdash t : A$ then there exist a finite reduction sequence starting from t and ending in an irreducible term.

Termination

(two notions)

Weak normalisation

If $\Gamma \vdash t : A$ then there exist a finite reduction sequence starting from t and ending in an irreducible term.

Together with confluence: existence and uniqueness of the normal form.

Termination

(two notions)

Weak normalisation

If $\Gamma \vdash t : A$ then there exist a finite reduction sequence starting from t and ending in an irreducible term.

Together with confluence: existence and uniqueness of the normal form.

So we can normalise terms to test their conversion.

Termination

(two notions)

Weak normalisation

If $\Gamma \vdash t : A$ then there exist a finite reduction sequence starting from t and ending in an irreducible term.

Together with confluence: existence and uniqueness of the normal form.

So we can normalise terms to test their conversion.

Strong normalisation

If $\Gamma \vdash t : A$ then all reductions starting from t are finite.
In other words there is not infinite reduction sequence starting from t.

Termination

(two notions)

Weak normalisation

If $\Gamma \vdash t : A$ then there exist a finite reduction sequence starting from `t` and ending in an irreducible term.

Together with confluence: existence and uniqueness of the normal form.

So we can normalise terms to test their conversion.

Strong normalisation

If $\Gamma \vdash t : A$ then all reductions starting from `t` are finite.
In other words there is not infinite reduction sequence starting from `t`.

Hence why it's safe to use whatever reduction strategy: `cbn`, `cbv`, `lazy`…

Termination

(two notions)

Weak normalisation

If $\Gamma \vdash t : A$ then there exist a finite reduction sequence starting from `t` and ending in an irreducible term.

Together with confluence: existence and uniqueness of the normal form.

So we can normalise terms to test their conversion.

Strong normalisation

If $\Gamma \vdash t : A$ then all reductions starting from `t` are finite.
In other words there is not infinite reduction sequence starting from `t`.

Hence why it's safe to use whatever reduction strategy: `cbn`, `cbv`, `lazy`…

So we can be smart when choosing how to reduce terms for conversion.
(Indeed, normalisation can be arbitrarily slow.)

How these properties interact with user-defined computation rules

Consistency

$$0 \longrightarrow 1$$

Consistency

$0 \longrightarrow 1$

Same as `Axiom err : 0 = 1`

Consistency

`0` ⟶ `1`

Same as `Axiom err : 0 = 1`

Theorem

If for each rewrite rule `l ⇾ r` we have a proof ⊢ `e : l = r` then the system remains **consistent**.

No need for termination.

Decidability of checking

`loop` $\longrightarrow$ `loop`

Decidability of checking

$$\texttt{loop} \longrightarrow \texttt{loop}$$

breaks termination, hence decidability of checking

Decidability of checking

`loop` $\longrightarrow$ `loop`

breaks termination, hence decidability of checking

Partial correctness

For **confluent** rewriting systems, the usual algorithm is still correct **if it terminates**!

Type safety

$$\mathbb{N} \to \mathbb{B} \quad \Longrightarrow \quad \mathbb{N} \to \mathbb{N}$$

Type safety

$$\mathbb{N} \to \mathbb{B} \longrightarrow \mathbb{N} \to \mathbb{N}$$

$$\lambda x.x : \mathbb{N} \to \mathbb{N}$$

Type safety

$$\mathbb{N} \to \mathbb{B} \longrightarrow \mathbb{N} \to \mathbb{N}$$

$$\lambda x.x : \mathbb{N} \to \mathbb{N}$$

hence $\lambda x.x : \mathbb{N} \to \mathbb{B}$

Type safety

$$\mathbb{N} \to \mathbb{B} \longrightarrow \mathbb{N} \to \mathbb{N}$$

$$\lambda x.x : \mathbb{N} \to \mathbb{N}$$

hence $\lambda x.x : \mathbb{N} \to \mathbb{B}$

so $(\lambda x.x)\ 0 \to 0 : \mathbb{N}$

Type safety

$$\mathbb{N} \to \mathbb{B} \longrightarrow \mathbb{N} \to \mathbb{N}$$

$$\lambda x.x : \mathbb{N} \to \mathbb{N}$$

hence $\lambda x.x : \mathbb{N} \to \mathbb{B}$

so $(\lambda x.x)\ 0 \to 0 : \mathbb{N}$

type safety is lost

$$\begin{tikzcd}
 & A \arrow[dl] \arrow[dr] & \\
\mathbb{N} \to \mathbb{B} & \equiv & \mathbb{N} \to \mathbb{N}
\end{tikzcd}$$

$$\lambda x.x \ : \ \mathbb{N} \to \mathbb{N}$$

hence $\lambda x.x \ : \ \mathbb{N} \to \mathbb{B}$

so $(\lambda x.x)\ 0 \rightarrowtriangle 0 \ : \ \mathbb{N}$

type safety is lost

even without adding rules on arrow types

$$\begin{tikzcd} & A \arrow[dl] \arrow[dr] & \\ \mathbb{N} \to \mathbb{B} \arrow[dr, dotted] & \equiv & \mathbb{N} \to \mathbb{N} \arrow[dl, dotted] \\ & ? & \end{tikzcd}$$

the key missing property is once again **confluence**

so $(\lambda x.x)\ 0 \to 0 : \mathbb{N}$

type safety is lost

even without adding rules on arrow types

How we establish confluence

Parallel reduction

following Tait, Martin-Löf and Takahashi

$$\rightarrow \;\subset\; \Rrightarrow \;\subset\; \rightarrow^*$$

Can reduce all immediate redexes in one step

How we establish confluence

Parallel reduction

following Tait, Martin-Löf and Takahashi

$$\rightarrow \subset \Rrightarrow \subset \rightarrow^*$$

Can reduce all immediate redexes in one step

$$(S\ a) + ((\lambda x.\ x + b)\ 0) \Rrightarrow S\ (a + (0 + b))$$

How we establish confluence

Parallel reduction

following Tait, Martin-Löf and Takahashi

$$\rightarrow \subset \Rrightarrow \subset \rightarrow^*$$

Can reduce all immediate redexes in one step

$$\texttt{(S a) + ((λx. x + b) 0)} \Rrightarrow \texttt{S (a + (0 + b))}$$

but also

$$\texttt{(S a) + ((λx. x + b) 0)} \Rrightarrow \texttt{(S a) + (0 + b)}$$

How we establish confluence

Parallel reduction

following Tait, Martin-Löf and Takahashi

$$\rightarrow \subset \Rrightarrow \subset \rightarrow^*$$

$\rightarrow$ confluent iff $\Rrightarrow$ confluent

Can reduce all immediate redexes in one step

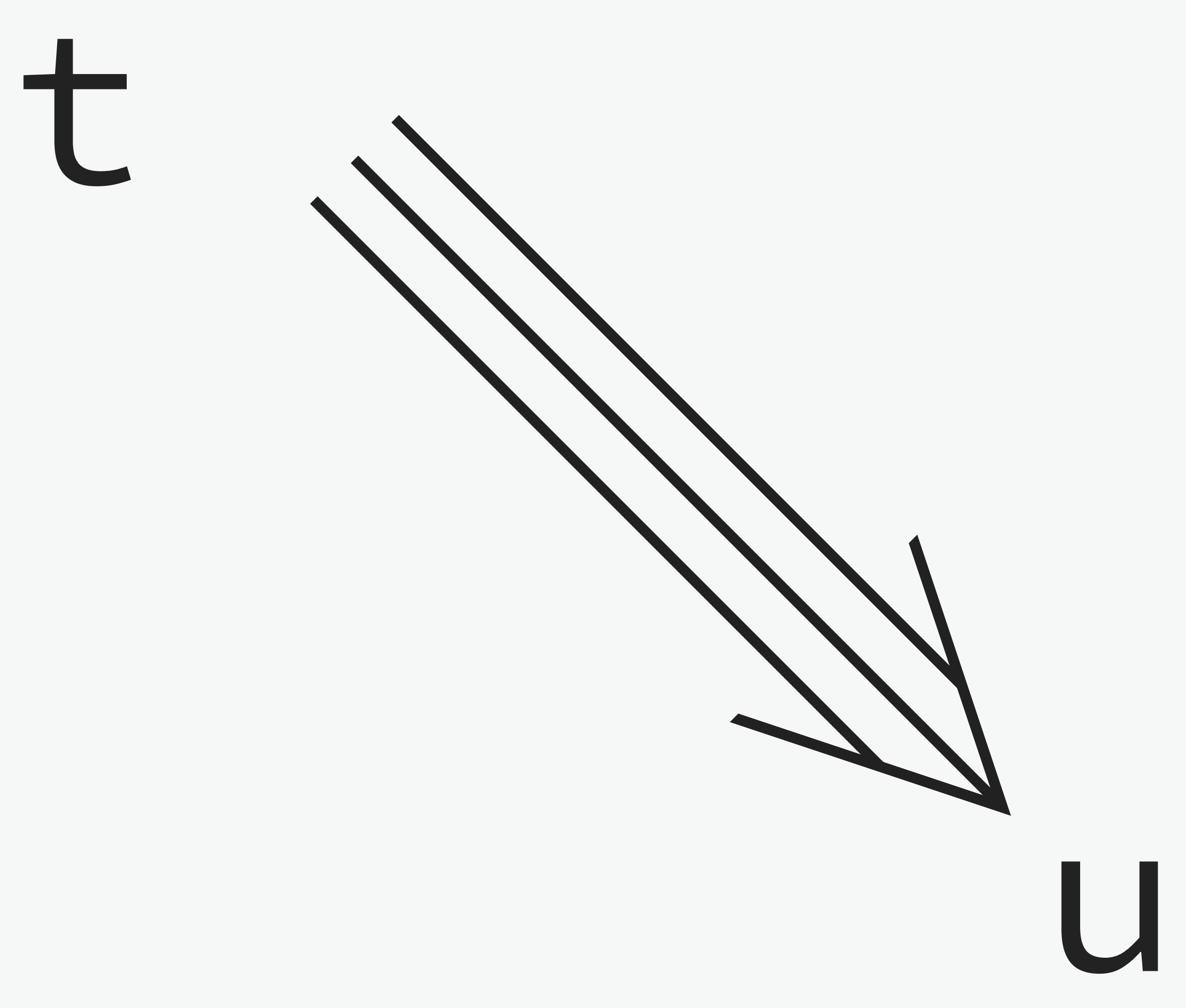
`(S a) + ((λx. x + b) 0) ⇛ S (a + (0 + b))`

but also

`(S a) + ((λx. x + b) 0) ⇛ (S a) + (0 + b)`

How we establish confluence

Triangle property



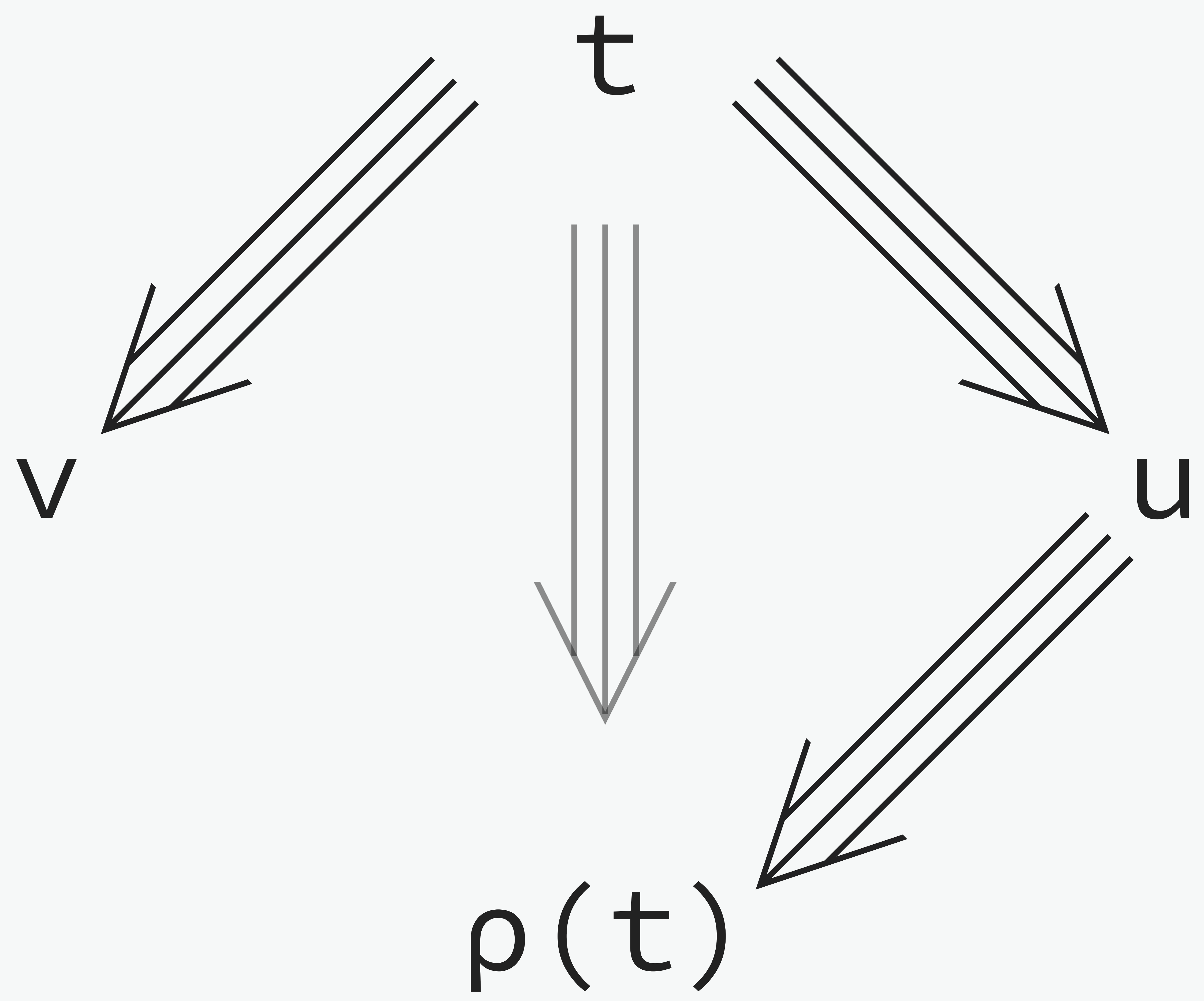
How we establish confluence

Triangle property

$$
\begin{tikzcd}
t \arrow[dr, Rrightarrow] \arrow[dd, Rrightarrow, gray] & \\
& u \arrow[dl, Rrightarrow] \\
\rho(t) &
\end{tikzcd}
$$

How we establish confluence

Triangle property



How we establish confluence

Triangle property

$$
\begin{tikzcd}
 & t \arrow[dl, Rrightarrow] \arrow[dr, Rrightarrow] \arrow[dd, Rrightarrow] & \\
v \arrow[dr, Rrightarrow] & & u \arrow[dl, Rrightarrow] \\
 & \rho(t) &
\end{tikzcd}
$$

How we establish confluence

Triangle property

$$
\begin{tikzcd}
 & t \arrow[dl, Rrightarrow] \arrow[dr, Rrightarrow] \arrow[dd, Rrightarrow] & \\
v \arrow[dr, Rrightarrow] & & u \arrow[dl, Rrightarrow] \\
 & \rho(t) &
\end{tikzcd}
$$

Parallel reduction enjoys the diamond property hence **confluence**

Triangle criterion

the idea

$$\frac{l \rightarrow r \qquad\qquad \sigma \Rrightarrow \sigma'}{l\sigma \Rrightarrow r\sigma'}$$

Triangle criterion

the idea

$$\frac{l \rightarrow r \qquad\qquad \sigma \Rrightarrow \sigma'}{l\sigma \Rrightarrow r\sigma'}$$

```
ρ(lσ) := r(map ρ σ)
```

Triangle criterion

parallel plus example

```
ρ(lσ) := r(map ρ σ)
```

```
0     +    ?y → ?y
?x    +     0 → ?x
S ?x  +    ?y → S (?x + ?y)
?x    + S ?y → S (?x + ?y)
```

Triangle criterion

parallel plus example

$$\rho(\mathrm{S}\ u + \mathrm{S}\ v) := ??$$

```
0     +    ?y → ?y
?x    +     0 → ?x
S ?x  +    ?y → S (?x + ?y)
?x    + S ?y → S (?x + ?y)
```

Triangle criterion

parallel plus example

$$\rho(S\ u + S\ v) := ??$$

```
S ?x + S ?y ⇛ S (S (?x + ?y))
```

```
0     +    ?y → ?y
?x    +     0 → ?x
S ?x  +    ?y → S (?x + ?y)
?x    + S ?y → S (?x + ?y)
```

Triangle criterion

parallel plus example

$$\rho(S\ u + S\ v) := S\ (S\ (\rho(u) + \rho(v)))$$

```
S ?x + S ?y ⇛ S (S (?x + ?y))
```

```
0    +   ?y ⇾ ?y
?x   +    0 ⇾ ?x
S ?x +   ?y ⇾ S (?x + ?y)
?x   + S ?y ⇾ S (?x + ?y)
```

Triangle criterion

parallel plus example

$$\rho(S\ u + S\ v) := S\ (S\ (\rho(u) + \rho(v)))$$

1. `S ?x + S ?y ⇛ S (S (?x + ?y))`

```
2. 0      +    ?y ⇾ ?y
3. ?x     +     0 ⇾ ?x
4. S ?x +      ?y ⇾ S (?x + ?y)
5. ?x     + S ?y ⇾ S (?x + ?y)
```

Triangle criterion

the idea

Triangle criterion

the idea

Order on the rules (including extra parallel rules) to define ρ

Triangle criterion

the idea

Order on the rules (including extra parallel rules) to define ρ

Every time `lσ` is an instance `l'θ` of an earlier rule, we ask:

$$r\sigma \Rrightarrow r'\theta$$

for σ, θ **pattern** substitutions and `l' → r' < l → r`

Triangle criterion

the idea

Order on the rules (including extra parallel rules) to define ρ

Every time `lσ` is an instance `l'θ` of an earlier rule, we ask:

$$r\sigma \Rrightarrow r'\theta$$

for σ, θ **pattern** substitutions and `l' → r' < l → r`

Local criterion, checked **modularly**

Triangle criterion

the idea

Order on the rules (including extra parallel rules) to define ρ

Every time `lσ` is an instance `l’θ` of an earlier rule, we ask:

$$r\sigma \Rrightarrow r'\theta$$

for σ, θ **pattern** substitutions and `l' → r' < l → r`

Local criterion, checked modularly

Theorem | Confluence holds assuming the above.

Taming user-defined computation

Modular confluence criterion

The Taming of the Rew: A type theory with computational assumptions

Cockx, Tabareau, Winterhalter

2021

Taming user-defined computation

Modular confluence criterion

The Taming of the Rew: A type theory with computational assumptions
Cockx, Tabareau, Winterhalter
2021

Modular type safety conditions

Type safety of rewrite rules in dependent types
Blanqui
2020

The Rewster: type-preserving rewrite rules for the Coq proof assistant
Leray, Gilbert, Tabareau, Winterhalter
2024

Taming user-defined computation

Modular confluence criterion

The Taming of the Rew: A type theory with computational assumptions
Cockx, Tabareau, Winterhalter
2021

Modular type safety conditions

Type safety of rewrite rules in dependent types
Blanqui
2020

The Rewster: type-preserving rewrite rules for the Coq proof assistant
Leray, Gilbert, Tabareau, Winterhalter
2024

Termination is hard…
Partial correctness is fine!

Taming user-defined computation

Modular confluence criterion

The Taming of the Rew: A type theory with computational assumptions
Cockx, Tabareau, Winterhalter
2021

Modular type safety conditions

Type safety of rewrite rules in dependent types
Blanqui
2020

The Rewster: type-preserving rewrite rules for the Coq proof assistant
Leray, Gilbert, Tabareau, Winterhalter
2024

Termination is hard…
Partial correctness is fine!

Consistency
left to the user
like axioms

Taming user-defined computation

Modular confluence criterion

The Taming of the Rew: A type theory with computational assumptions
Cockx, Tabareau, Winterhalter
2021

Modular type safety conditions

Type safety of rewrite rules in dependent types
Blanqui
2020

The Rewster: type-preserving rewrite rules for the Coq proof assistant
Leray, Gilbert, Tabareau, Winterhalter
2024

Termination is hard…
Partial correctness is fine!

Consistency
left to the user
like axioms

+ many tools developed by the community!

Go local!

Why locality matters

Example: exceptions

```
symb raise : ∀ {A}. A
rule if raise then t else f ↦ raise
def nth_exn : list A → ℕ → A := …
```

Agda Rocq

Computation rules must be assumed forever

Why locality matters

Example: exceptions

```
symb raise : ∀ {A}. A
rule if raise then t else f ↪ raise
def nth_exn : list A → ℕ → A := …
```

```
lemma something_unrelated : …
```

Agda · Lean · ROCQ

Computation rules must be assumed **forever**

No way to ensure the rules aren't used here

(unlike axioms, there is no `Print Assumptions`)

Why locality matters

Example: exceptions

```
symb raise : ∀ {A}. A
rule if raise then t else f ↪ raise
def nth_exn : list A → ℕ → A := …
```

```
lemma something_unrelated : …
```

Agda ROCQ

Computation rules must be assumed forever

No way to ensure the rules aren't used here
(unlike axioms, there is no `Print Assumptions`)

Example: booleans

```
symb bool : Type
symb true, false : bool
symb ifte : bool → A → A → A
rule ifte true t f ↪ t
rule ifte false t f ↪ t
```

Rules extend the trusted computing base

Why locality matters

Example: exceptions

```
symb raise : ∀ {A}. A
rule if raise then t else f ↪ raise
def nth_exn : list A → ℕ → A := …
```

```
lemma something_unrelated : …
```

Agda · ROCQ

Computation rules must be assumed forever

No way to ensure the rules aren't used here
(unlike axioms, there is no `Print Assumptions`)

Example: booleans

```
symb bool : Type
symb true, false : bool
symb ifte : bool → A → A → A
rule ifte true t f ↪ t
rule ifte false t f ↪ t
```

Rules extend the trusted computing base

Uncaught mistake without a model
(somewhat mitigated in Agda)

Why locality matters

Example: exceptions

```
symb raise : ∀ {A}. A
rule if raise then t else f ↪ raise
def nth_exn : list A → ℕ → A := …
```

```
lemma something_unrelated : …
```

Agda · Rocq

Computation rules must be assumed **forever**

No way to ensure the rules aren't used here
(unlike axioms, there is no `Print Assumptions`)

Example: booleans

```
symb bool : Type
symb true, false : bool
symb ifte : bool → A → A → A
rule ifte true t f ↪ t
rule ifte false t f ↪ t
```

Rules extend the **trusted computing base**

Uncaught mistake without a model
(somewhat mitigated in Agda)

Overall, not very **modular**
(or type-theoretic)

Our proposal

Prenex quantification over computation rules

```
interface Bool
  assumes
    bool : Type
    true, false : bool
    ifte : ∀ (P : bool → Type). P true → P false → ∀ b. P b
  where
    ifte P t f true ⇾ t
    ifte P t f false ⇾ f
```

Our proposal

Prenex quantification over computation rules

```
interface Bool
  assumes
    bool : Type
    true, false : bool
    ifte : ∀ (P : bool → Type). P true → P false → ∀ b. P b
  where
    ifte P t f true ⇾ t
    ifte P t f false ⇾ f
```

```
def negb⟨ B : Bool ⟩ (b : B.bool) : B.bool :=
  B.ifte (λ _. B.bool) B.false B.true b
```

Our proposal

Prenex quantification over computation rules

```
interface Bool
  assumes
    bool : Type
    true, false : bool
    ifte : ∀ (P : bool → Type). P true → P false → ∀ b. P b
  where
    ifte P t f true ⇢ t
    ifte P t f false ⇢ f
```

```
def negb⟨ B : Bool ⟩ (b : B.bool) : B.bool :=
  B.ifte (λ _. B.bool) B.false B.true b
```

```
interface Sum
  assumes
    sum : Type → Type → Type
    inl : ∀ {A B}. A → sum A B
    inr : ∀ {A B}. B → sum A B
    elim :
      ∀ {A B} (P : sum A B → Type).
        (∀ a, P (inl a)) →
        (∀ b, P (inr b)) →
        ∀ s. P s
  where
    elim P l r (inl a) ⇢  l a
    elim P l r (inr b) ⇢  r b
```

Our proposal

Prenex quantification over computation rules

```
interface Bool
  assumes
    bool : Type
    true, false : bool
    ifte : ∀ (P : bool → Type). P true → P false → ∀ b. P b
  where
    ifte P t f true ⇢ t
    ifte P t f false ⇢ f
```

```
def negb⟨ B : Bool ⟩ (b : B.bool) : B.bool :=
  B.ifte (λ _. B.bool) B.false B.true b
```

```
interface Sum
  assumes
    sum : Type → Type → Type
    inl : ∀ {A B}. A → sum A B
    inr : ∀ {A B}. B → sum A B
    elim :
      ∀ {A B} (P : sum A B → Type).
        (∀ a, P (inl a)) →
        (∀ b, P (inr b)) →
        ∀ s. P s
  where
    elim P l r (inl a) ⇢  l a
    elim P l r (inr b) ⇢  r b
```

```
instance Bool-as-sum⟨ U : Unit, S : Sum ⟩ : Bool
  bool := S.sum U.unit U.unit
  true := S.inl U.⋆
  false := S.inr U.⋆
  ifte P t f := S.elim P (U.elim _ t) (U.elim _ f)
```

Equations are verified implicitly

Our proposal

Prenex quantification over computation rules

```
interface Bool
  assumes
    bool : Type
    true, false : bool
    ifte : ∀ (P : bool → Type). P true → P false → ∀ b. P b
  where
    ifte P t f true ⇢ t
    ifte P t f false ⇢ f
```

```
def negb⟨ B : Bool ⟩ (b : B.bool) : B.bool :=
  B.ifte (λ _. B.bool) B.false B.true b
```

```
interface Sum
  assumes
    sum : Type → Type → Type
    inl : ∀ {A B}. A → sum A B
    inr : ∀ {A B}. B → sum A B
    elim :
      ∀ {A B} (P : sum A B → Type).
        (∀ a, P (inl a)) →
        (∀ b, P (inr b)) →
        ∀ s. P s
  where
    elim P l r (inl a) ⇢  l a
    elim P l r (inr b) ⇢  r b
```

```
def foo⟨ U : Unit, S : Sum ⟩ :=
  negb⟨ Bool-as-sum⟨ U, S ⟩ ⟩
```

```
instance Bool-as-sum⟨ U : Unit, S : Sum ⟩ : Bool
  bool := S.sum U.unit U.unit
  true := S.inl U.⋆
  false := S.inr U.⋆
  ifte P t f := S.elim P (U.elim _ t) (U.elim _ f)
```

Equations are verified implicitly

Our proposal

Prenex quantification over computation rules

```
interface Bool
  assumes
    bool : Type
    true, false : bool
    ifte : ∀ (P : bool → Type). P true → P false → ∀ b. P b
  where
    ifte P t f true ⇝ t
    ifte P t f false ⇝ f
```

```
def negb⟨ B : Bool ⟩ (b : B.bool) : B.bool :=
  B.ifte (λ _. B.bool) B.false B.true b
```

```
interface Sum
  assumes
    sum : Type → Type → Type
    inl : ∀ {A B}. A → sum A B
    inr : ∀ {A B}. B → sum A B
    elim :
      ∀ {A B} (P : sum A B → Type).
        (∀ a, P (inl a)) →
        (∀ b, P (inr b)) →
        ∀ s. P s
  where
    elim P l r (inl a) ⇝  l a
    elim P l r (inr b) ⇝  r b
```

```
def foo⟨ U : Unit, S : Sum ⟩ :=
  negb⟨ Bool-as-sum⟨ U, S ⟩ ⟩
```

```
instance Bool-as-sum⟨ U : Unit, S : Sum ⟩ : Bool
  bool := S.sum U.unit U.unit
  true := S.inl U.⋆
  false := S.inr U.⋆
  ifte P t f := S.elim P (U.elim _ t) (U.elim _ f)
```

Equations are verified implicitly

You can exploit the encoding without having to work directly with it!

Example: Strictifying substitutions

and renamings

Autosubst encodes substitutions as functions $\mathbb{N} \to$ `term` and renamings as $\mathbb{N} \to \mathbb{N}$
to get definitional unitality and associativity of composition

Example: Strictifying substitutions

and renamings

Autosubst encodes substitutions as functions $\mathbb{N} \to$ `term` and renamings as $\mathbb{N} \to \mathbb{N}$
to get definitional unitality and associativity of composition

induces issues with meta-programming
(confusion with regular function composition)

Example: Strictifying substitutions

and renamings

Autosubst encodes substitutions as functions `ℕ → term` and renamings as `ℕ → ℕ`
to get definitional unitality and associativity of composition

induces issues with meta-programming
(confusion with regular function composition)

```
interface Renaming
  assumes
    renaming : Type
    comp : renaming → renaming → renaming
    id : renaming
  where
    comp f id ⇢ f
    comp id f ⇢ f
    comp (comp f g) h ⇢ comp f (comp g h)
```

Example: Strictifying substitutions

and renamings

Autosubst encodes substitutions as functions `ℕ → term` and renamings as `ℕ → ℕ`
to get definitional unitality and associativity of composition

induces issues with meta-programming
(confusion with regular function composition)

```
interface Renaming
  assumes
    renaming : Type
    comp : renaming → renaming → renaming
    id : renaming
  where
    comp f id ⇝ f
    comp id f ⇝ f
    comp (comp f g) h ⇝ comp f (comp g h)
```

```
instance RenamingFun
    renaming := ℕ → ℕ
    comp := λ f g x. f (g x)
    id := λ x. x
```

Example: Strictifying substitutions

and renamings

Autosubst encodes substitutions as functions `ℕ → term` and renamings as `ℕ → ℕ`
to get definitional unitality and associativity of composition

induces issues with meta-programming
(confusion with regular function composition)

```
interface Renaming
  assumes
    renaming : Type
    comp : renaming → renaming → renaming
    id : renaming
  where
    comp f id ⇝ f
    comp id f ⇝ f
    comp (comp f g) h ⇝ comp f (comp g h)
```

```
instance RenamingFun
    renaming := ℕ → ℕ
    comp := λ f g x. f (g x)
    id := λ x. x
```

and so on...

Other examples include…

Hiding implementation details
while retaining computation

```
interface Shift
  assumes
    shift : list ℕ → list ℕ
  where
    shift (x :: l) ⇒ S x :: shift l
    shift [] ⇒ []
```

```
instance ShiftasMap
  shift := map S
```

This way, `map` never appears in goals out of nowhere
useful for automatically generated functions (eg. Equations in Rocq)

Other examples include...

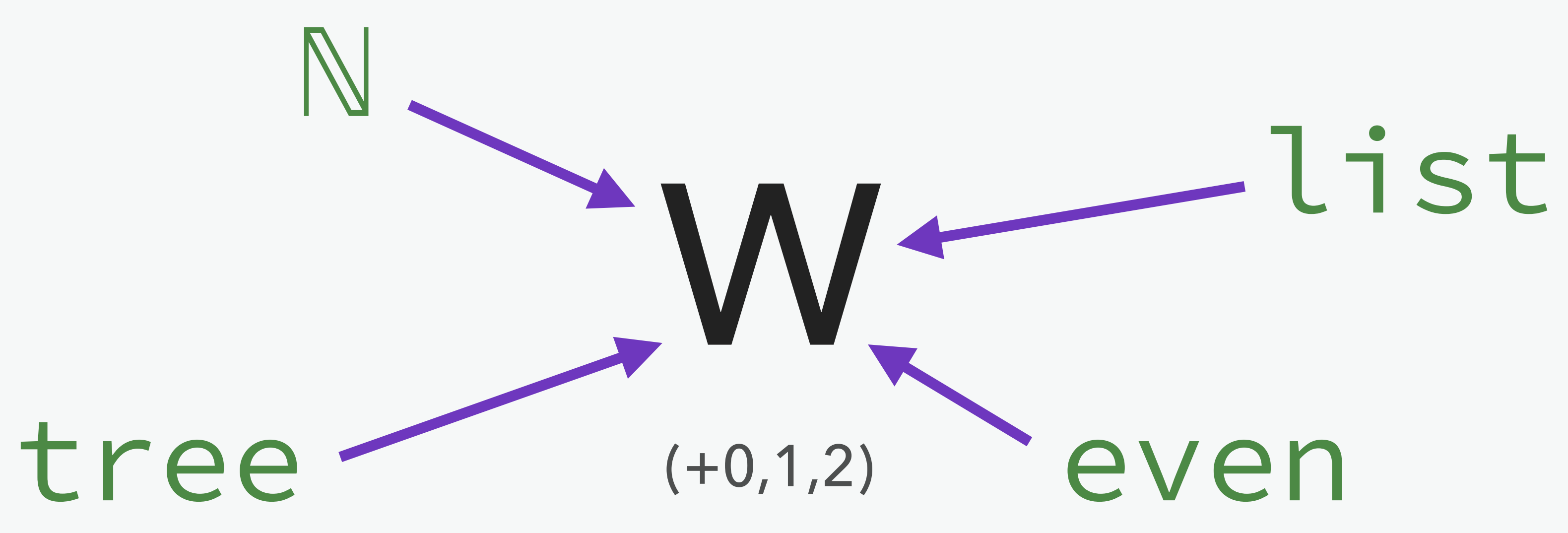
Hiding implementation details
while retaining computation

```
interface Shift
  assumes
    shift : list ℕ → list ℕ
  where
    shift (x :: l) ⇾ S x :: shift l
    shift [] ⇾ []
```

```
instance ShiftasMap
  shift := map S
```

This way, `map` never appears in goals out of nowhere
useful for automatically generated functions (eg. Equations in Rocq)

Encode features using simpler ones



Why not W?

Hugunin

2021

Other examples include…

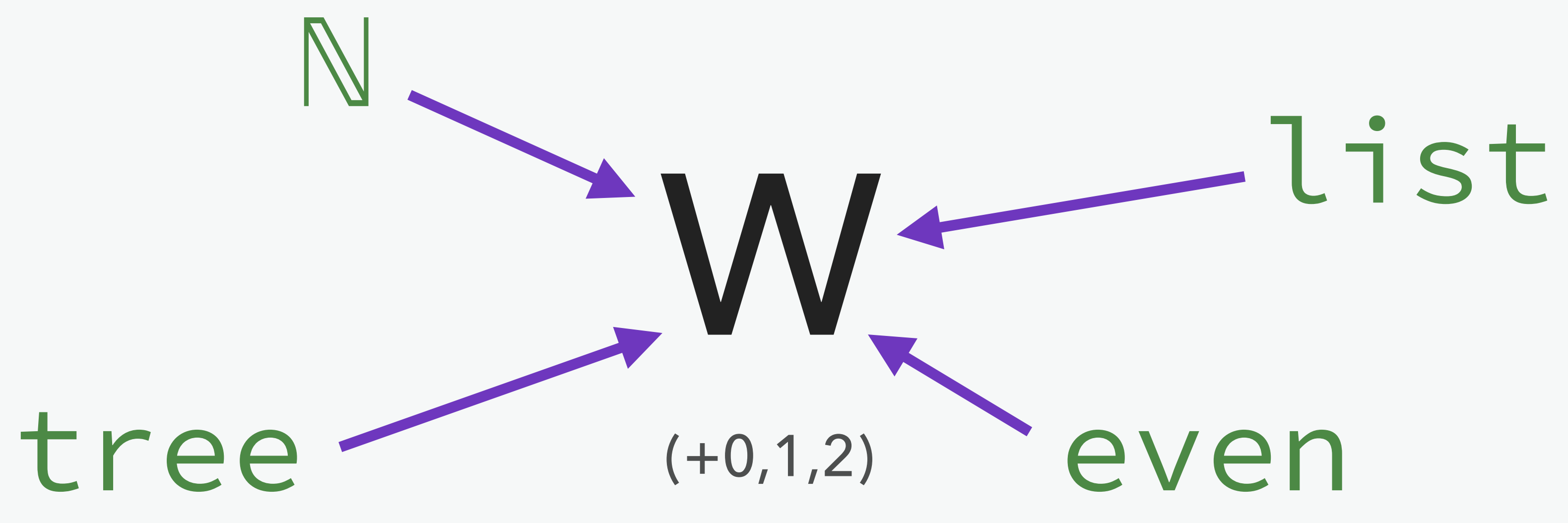
Hiding implementation details
while retaining computation

```
interface Shift
  assumes
    shift : list ℕ → list ℕ
  where
    shift (x :: l) ⇢ S x :: shift l
    shift [] ⇢ []
```

```
instance ShiftasMap
  shift := map S
```

This way, `map` never appears in goals out of nowhere
useful for automatically generated functions (eg. Equations in Rocq)

Encode features using simpler ones



Why not W?

Hugunin

2021

Use effects locally, eg. exceptions

The type theory: LRTT

$$\Sigma \mid \Xi \mid \Gamma \vdash t : A$$

The type theory: LRTT

```
symb x : A
```

```
rule l ↪ r
```

Interface environment

$$\Sigma \mid \Xi \mid \Gamma \vdash t : A$$

The type theory: LRTT

```
symb x : A
```

```
def f⟨ Ξ’ ⟩ : A := t
```

```
rule l → r
```

Global environment

Interface environment

$$\Sigma \mid \Xi \mid \Gamma \vdash t : A$$

The type theory: LRTT

```
symb x : A
```

```
def f⟨ Ξ’ ⟩ : A := t
```

```
rule l → r
```

Global environment

Interface environment

Basically regular MLTT

$$\Sigma \mid \Xi \mid \boxed{\Gamma \vdash t : A}$$

The type theory: LRTT

`symb x : A`

`def f⟨ Ξ’ ⟩ : A := t`

`rule l ⇢ r`

Global environment

Interface environment

Basically regular MLTT

$$\Sigma \mid \Xi \mid \boxed{\Gamma \vdash t : A}$$

Computation rule

$$\frac{(l \rightarrow r) \in \Xi}{\Sigma \mid \Xi \mid \Gamma \vdash l[\sigma] \equiv r[\sigma]}$$

The type theory: LRTT

```
symb x : A
```

```
def f⟨ Ξ’ ⟩ : A := t
```

```
rule l → r
```

Global environment

Interface environment

Basically regular MLTT

$$\Sigma \mid \Xi \mid \boxed{\Gamma \vdash t : A}$$

Computation rule

$$\frac{(l \to r) \in \Xi}{\Sigma \mid \Xi \mid \Gamma \vdash l[\sigma] \equiv r[\sigma]}$$

Unfolding rule

$$\frac{(\mathtt{def}\ f\langle \Xi' \rangle : A := t) \in \Sigma \qquad \Sigma \mid \Xi \mid \Gamma \vdash \xi : \Xi'}{\Sigma \mid \Xi \mid \Gamma \vdash f\langle \xi \rangle \equiv t\xi}$$

Meta-theory

mostly usual

Environment weakening (Σ, Ξ, Γ), substitution, instantiation, validity

Meta-theory

mostly usual

Environment weakening (Σ, Ξ, Γ), **substitution, instantiation, validity**

```
Σ | Ξ | Γ ⊢ ξ : Ξ' →
Σ | Ξ' | · ⊢ t : A →
Σ | Ξ | Γ ⊢ tξ : Aξ
```

Meta-theory

mostly usual

Environment weakening (Σ, Ξ, Γ), **substitution, instantiation, validity**

```
Σ | Ξ | Γ ⊢ ξ : Ξ' →
Σ | Ξ' | · ⊢ t : A →
Σ | Ξ | Γ ⊢ tξ : Aξ
```

Consistency

A given by embedding into a theory
with equality reflection

Meta-theory

mostly usual

Environment weakening (Σ, Ξ, Γ), **substitution, instantiation, validity**

$\Sigma \mid \Xi \mid \Gamma \vdash \xi : \Xi' \to$
$\Sigma \mid \Xi' \mid \cdot \vdash t : A \to$
$\Sigma \mid \Xi \mid \Gamma \vdash t\xi : A\xi$

Consistency

A given by embedding into a theory
with equality reflection

more interesting:

Conservativity over MLTT

$\cdot \mid \cdot \mid \cdot \vdash A : \mathsf{Type} \to$
$\Sigma \mid \cdot \mid \cdot \vdash t : A \to$
$\exists\ t'.\ \cdot \mid \cdot \mid \cdot \vdash t' : A$

Obtained by **inlining** definitions

Inlining

if $\Sigma \mid \Xi \mid \Gamma \vdash t : A$

then $\cdot \mid [\![\Xi]\!]\langle \kappa \rangle \mid [\![\Gamma]\!]\langle \kappa \rangle \vdash [\![t]\!]\langle \kappa \rangle : [\![A]\!]\langle \kappa \rangle$

where κ interprets the definitions of Σ

Inlining

if $\Sigma \mid \Xi \mid \Gamma \vdash t : A$

then $\cdot \mid [\![\Xi]\!]\langle \kappa \rangle \mid [\![\Gamma]\!]\langle \kappa \rangle \vdash [\![t]\!]\langle \kappa \rangle : [\![A]\!]\langle \kappa \rangle$

all definitions are removed

where κ interprets the definitions of Σ

Inlining

if $\Sigma \mid \Xi \mid \Gamma \vdash t : A$

then $\cdot \mid [\![\Xi]\!]\langle \kappa \rangle \mid [\![\Gamma]\!]\langle \kappa \rangle \vdash [\![t]\!]\langle \kappa \rangle : [\![A]\!]\langle \kappa \rangle$

all definitions are removed

where κ interprets the definitions of Σ

with κ fixed (and abstract):

$[\![x]\!] := x$

$[\![\lambda\,(x : A).\ t]\!] := \lambda\,(x : [\![A]\!]).\ [\![t]\!]$

$[\![u\ v]\!] := [\![u]\!]\ [\![v]\!]$

$[\![M.x]\!] := M.x$

$\boxed{[\![f\langle \xi \rangle]\!] := (\kappa\ f)[\![\xi]\!]}$

Inlining

if $\Sigma \mid \Xi \mid \Gamma \vdash t : A$

then $\cdot \mid [\![\Xi]\!]\langle \kappa \rangle \mid [\![\Gamma]\!]\langle \kappa \rangle \vdash [\![t]\!]\langle \kappa \rangle : [\![A]\!]\langle \kappa \rangle$

all definitions are removed

where κ interprets the definitions of Σ

with κ fixed (and abstract):

$[\![x]\!] := x$

$[\![\lambda\, (x : A).\ t]\!] := \lambda\, (x : [\![A]\!]).\ [\![t]\!]$

$[\![u\ v]\!] := [\![u]\!]\ [\![v]\!]$

$[\![M.x]\!] := M.x$

$\boxed{[\![f\langle \xi \rangle]\!] := (\kappa\ f)[\![\xi]\!]}$

κ is then defined by induction on ⊢ Σ such that

when $(\mathrm{def}\ f\langle\ \Xi'\ \rangle : A := t) \in \Sigma$ we have $\kappa\ f := [\![t]\!]\langle\ \kappa\_rec\ \rangle$

Inlining

if $\Sigma \mid \Xi \mid \Gamma \vdash t : A$

then $\cdot \mid [\![\Xi]\!]\langle \kappa \rangle \mid [\![\Gamma]\!]\langle \kappa \rangle \vdash [\![t]\!]\langle \kappa \rangle : [\![A]\!]\langle \kappa \rangle$

where κ interprets the definitions of Σ

all definitions are removed

with κ fixed (and abstract):

$[\![x]\!] := x$

$[\![\lambda\, (x : A).\ t]\!] := \lambda\, (x : [\![A]\!]).\ [\![t]\!]$

$[\![u\ v]\!] := [\![u]\!]\ [\![v]\!]$

$[\![M.x]\!] := M.x$

$\boxed{[\![f\langle \xi \rangle]\!] := (\kappa\ f)[\![\xi]\!]}$

κ is then defined by induction on $\vdash\ \Sigma$ such that

recursive call ok because t lives in an environment *smaller* than Σ

when `(def f⟨ Ξ’ ⟩ : A := t) ∈ Σ` we have `κ f := ⟦ t ⟧⟨ κ_rec ⟩`

Inlining

Compared to conservativity, we need full generality here

if $\Sigma \mid \Xi \mid \Gamma \vdash t : A$

then $\cdot \mid [\![\Xi]\!]\langle \kappa \rangle \mid [\![\Gamma]\!]\langle \kappa \rangle \vdash [\![t]\!]\langle \kappa \rangle : [\![A]\!]\langle \kappa \rangle$

all definitions are removed

where κ interprets the definitions of Σ

with κ fixed (and abstract):

$[\![x]\!] := x$ $[\![\lambda\, (x : A).\ t]\!] := \lambda\, (x : [\![A]\!]).\ [\![t]\!]$

$[\![u\ v]\!] := [\![u]\!]\ [\![v]\!]$ $[\![M.x]\!] := M.x$ $\boxed{[\![f\langle \xi \rangle]\!] := (\kappa\ f)[\![\xi]\!]}$

κ is then defined by induction on $\vdash\ \Sigma$ such that

recursive call ok because t lives in an environment *smaller* than Σ

when `(def f⟨ Ξ’ ⟩ : A := t) ∈ Σ` we have `κ f := ⟦ t ⟧⟨ κ_rec ⟩`

Checking for conversion

Characterising conversion with reduction

proof-of-concept confluence proof similar to the one presented earlier

Checking for conversion

Characterising conversion with reduction

proof-of-concept confluence proof similar to the one presented earlier

$$\begin{tikzcd}
\Sigma \mid \Xi \vdash u \equiv v \arrow[rr] & & \Sigma \mid \Xi \vdash u \leftrightsquigarrow v \arrow[dl, dashed, "\text{assuming confluence}"] \\
& \Sigma \mid \Xi \vdash u \bowtie v \arrow[ul, dashed, "\text{starting from well-typed terms}"] &
\end{tikzcd}$$

Checking for conversion

Characterising conversion with reduction

proof-of-concept confluence proof similar to the one presented earlier

$$\begin{tikzcd}
\Sigma \mid \Xi \vdash u \equiv v \arrow[rr] & & \Sigma \mid \Xi \vdash u \leftrightsquigarrow v \arrow[dl, dashed, "\text{assuming confluence}"] \\
& \Sigma \mid \Xi \vdash u \bowtie v \arrow[ul, dashed, "\text{starting from well-typed terms}"] &
\end{tikzcd}$$

Injectivity of Π-types

if `Π (x : A). B ≡ Π (x : C). D` then `A ≡ C` and `B ≡ D`

Checking for conversion

Characterising conversion with reduction

proof-of-concept confluence proof similar to the one presented earlier

$$
\begin{tikzcd}
\Sigma \mid \Xi \vdash u \equiv v \arrow[rr] & & \Sigma \mid \Xi \vdash u \leftrightsquigarrow v \arrow[dl, dashed, "\text{assuming confluence}"] \\
& \Sigma \mid \Xi \vdash u \bowtie v \arrow[ul, dashed, "\text{starting from well-typed terms}"] &
\end{tikzcd}
$$

Injectivity of Π-types

if `Π (x : A). B ≡ Π (x : C). D` then `A ≡ C` and `B ≡ D`

Subject reduction

assuming rewrite rules preserve types and are confluent

ROCQ prototype

on top of

The Rewster: type-preserving rewrite rules for the Coq proof assistant

Leray, Gilbert, Tabareau, Winterhalter

2024

ROCQ prototype

on top of

The Rewster: type-preserving rewrite rules for the Coq proof assistant

Leray, Gilbert, Tabareau, Winterhalter

2024

using the module system for interfaces and instances
and functors to quantify over interfaces

```
Module Type Plus.
  Symbol plus : ℕ → ℕ → ℕ.
  Rewrite Rules plus_r :=
  | plus 0 ?n => ?n
  | plus (S ?m) ?n => S (plus ?m ?n).
End Plus.
```

ROCQ prototype

on top of

The Rewster: type-preserving rewrite rules for the Coq proof assistant

Leray, Gilbert, Tabareau, Winterhalter

2024

using the module system for interfaces and instances
and functors to quantify over interfaces

```
Module Type Plus.
  Symbol plus : ℕ → ℕ → ℕ.
  Rewrite Rules plus_r :=
  | plus 0 ?n => ?n
  | plus (S ?m) ?n => S (plus ?m ?n).
End Plus.
```

```
Module UsePlus (P : Plus).
  Import P.

  Compute (plus 2 3). (* 5 *)

End UsePlus.
```

ROCQ prototype

on top of

The Rewster: type-preserving rewrite rules for the Coq proof assistant
Leray, Gilbert, Tabareau, Winterhalter
2024

using the module system for interfaces and instances
and functors to quantify over interfaces

```
Module Type Plus.
  Symbol plus : ℕ → ℕ → ℕ.
  Rewrite Rules plus_r :=
  | plus 0 ?n => ?n
  | plus (S ?m) ?n => S (plus ?m ?n).
End Plus.
```

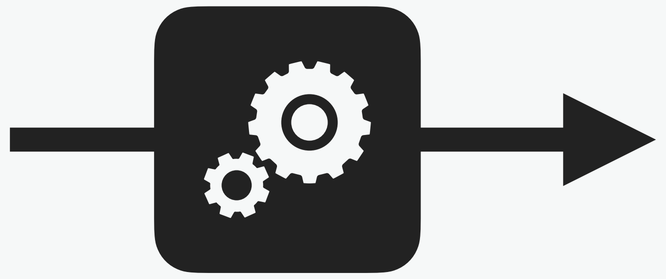
```
Module UsePlus (P : Plus).
  Import P.

  Compute (plus 2 3). (* 5 *)

End UsePlus.
```

```
Module Import PlusImpl : Plus.
  Definition plus := Nat.add.
End PlusImpl.
```

Conclusion

Conservative extension of MLTT with **local computation**

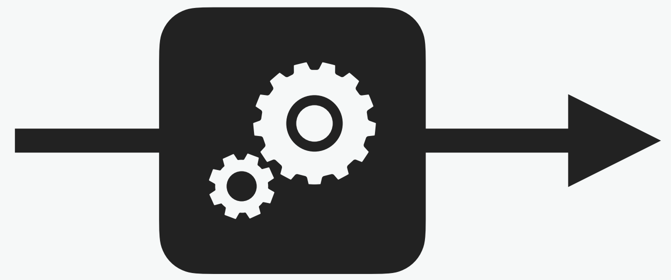
Conclusion



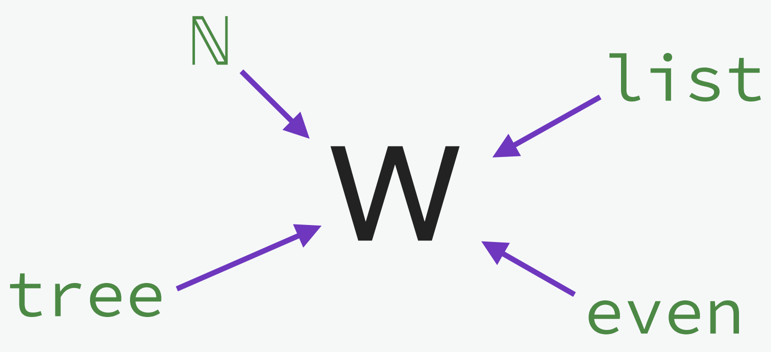
hide implementation details

Conservative extension of MLTT with **local computation**

Conclusion



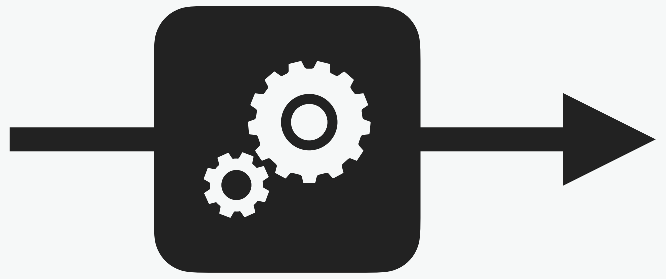
hide implementation details



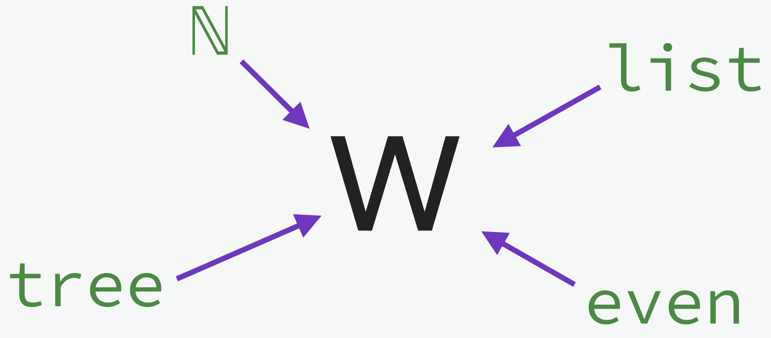
encode features

Conservative extension of MLTT with **local computation**

Conclusion



hide implementation details

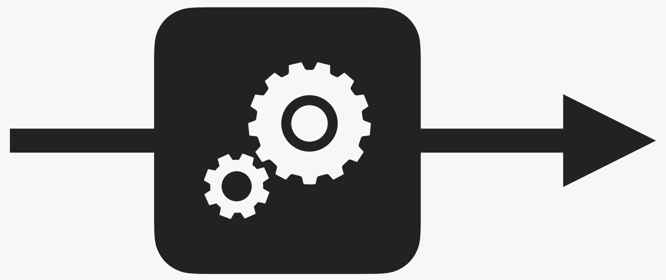


encode features

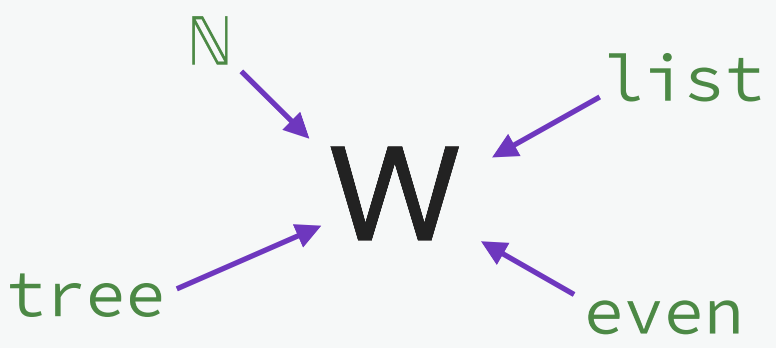
contained extensions (safer)

Conservative extension of MLTT with **local computation**

Conclusion



hide implementation details



encode features

contained extensions (safer)

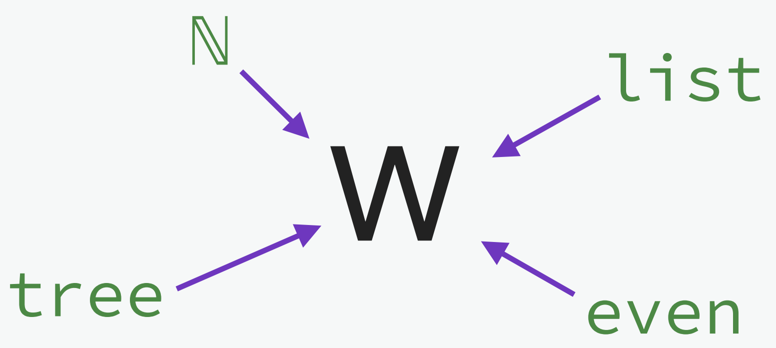
Conservative extension of MLTT with **local computation**

ROCQ APPROVED

/TheoWinterhalter/local-comp

Conclusion

hide implementation details



encode features

contained extensions (safer)

Conservative extension of MLTT with **local computation**

ROCQ APPROVED

/TheoWinterhalter/local-comp

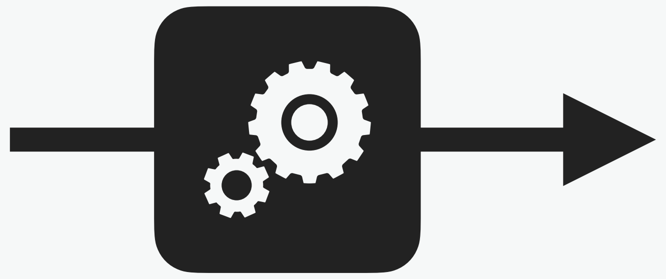
Perspectives (ongoing)

ROCQ

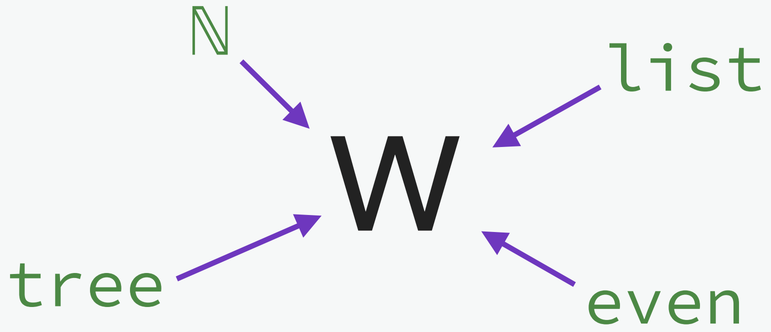
Concrete implementations

Agda

Conclusion



hide implementation details



encode features

contained extensions (safer)

Conservative extension of MLTT with **local computation**

ROCQ APPROVED

/TheoWinterhalter/local-comp

Perspectives (ongoing)

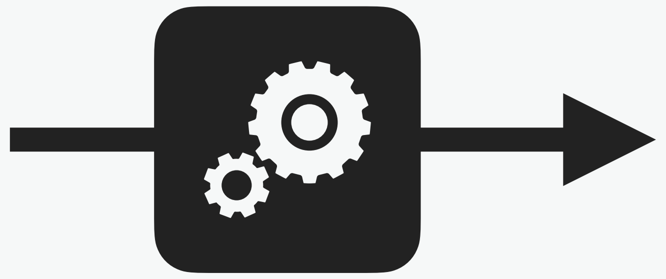
ROCQ

Concrete implementations

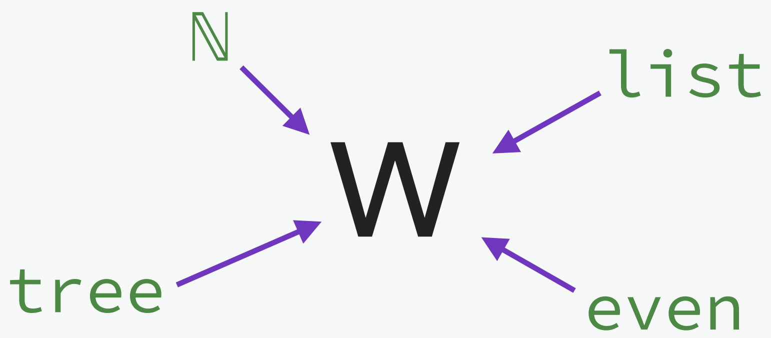
Agda

Propositional instances

Conclusion



hide implementation details



encode features

contained extensions (safer)

Conservative extension of MLTT with **local computation**

ROCQ APPROVED

/TheoWinterhalter/local-comp

Perspectives (ongoing)

ROCQ

Concrete implementations

Agda

👻

Propositional instances

Thank you!