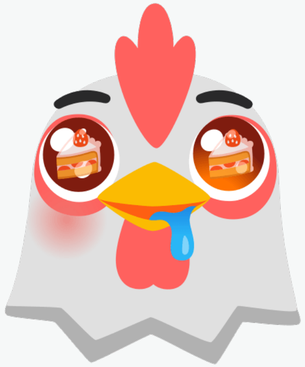
25 March 2026

Proofs and Algorithms seminar

Encode the Cake and Eat it Too

Controlling computation in type theory, locally



Théo Winterhalter

joint work with Yann Leray

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

Focus of my research 🔬 ⟶ **Dependent type theory**

Agda

Lean

Rocq (formerly Coq)

Computation in type theory

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

both sides reduce to 8

Computation in type theory

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

both sides reduce 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 reduce 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 reduce 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 reduce 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
```

both sides reduce 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)
```

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 reduce 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)
```

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

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

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

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

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
```

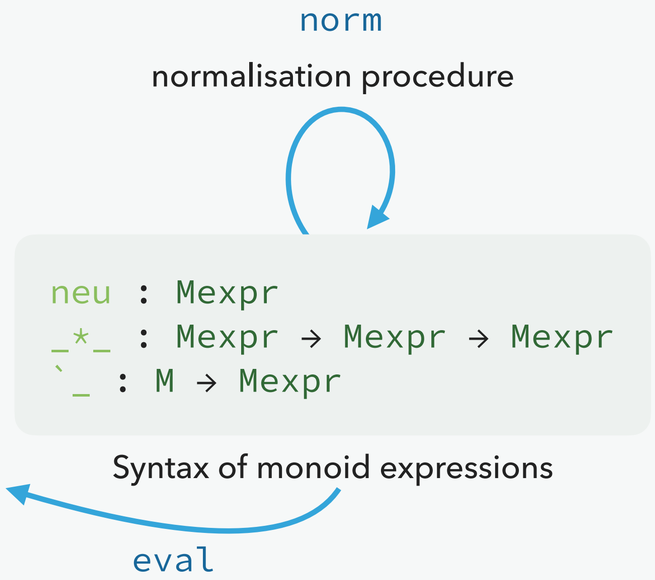
Syntax of monoid expressions

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
```



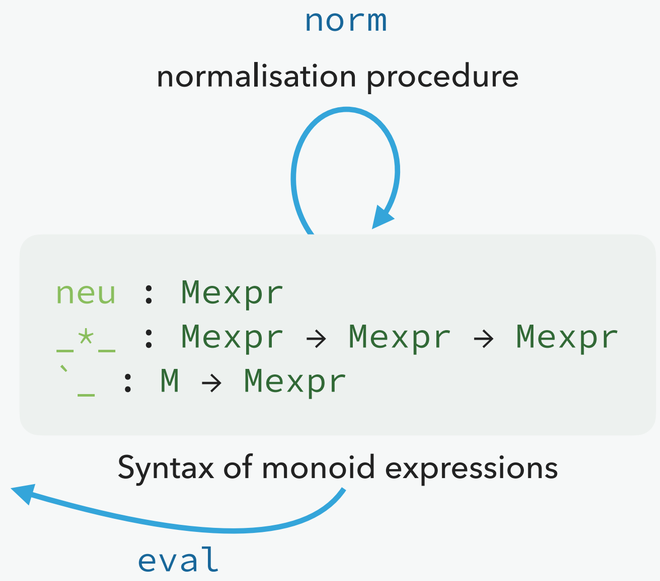
Proofs by computation ✨

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
```

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



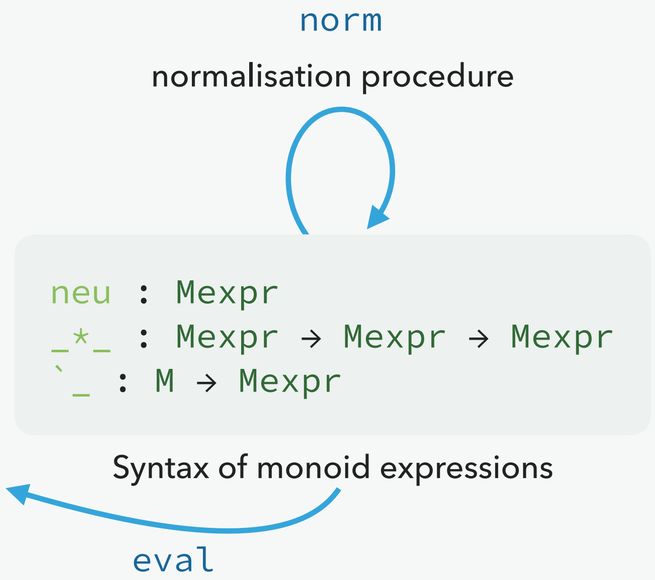

```
lemma norm_sound : ∀ u v. norm u = norm v → eval u = eval v
```

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
```




```
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

```
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
```

computation

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

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

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
```

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

Syntax of monoid expressions

norm

normalisation procedure

eval

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

computation

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

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

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

apply norm_sound

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
```

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

Syntax of monoid expressions

norm

normalisation procedure

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
```

doesn't scale 😵‍💫

```
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
```

doesn't scale 😵‍💫

```
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
```

```
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
```

doesn't scale 😵‍💫

```
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
```

```
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

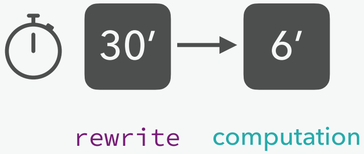
Interpret ring expressions as multivariate polynomials much more involved and useful than monoids

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

Sulfur: a Reflective Tactic for Substitution Simplification
Bouverot-Dupuis, Winterhalter, Stark, Maillard
2026



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

```
simpl
```

```
m = m + 0
```

```
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

```
simpl
```

```
m = m + 0
```

computation is stuck 😔

```
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
```

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
```

`induction n`

```
0 + m = m + 0
```
1

`simpl`

```
m = m + 0
```

computation is stuck 😔

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

```
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

`simpl`

```
m = m + 0
```

computation is stuck 😔

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

```
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

First extensions of calculus of constructions with rewrite rules

Multiparadigm computational models based on rewriting

Fernández (PhD supervised by Jouannaud)

1993

Modularity of Strong Normalisation and Confluence in the algebraic λ-cube

Barbanera, Fernández, Geuvers

1994

Controlling and extending computation in ITPs

First extensions of calculus of constructions with rewrite rules

Multiparadigm computational models based on rewriting

Fernández (PhD supervised by Jouannaud)

1993

Modularity of Strong Normalisation and Confluence in the algebraic λ-cube

Barbanera, Fernández, Geuvers

1994

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

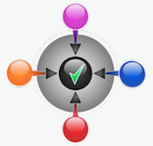
Dedukti: a logical framework based on the λΠ-calculus modulo theory

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

Dedukti: a logical framework based on the λΠ-calculus modulo theory

Assaf, Burel, Cauderlier, Delahaye, Dowek, Dubois, Gilbert, Halmagrand, Hermant, Saillard

2016

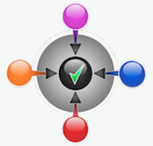
Sprinkles of extensionality for your vanilla type theory

Cockx, Abel

2016

ROCQ

prototypes



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

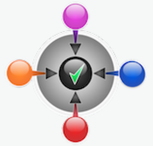
ROCQ

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

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

Leray, Gilbert, Tabareau, Winterhalter

2024

ROCQ

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

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

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

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

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!

Let’s 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 := …
```

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 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 : …
```


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

Computation rules must be assumed forever

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 ⟩ ⟩
```

Equations are verified implicitly

```
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)
```

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

Strictification

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

Example

Strictification

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

Strictification

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

Strictification

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

Strictification

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...

Potential example

Strictification

Type theory in type theory using a strictified syntax

Kaposi, Pujet

2025

Categories with families with strict substitution laws

$$(\Pi\ A\ B)[\sigma] = \Pi\ A[\sigma]\ B[\sigma\uparrow]$$

$$(\Pi\ A\ B)[\sigma] \equiv \Pi\ A[\sigma]\ B[\sigma\uparrow]$$

using rewrite rules in Agda

to implement observational equality

Picture from Loïc Pujet

Potential example

Strictification

Type theory in type theory using a strictified syntax

Kaposi, Pujet

2025

Categories with families with strict substitution laws

$$(\Pi\ A\ B)[\sigma] = \Pi\ A[\sigma]\ B[\sigma\uparrow]$$

$$(\Pi\ A\ B)[\sigma] \equiv \Pi\ A[\sigma]\ B[\sigma\uparrow]$$

using rewrite rules in Agda

to implement observational equality

🕵 We need to investigate if our solution could compensate slowdowns

Picture from Loïc Pujet

Example

Hiding implementation details

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

Example

Hiding implementation details

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

Dream

Use it with Equations (Magin and Sozeau)

```
equations shift : list ℕ → list ℕ
  shift (x :: l) := S x :: shift l
  shift [] := []
```

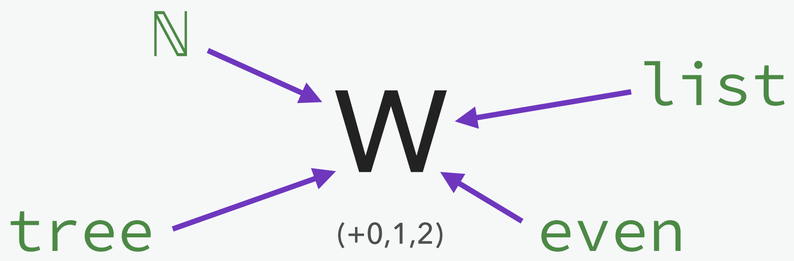
Agda in ROCQ

No more `shift_equation_1` appearing in the goal! 🎉

Example

Inductive types and extensions

Encode features using simpler ones



Why not W?

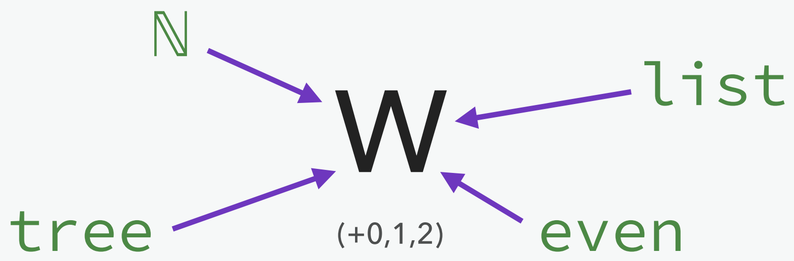
Hugunin

2021

Example

Inductive types and extensions

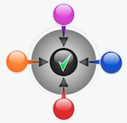
Encode features using simpler ones



Why not W?

Hugunin

2021

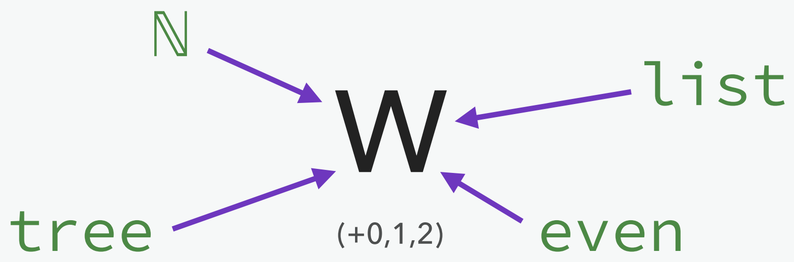


useful for Dedukti which doesn't support inductives

Example

Inductive types and extensions

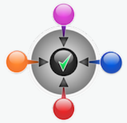
Encode features using simpler ones



Why not W?

Hugunin

2021



useful for Dedukti which
doesn't support inductives

Dream

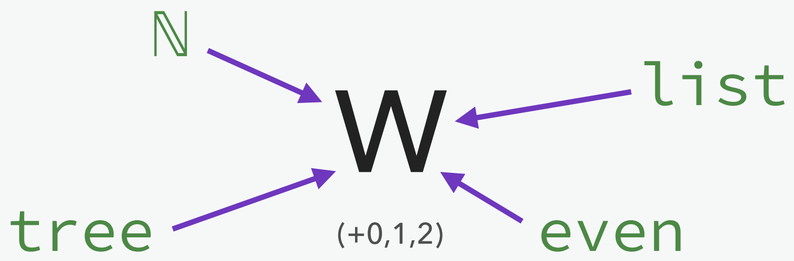
Use a similar idea to support…

inductive inductive types
inductive recursive types
HITs, QIITs, …

Example

Inductive types and extensions

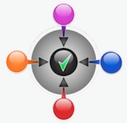
Encode features using simpler ones



Why not W?

Hugunin

2021



useful for Dedukti which doesn't support inductives

Dream

Use a similar idea to support…

inductive inductive types
inductive recursive types
HITs, QIITs, …

cheaper than extending ROCQ

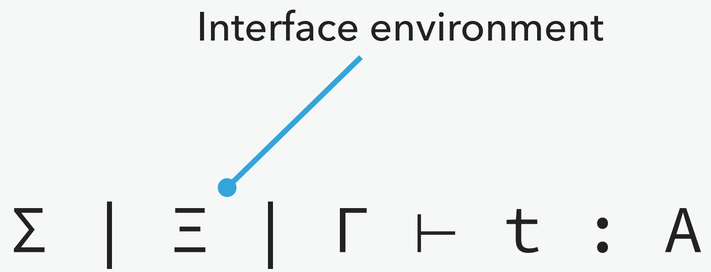
The type theory: LRTT

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

The type theory: LRTT

```
symb x : A
```

```
rule l ↪ r
```



The type theory: LRTT

```
symb x : A
```

```
def f⟨ Ξ' ⟩ : A := t
```

```
rule l ↪ r
```

Global environment

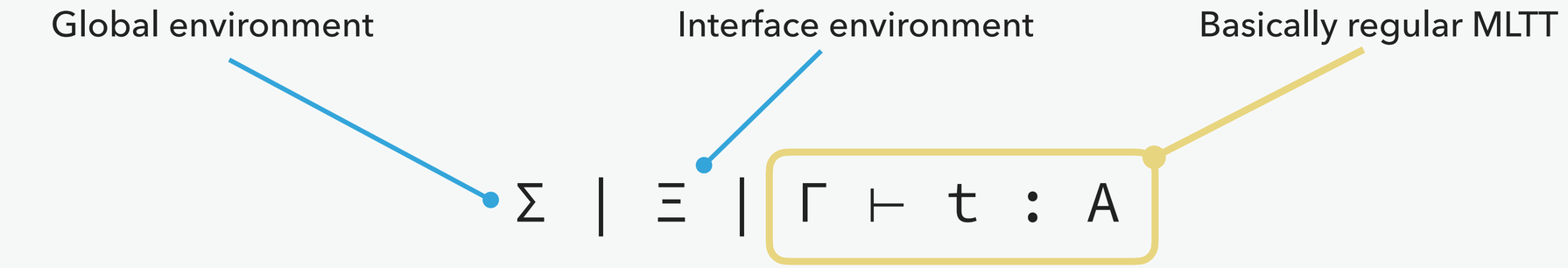
Interface environment

The type theory: LRTT

```
symb x : A
```

```
def f⟨ Ξ' ⟩ : A := t
```

```
rule l ↪ r
```



The type theory: LRTT

```
symb x : A
```

```
def f⟨ Ξ' ⟩ : A := t
```

```
rule l ↪ r
```

Global environment

Interface environment

Basically regular MLTT

Computation rule

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

The type theory: LRTT

```
symb x : A
```

```
def f⟨ Ξ' ⟩ : A := t
```

```
rule l ↪ r
```

Global environment

Interface environment

Basically regular MLTT

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

Computation rule

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

Unfolding rule

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

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**

```
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

more interesting:

Conservativity over MLTT

```
· | · | · ⊢ A : Type →
Σ | · | · ⊢ t : A →
∃ t'. · | · | · ⊢ 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 $\cdot \mid [\![\Xi]\!]\langle \kappa \rangle \mid [\![\Gamma]\!]\langle \kappa \rangle \vdash [\![t]\!]\langle \kappa \rangle : [\![A]\!]\langle \kappa \rangle$

where κ interprets the definitions of Σ

all definitions are removed

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):

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

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

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

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

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

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):

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

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

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

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

Inlining

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

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

all definitions are removed

where κ interprets the definitions of Σ

with κ fixed (and abstract):

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

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

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

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

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

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):

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

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

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

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

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

Inlining

if $\Sigma \mid \Xi \mid \Gamma \vdash t : A$ (and $\vdash \Sigma$)

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

where κ defined by induction on $\vdash \Sigma$

now we prove

Conservativity over MLTT

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

Inlining

if $\Sigma \mid \Xi \mid \Gamma \vdash t : A$ (and $\vdash \Sigma$)

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

where κ defined by induction on $\vdash \Sigma$

now we prove

Conservativity over MLTT

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

By inlining we get

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

Inlining

if $\Sigma \mid \Xi \mid \Gamma \vdash t : A$ (and $\vdash \Sigma$)

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

where κ defined by induction on $\vdash \Sigma$

now we prove

Conservativity over MLTT

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

By inlining we get

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

Since A is typed in MLTT
we know it cannot refer to any definition in Σ

$$[\![A]\!]\langle \kappa \rangle = A$$

Inlining

if $\Sigma \mid \Xi \mid \Gamma \vdash t : A$ (and $\vdash \Sigma$)

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

where κ defined by induction on $\vdash \Sigma$

now we prove

Conservativity over MLTT

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

By inlining we get

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

Since A is typed in MLTT
we know it cannot refer to any definition in Σ

$$[\![A]\!]\langle \kappa \rangle = A$$

Done!

What about type checking?

Checking conversion

We typically need properties like subject reduction and thus injectivity of type formers

so we characterise conversion with (confluent!) reduction

Checking conversion

We typically need properties like subject reduction and thus injectivity of type formers

so we characterise conversion with (confluent!) reduction

Unfolding rule (conversion)

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

Checking conversion

We typically need properties like subject reduction and thus injectivity of type formers

so we characterise conversion with (confluent!) reduction

Unfolding rule (conversion)

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

Unfolding rule (reduction)

$$\frac{(\mathtt{def}\ f\langle\ \Xi'\ \rangle\ :\ A\ :=\ t)\ \in\ \Sigma}{\Sigma\ |\ \Xi\ \vdash\ f\langle\xi\rangle\ \rightsquigarrow\ t\xi}$$

Checking conversion

We typically need properties like subject reduction and thus injectivity of type formers

so we characterise conversion with (confluent!) reduction

Unfolding rule (conversion)

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

Unfolding rule (reduction)

$$\frac{(\mathtt{def}\ f\langle\ \Xi'\ \rangle\ :\ A\ :=\ t)\ \in\ \Sigma}{\Sigma\ |\ \Xi\ \vdash\ f\langle\xi\rangle\ \twoheadrightarrow\ t\xi}$$

strip away side conditions to make confluence easier

Checking conversion

We typically need properties like subject reduction and thus injectivity of type formers

so we characterise conversion with (confluent!) reduction

Unfolding rule (conversion)

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

Unfolding rule (reduction)

$$\frac{(\texttt{def}\ f\langle\ \Xi'\ \rangle\ :\ A\ :=\ t) \in \Sigma}{\Sigma \mid \Xi \vdash f\langle\xi\rangle \twoheadrightarrow t\xi}$$

no `let` so
no local env

strip away side conditions to make confluence easier

Checking conversion

Computation rule (conversion)

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

Computation rule (reduction)

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

Unfolding rule (conversion)

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

Unfolding rule (reduction)

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

the rest is similar...

Checking conversion

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

Checking conversion

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

many-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$$

Checking conversion

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

many-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow^{*} v$$

congruence closure of reduction

$$\Sigma \mid \Xi \vdash u \nsim v$$

Checking conversion

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

many-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow^{*} v$$

congruence closure of reduction

$$\Sigma \mid \Xi \vdash u \leftrightsquigarrow v$$

joinability ($\rightsquigarrow^{*} \cdot {}^{*}\!\!\leftsquigarrow$)

$$\Sigma \mid \Xi \vdash u \bowtie v$$

Checking conversion

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

many-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$$

congruence closure of reduction

$$\Sigma \mid \Xi \vdash u \nsim v$$

joinability ($\rightsquigarrow^{*}\cdot{}^{*}\!\!\leftsquigarrow$)

$$\Sigma \mid \Xi \vdash u \bowtie v$$

$$\Sigma \mid \Xi \vdash u \equiv v \longrightarrow \Sigma \mid \Xi \vdash u \leftrightsquigarrow v$$

Checking conversion

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

many-step reduction

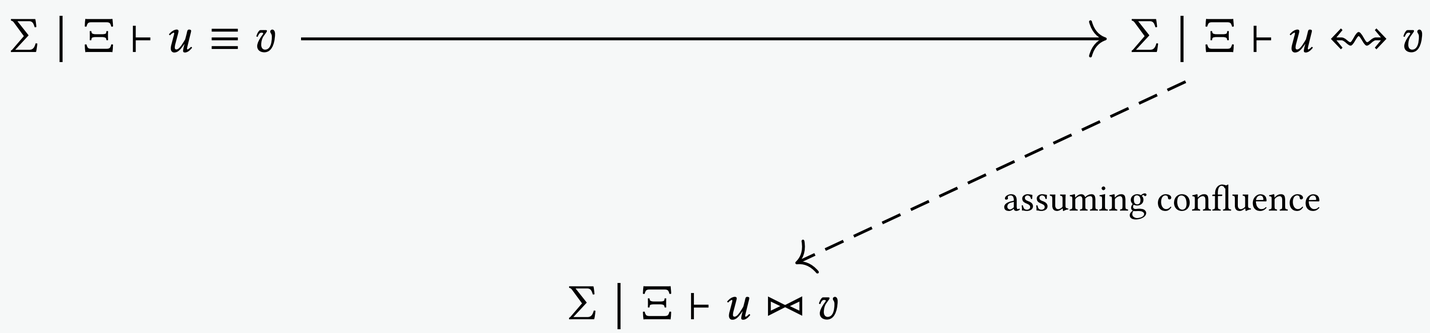
$$\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$$

congruence closure of reduction

$$\Sigma \mid \Xi \vdash u \leftrightsquigarrow v$$

joinability ($\rightsquigarrow^* \cdot {}^* \!\!\leftsquigarrow$)

$$\Sigma \mid \Xi \vdash u \bowtie v$$



Checking conversion

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

many-step reduction

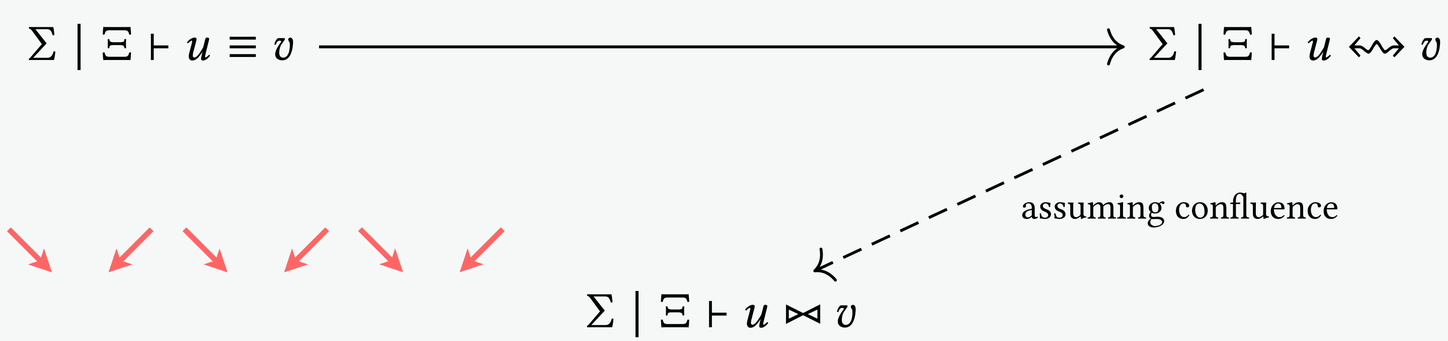
$$\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$$

congruence closure of reduction

$$\Sigma \mid \Xi \vdash u \leftrightsquigarrow v$$

joinability ($\rightsquigarrow^* \cdot {}^* \leftsquigarrow$)

$$\Sigma \mid \Xi \vdash u \bowtie v$$



Checking conversion

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

many-step reduction

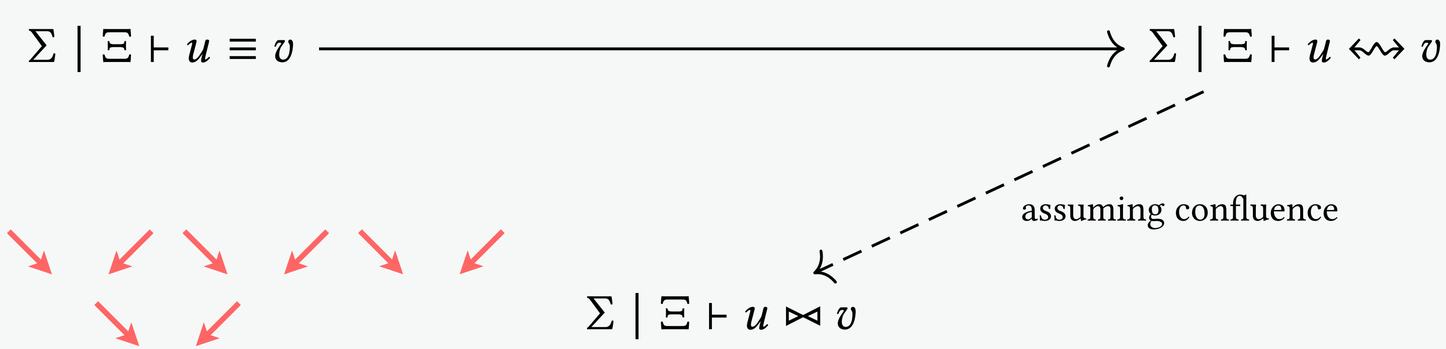
$$\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$$

congruence closure of reduction

$$\Sigma \mid \Xi \vdash u \leftrightsquigarrow v$$

joinability ($\rightsquigarrow^* \cdot {}^* \!\!\leftsquigarrow$)

$$\Sigma \mid \Xi \vdash u \bowtie v$$



Checking conversion

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

many-step reduction

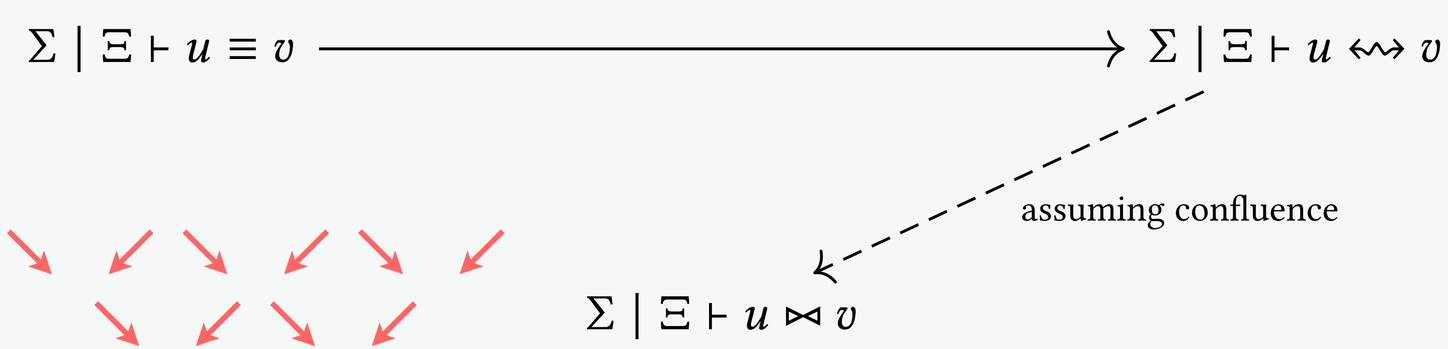
$$\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$$

congruence closure of reduction

$$\Sigma \mid \Xi \vdash u \leftrightsquigarrow v$$

joinability ($\rightsquigarrow^* \cdot {}^* \leftsquigarrow$)

$$\Sigma \mid \Xi \vdash u \bowtie v$$



Checking conversion

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

many-step reduction

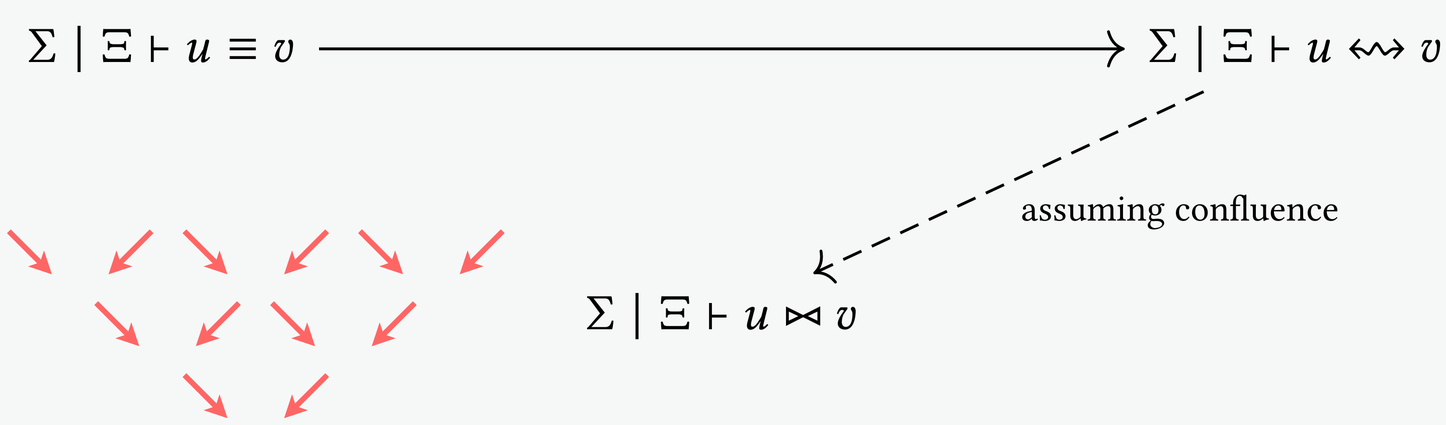
$$\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$$

congruence closure of reduction

$$\Sigma \mid \Xi \vdash u \leftrightsquigarrow v$$

joinability ($\rightsquigarrow^* \cdot {}^* \leftsquigarrow$)

$$\Sigma \mid \Xi \vdash u \bowtie v$$



Checking conversion

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

many-step reduction

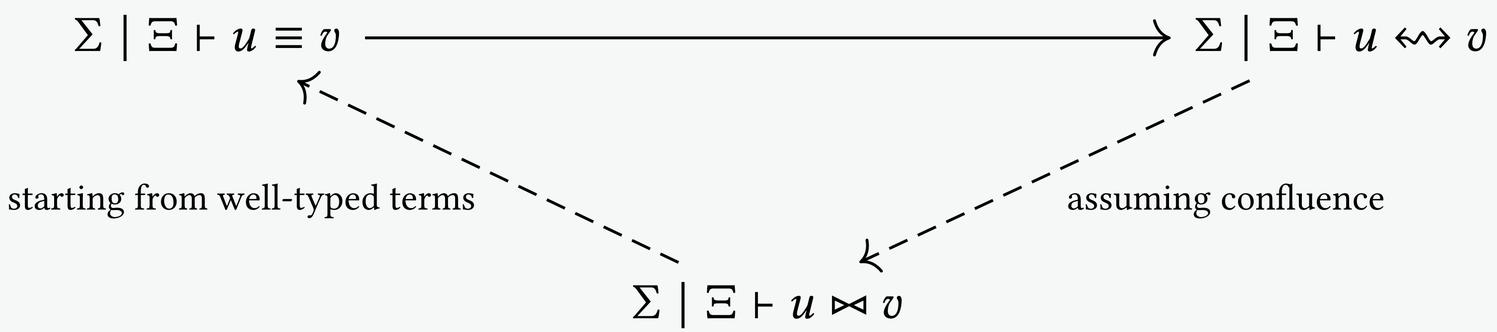
$$\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$$

congruence closure of reduction

$$\Sigma \mid \Xi \vdash u \leftrightsquigarrow v$$

joinability ($\rightsquigarrow^* \cdot {}^* \leftsquigarrow$)

$$\Sigma \mid \Xi \vdash u \bowtie v$$



Checking conversion

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

many-step reduction

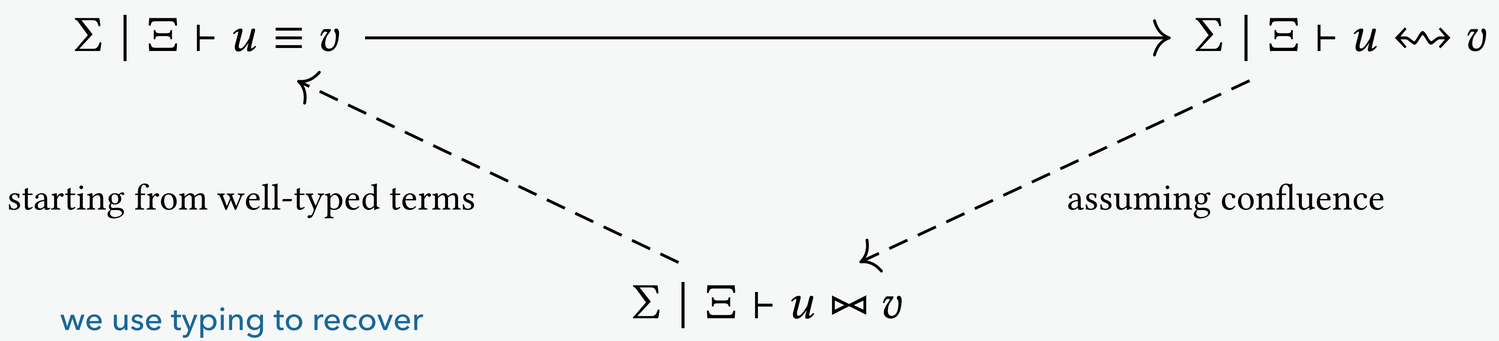
$$\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$$

congruence closure of reduction

$$\Sigma \mid \Xi \vdash u \leftrightsquigarrow v$$

joinability ($\rightsquigarrow^* \cdot {}^* \!\!\leftsquigarrow$)

$$\Sigma \mid \Xi \vdash u \bowtie v$$



we use typing to recover
the missing side conditions!

Checking conversion

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

many-step reduction

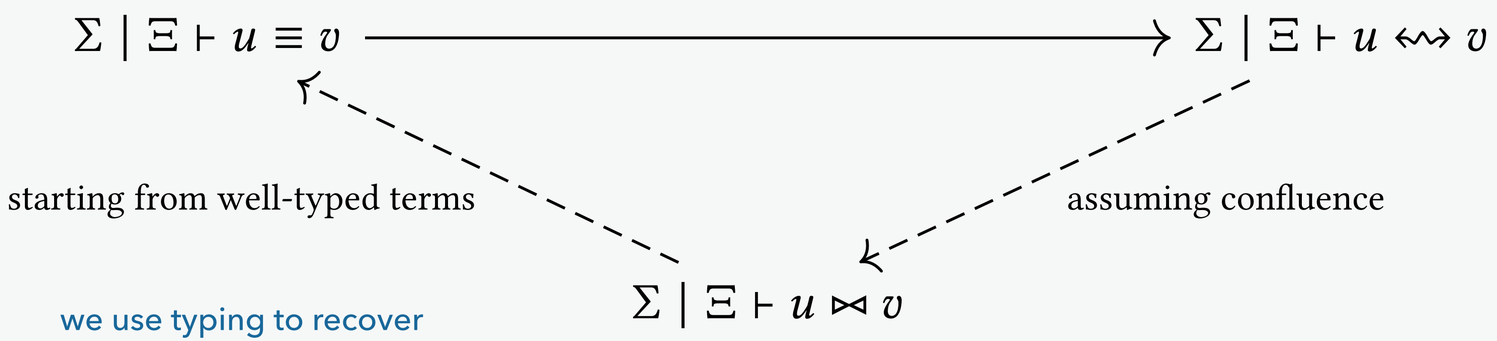
$$\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$$

congruence closure of reduction

$$\Sigma \mid \Xi \vdash u \nsim v$$

joinability ($\rightsquigarrow^* \cdot {}^* \!\!\leftsquigarrow$)

$$\Sigma \mid \Xi \vdash u \bowtie v$$



we use typing to recover
the missing side conditions!

easy?

From joinability to conversion (failed attempt)

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

many-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$$

congruence closure of reduction

$$\Sigma \mid \Xi \vdash u \leftrightsquigarrow v$$

joinability ($\rightsquigarrow^* \cdot {}^* \leftsquigarrow$)

$$\Sigma \mid \Xi \vdash u \bowtie v$$

Assume that if $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ and $\Sigma \mid \Xi \mid \Gamma \vdash u : A$ then $\Sigma \mid \Xi \vdash u \equiv v$

From joinability to conversion (failed attempt)

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

many-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$$

congruence closure of reduction

$$\Sigma \mid \Xi \vdash u \leftrightsquigarrow v$$

joinability ($\rightsquigarrow^* \cdot {}^* \leftsquigarrow$)

$$\Sigma \mid \Xi \vdash u \bowtie v$$

Assume that if $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ and $\Sigma \mid \Xi \mid \Gamma \vdash u : A$ then $\Sigma \mid \Xi \vdash u \equiv v$

By induction on $\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$ such that $\Sigma \mid \Xi \mid \Gamma \vdash u : A$ we show $\Sigma \mid \Xi \vdash u \equiv v$

From joinability to conversion (failed attempt)

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

many-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$$

congruence closure of reduction

$$\Sigma \mid \Xi \vdash u \leftrightsquigarrow v$$

joinability ($\rightsquigarrow^* \cdot {}^* \!\!\leftsquigarrow$)

$$\Sigma \mid \Xi \vdash u \bowtie v$$

Assume that if $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ and $\Sigma \mid \Xi \mid \Gamma \vdash u : A$ then $\Sigma \mid \Xi \vdash u \equiv v$

By induction on $\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$ such that $\Sigma \mid \Xi \mid \Gamma \vdash u : A$ we show $\Sigma \mid \Xi \vdash u \equiv v$

We have $\Sigma \mid \Xi \vdash u \rightsquigarrow w$ and $\Sigma \mid \Xi \vdash w \rightsquigarrow^* v$

From joinability to conversion (failed attempt)

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

many-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$$

congruence closure of reduction

$$\Sigma \mid \Xi \vdash u \leftrightsquigarrow v$$

joinability ($\rightsquigarrow^* \cdot {}^* \!\!\leftsquigarrow$)

$$\Sigma \mid \Xi \vdash u \bowtie v$$

Assume that if $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ and $\Sigma \mid \Xi \mid \Gamma \vdash u : A$ then $\Sigma \mid \Xi \vdash u \equiv v$

By induction on $\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$ such that $\Sigma \mid \Xi \mid \Gamma \vdash u : A$ we show $\Sigma \mid \Xi \vdash u \equiv v$

We have $\Sigma \mid \Xi \vdash u \rightsquigarrow w$ and $\Sigma \mid \Xi \vdash w \rightsquigarrow^* v$ thus $\Sigma \mid \Xi \vdash u \equiv w$

From joinability to conversion (failed attempt)

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

many-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$$

congruence closure of reduction

$$\Sigma \mid \Xi \vdash u \leftrightsquigarrow v$$

joinability ($\rightsquigarrow^*\cdot{}^*\!\!\leftsquigarrow$)

$$\Sigma \mid \Xi \vdash u \bowtie v$$

Assume that if $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ and $\Sigma \mid \Xi \mid \Gamma \vdash u : A$ then $\Sigma \mid \Xi \vdash u \equiv v$

By induction on $\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$ such that $\Sigma \mid \Xi \mid \Gamma \vdash u : A$ we show $\Sigma \mid \Xi \vdash u \equiv v$

We have $\Sigma \mid \Xi \vdash u \rightsquigarrow w$ and $\Sigma \mid \Xi \vdash w \rightsquigarrow^* v$ thus $\Sigma \mid \Xi \vdash u \equiv w$

To apply IH and conclude with $\Sigma \mid \Xi \vdash w \equiv v$ we need $\Sigma \mid \Xi \mid \Gamma \vdash w : A$

From joinability to conversion (failed attempt)

conversion

$$\Sigma \mid \Xi \mid \Gamma \vdash u \equiv v$$

one-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow v$$

many-step reduction

$$\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$$

congruence closure of reduction

$$\Sigma \mid \Xi \vdash u \leftrightsquigarrow v$$

joinability ($\rightsquigarrow^* \cdot {}^* \!\!\leftsquigarrow$)

$$\Sigma \mid \Xi \vdash u \bowtie v$$

Assume that if $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ and $\Sigma \mid \Xi \mid \Gamma \vdash u : A$ then $\Sigma \mid \Xi \vdash u \equiv v$

By induction on $\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$ such that $\Sigma \mid \Xi \mid \Gamma \vdash u : A$ we show $\Sigma \mid \Xi \vdash u \equiv v$

We have $\Sigma \mid \Xi \vdash u \rightsquigarrow w$ and $\Sigma \mid \Xi \vdash w \rightsquigarrow^* v$ thus $\Sigma \mid \Xi \vdash u \equiv w$

To apply IH and conclude with $\Sigma \mid \Xi \vdash w \equiv v$ we need $\Sigma \mid \Xi \mid \Gamma \vdash w : A$ meaning subject reduction!

Subject reduction (failed attempt)

Subject reduction

If $\Sigma \mid \Xi \mid \Gamma \vdash u : T$ and $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ then $\Sigma \mid \Xi \mid \Gamma \vdash v : T$

Subject reduction (failed attempt)

Subject reduction

If $\Sigma \mid \Xi \mid \Gamma \vdash u : T$ and $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ then $\Sigma \mid \Xi \mid \Gamma \vdash v : T$

By induction on reduction, β case:

$$\Sigma \mid \Xi \vdash (\lambda\ (x : A).\ t)\ u \rightsquigarrow t[x := u]$$

Subject reduction (failed attempt)

Subject reduction

If $\Sigma \mid \Xi \mid \Gamma \vdash u : T$ and $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ then $\Sigma \mid \Xi \mid \Gamma \vdash v : T$

By induction on reduction, β case:

$$\Sigma \mid \Xi \vdash (\lambda\ (x : A).\ t)\ u \rightsquigarrow t[x := u]$$

inversion on typing on $\Sigma \mid \Xi \mid \Gamma \vdash (\lambda\ (x : A).\ t)\ u : T$ gives

Subject reduction (failed attempt)

Subject reduction

If `Σ | Ξ | Γ ⊢ u : T` and `Σ | Ξ ⊢ u ⇝ v` then `Σ | Ξ | Γ ⊢ v : T`

By induction on reduction, β case:

```
Σ | Ξ ⊢ (λ (x : A). t) u ⇝ t[x := u]
```

inversion on typing on `Σ | Ξ | Γ ⊢ (λ (x : A). t) u : T` gives

```
Σ | Ξ | Γ ⊢ λ (x : A). t : Π (x : C). D
```

Subject reduction (failed attempt)

Subject reduction

If $\Sigma \mid \Xi \mid \Gamma \vdash u : T$ and $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ then $\Sigma \mid \Xi \mid \Gamma \vdash v : T$

By induction on reduction, β case:

$$\Sigma \mid \Xi \vdash (\lambda\ (x : A).\ t)\ u \rightsquigarrow t[x := u]$$

inversion on typing on $\Sigma \mid \Xi \mid \Gamma \vdash (\lambda\ (x : A).\ t)\ u : T$ gives

$\Sigma \mid \Xi \mid \Gamma \vdash \lambda\ (x : A).\ t : \Pi\ (x : C).\ D$ $\Sigma \mid \Xi \mid \Gamma \vdash u : C$

Subject reduction (failed attempt)

Subject reduction

If $\Sigma \mid \Xi \mid \Gamma \vdash u : T$ and $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ then $\Sigma \mid \Xi \mid \Gamma \vdash v : T$

By induction on reduction, β case:

$$\Sigma \mid \Xi \vdash (\lambda\ (x : A).\ t)\ u \rightsquigarrow t[x := u]$$

inversion on typing on $\Sigma \mid \Xi \mid \Gamma \vdash (\lambda\ (x : A).\ t)\ u : T$ gives

$\Sigma \mid \Xi \mid \Gamma \vdash \lambda\ (x : A).\ t : \Pi\ (x : C).\ D$ $\quad$ $\Sigma \mid \Xi \mid \Gamma \vdash u : C$ $\quad$ $\Sigma \mid \Xi \mid \Gamma \vdash D[x := u] \equiv T$

Subject reduction (failed attempt)

Subject reduction

If $\Sigma \mid \Xi \mid \Gamma \vdash u : T$ and $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ then $\Sigma \mid \Xi \mid \Gamma \vdash v : T$

By induction on reduction, β case:

$$\Sigma \mid \Xi \vdash (\lambda\ (x : A).\ t)\ u \rightsquigarrow t[x := u]$$

inversion on typing on $\Sigma \mid \Xi \mid \Gamma \vdash (\lambda\ (x : A).\ t)\ u : T$ gives

$\Sigma \mid \Xi \mid \Gamma \vdash \lambda\ (x : A).\ t : \Pi\ (x : C).\ D$ $\quad \Sigma \mid \Xi \mid \Gamma \vdash u : C$ $\quad \Sigma \mid \Xi \mid \Gamma \vdash D[x := u] \equiv T$

inversion again

Subject reduction (failed attempt)

Subject reduction

If $\Sigma \mid \Xi \mid \Gamma \vdash u : T$ and $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ then $\Sigma \mid \Xi \mid \Gamma \vdash v : T$

By induction on reduction, β case:

$$\Sigma \mid \Xi \vdash (\lambda\ (x : A).\ t)\ u \rightsquigarrow t[x := u]$$

inversion on typing on $\Sigma \mid \Xi \mid \Gamma \vdash (\lambda\ (x : A).\ t)\ u : T$ gives

$\Sigma \mid \Xi \mid \Gamma \vdash \lambda\ (x : A).\ t : \Pi\ (x : C).\ D$ $\quad \Sigma \mid \Xi \mid \Gamma \vdash u : C$ $\quad \Sigma \mid \Xi \mid \Gamma \vdash D[x := u] \equiv T$

inversion again $\Sigma \mid \Xi \mid \Gamma, x : A \vdash t : B$

Subject reduction (failed attempt)

Subject reduction

If $\Sigma \mid \Xi \mid \Gamma \vdash u : T$ and $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ then $\Sigma \mid \Xi \mid \Gamma \vdash v : T$

By induction on reduction, β case:

$$\Sigma \mid \Xi \vdash (\lambda\ (x : A).\ t)\ u \rightsquigarrow t[x := u]$$

inversion on typing on $\Sigma \mid \Xi \mid \Gamma \vdash (\lambda\ (x : A).\ t)\ u : T$ gives

$\Sigma \mid \Xi \mid \Gamma \vdash \lambda\ (x : A).\ t : \Pi\ (x : C).\ D$ $\quad \Sigma \mid \Xi \mid \Gamma \vdash u : C$ $\quad \Sigma \mid \Xi \mid \Gamma \vdash D[x := u] \equiv T$

inversion again $\Sigma \mid \Xi \mid \Gamma, x : A \vdash t : B$ $\quad \Sigma \mid \Xi \mid \Gamma \vdash \Pi\ (x : A).\ B \equiv \Pi\ (x : C).\ D$

Subject reduction (failed attempt)

Subject reduction

If $\Sigma \mid \Xi \mid \Gamma \vdash u : T$ and $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ then $\Sigma \mid \Xi \mid \Gamma \vdash v : T$

By induction on reduction, β case:

$$\Sigma \mid \Xi \vdash (\lambda\ (x : A).\ t)\ u \rightsquigarrow t[x := u]$$

inversion on typing on $\Sigma \mid \Xi \mid \Gamma \vdash (\lambda\ (x : A).\ t)\ u : T$ gives

$\Sigma \mid \Xi \mid \Gamma \vdash \lambda\ (x : A).\ t : \Pi\ (x : C).\ D$ $\quad \Sigma \mid \Xi \mid \Gamma \vdash u : C$ $\quad \Sigma \mid \Xi \mid \Gamma \vdash D[x := u] \equiv T$

inversion again $\Sigma \mid \Xi \mid \Gamma, x : A \vdash t : B$ $\quad \Sigma \mid \Xi \mid \Gamma \vdash \Pi\ (x : A).\ B \equiv \Pi\ (x : C).\ D$

to conclude $\Sigma \mid \Xi \mid \Gamma \vdash t[x := u] : D[x := u]$ we want

$\Sigma \mid \Xi \mid \Gamma \vdash A \equiv C$ $\quad \Sigma \mid \Xi \mid \Gamma \vdash B[x := u] \equiv D[x := u]$

Subject reduction (failed attempt)

Subject reduction

If $\Sigma \mid \Xi \mid \Gamma \vdash u : T$ and $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ then $\Sigma \mid \Xi \mid \Gamma \vdash v : T$

By induction on reduction, β case:

$$\Sigma \mid \Xi \vdash (\lambda\ (x : A).\ t)\ u \rightsquigarrow t[x := u]$$

inversion on typing on $\Sigma \mid \Xi \mid \Gamma \vdash (\lambda\ (x : A).\ t)\ u : T$ gives

$\Sigma \mid \Xi \mid \Gamma \vdash \lambda\ (x : A).\ t : \Pi\ (x : C).\ D$ $\quad$ $\Sigma \mid \Xi \mid \Gamma \vdash u : C$ $\quad$ $\Sigma \mid \Xi \mid \Gamma \vdash D[x := u] \equiv T$

inversion again $\Sigma \mid \Xi \mid \Gamma, x : A \vdash t : B$ $\quad$ $\Sigma \mid \Xi \mid \Gamma \vdash \Pi\ (x : A).\ B \equiv \Pi\ (x : C).\ D$

injectivity of Π

to conclude $\Sigma \mid \Xi \mid \Gamma \vdash t[x := u] : D[x := u]$ we want

$\Sigma \mid \Xi \mid \Gamma \vdash A \equiv C$ $\quad$ $\Sigma \mid \Xi \mid \Gamma \vdash B[x := u] \equiv D[x := u]$

Injectivity of Π (failed attempt)

Injectivity of Π

If `Σ | Ξ | Γ ⊢ Π (x : A). B ≡ Π (x : C). D` then
`Σ | Ξ | Γ ⊢ A ≡ C` and `Σ | Ξ | Γ, x : A ⊢ B ≡ D`

Injectivity of Π (failed attempt)

Injectivity of Π

If $\Sigma \mid \Xi \mid \Gamma \vdash \Pi\ (x : A).\ B \equiv \Pi\ (x : C).\ D$ then
$\Sigma \mid \Xi \mid \Gamma \vdash A \equiv C$ and $\Sigma \mid \Xi \mid \Gamma, x : A \vdash B \equiv D$

assuming confluence $\Sigma \mid \Xi \vdash \Pi\ (x : A).\ B \bowtie \Pi\ (x : C).\ D$

Injectivity of Π (failed attempt)

Injectivity of Π

If $\Sigma \mid \Xi \mid \Gamma \vdash \Pi\ (x : A).\ B \equiv \Pi\ (x : C).\ D$ then
$\Sigma \mid \Xi \mid \Gamma \vdash A \equiv C$ and $\Sigma \mid \Xi \mid \Gamma, x : A \vdash B \equiv D$

assuming confluence $\Sigma \mid \Xi \vdash \Pi\ (x : A).\ B \bowtie \Pi\ (x : C).\ D$

since no reduction rules on Π besides congruence

$\Sigma \mid \Xi \vdash A \bowtie C$ $\Sigma \mid \Xi \vdash B \bowtie D$

Injectivity of Π (failed attempt)

Injectivity of Π

If $\Sigma \mid \Xi \mid \Gamma \vdash \Pi\ (x : A).\ B \equiv \Pi\ (x : C).\ D$ then
$\Sigma \mid \Xi \mid \Gamma \vdash A \equiv C$ and $\Sigma \mid \Xi \mid \Gamma, x : A \vdash B \equiv D$

assuming confluence $\Sigma \mid \Xi \vdash \Pi\ (x : A).\ B \bowtie \Pi\ (x : C).\ D$

since no reduction rules on Π besides congruence

$\Sigma \mid \Xi \vdash A \bowtie C$ $\Sigma \mid \Xi \vdash B \bowtie D$

we conclude by going back to conversion 🎉

Injectivity of Π (failed attempt)

Injectivity of Π

If $\Sigma \mid \Xi \mid \Gamma \vdash \Pi\ (x : A).\ B \equiv \Pi\ (x : C).\ D$ then
$\Sigma \mid \Xi \mid \Gamma \vdash A \equiv C$ and $\Sigma \mid \Xi \mid \Gamma, x : A \vdash B \equiv D$

assuming confluence $\Sigma \mid \Xi \vdash \Pi\ (x : A).\ B \bowtie \Pi\ (x : C).\ D$

since no reduction rules on Π besides congruence ⚠️ needs restriction on Ξ

$\Sigma \mid \Xi \vdash A \bowtie C$ $\Sigma \mid \Xi \vdash B \bowtie D$

we conclude by going back to conversion 🎉

Injectivity of Π (failed attempt)

Injectivity of Π

If $\Sigma \mid \Xi \mid \Gamma \vdash \Pi\ (x : A).\ B \equiv \Pi\ (x : C).\ D$ then
$\Sigma \mid \Xi \mid \Gamma \vdash A \equiv C$ and $\Sigma \mid \Xi \mid \Gamma, x : A \vdash B \equiv D$

assuming confluence $\Sigma \mid \Xi \vdash \Pi\ (x : A).\ B \bowtie \Pi\ (x : C).\ D$

since no reduction rules on Π besides congruence

⚠️ needs restriction on Ξ

$\Sigma \mid \Xi \vdash A \bowtie C$ $\Sigma \mid \Xi \vdash B \bowtie D$

~~we conclude by going back to conversion~~ 🎉

we need the joinability to conversion
theorem that we're trying to prove!

Injectivity of Π (failed attempt)

Injectivity of Π

If $\Sigma \mid \Xi \mid \Gamma \vdash \Pi\ (x : A).\ B \equiv \Pi\ (x : C).\ D$ then
$\Sigma \mid \Xi \mid \Gamma \vdash A \equiv C$ and $\Sigma \mid \Xi \mid \Gamma, x : A \vdash B \equiv D$

assuming confluence $\Sigma \mid \Xi \vdash \Pi\ (x : A).\ B \bowtie \Pi\ (x : C).\ D$

since no reduction rules on Π besides congruence

⚠️ needs restriction on Ξ

$\Sigma \mid \Xi \vdash A \bowtie C$ $\Sigma \mid \Xi \vdash B \bowtie D$

~~we conclude by going back to conversion~~ 🎉

we need the joinability to conversion
theorem that we're trying to prove!

we're going to use a property weaker than typing to break the loop

From joinability to conversion (for real this time)

$$\Sigma \mid \Xi \ltimes t$$

instances that appear in `t`
respect the equations of the interface they correspond to

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

From joinability to conversion (for real this time)

$$\Sigma \mid \Xi \ltimes t$$

instances that appear in `t`
respect the equations of the interface they correspond to

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

Lemma If $\Sigma \mid \Xi \mid \Gamma \vdash u : T$ then $\Sigma \mid \Xi \ltimes u$

From joinability to conversion (for real this time)

$$\Sigma \mid \Xi \ltimes t$$

instances that appear in t
respect the equations of the interface they correspond to

$$\frac{(\mathsf{def}\ f\langle\ \Xi'\ \rangle : A := t) \in \Sigma \qquad \Sigma \mid \Xi \vdash \xi \vDash \Xi' \qquad \forall x.\ \Sigma \mid \Xi \ltimes \xi(x)}{\Sigma \mid \Xi \ltimes f\langle\xi\rangle \equiv t\xi}$$

Lemma If $\Sigma \mid \Xi \mid \Gamma \vdash u : T$ then $\Sigma \mid \Xi \ltimes u$

Lemma If $\Sigma \mid \Xi \ltimes u$ and $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ then $\Sigma \mid \Xi \ltimes v$

From joinability to conversion (for real this time)

$$\Sigma \mid \Xi \ltimes t$$

instances that appear in t
respect the equations of the interface they correspond to

$$\frac{(\mathsf{def}\ f\langle\ \Xi'\ \rangle : A := t) \in \Sigma \qquad \Sigma \mid \Xi \vdash \xi \vDash \Xi' \qquad \forall x.\ \Sigma \mid \Xi \ltimes \xi(x)}{\Sigma \mid \Xi \ltimes f\langle\xi\rangle \equiv t\xi}$$

Lemma If $\Sigma \mid \Xi \mid \Gamma \vdash u : T$ then $\Sigma \mid \Xi \ltimes u$

Lemma If $\Sigma \mid \Xi \ltimes u$ and $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ then $\Sigma \mid \Xi \ltimes v$

Lemma If $\Sigma \mid \Xi \ltimes u$ and $\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$ then $\Sigma \mid \Xi \vdash u \equiv v$

From joinability to conversion (for real this time)

$$\Sigma \mid \Xi \ltimes t$$

instances that appear in t
respect the equations of the interface they correspond to

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

Lemma If $\Sigma \mid \Xi \mid \Gamma \vdash u : T$ then $\Sigma \mid \Xi \ltimes u$

Lemma If $\Sigma \mid \Xi \ltimes u$ and $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ then $\Sigma \mid \Xi \ltimes v$

Lemma If $\Sigma \mid \Xi \ltimes u$ and $\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$ then $\Sigma \mid \Xi \vdash u \equiv v$

Theorem If $\Sigma \mid \Xi \ltimes u$ and $\Sigma \mid \Xi \ltimes v$ then conversion and joinability are equivalent for u and v

From joinability to conversion (for real this time)

$$\Sigma \mid \Xi \ltimes t$$

instances that appear in t
respect the equations of the interface they correspond to

$$\frac{(\texttt{def}\ f\langle\ \Xi'\ \rangle : A := t) \in \Sigma \qquad \Sigma \mid \Xi \vdash \xi \vDash \Xi' \qquad \forall x.\ \Sigma \mid \Xi \ltimes \xi(x)}{\Sigma \mid \Xi \ltimes f\langle\xi\rangle \equiv t\xi}$$

Lemma If $\Sigma \mid \Xi \mid \Gamma \vdash u : T$ then $\Sigma \mid \Xi \ltimes u$

Lemma If $\Sigma \mid \Xi \ltimes u$ and $\Sigma \mid \Xi \vdash u \rightsquigarrow v$ then $\Sigma \mid \Xi \ltimes v$

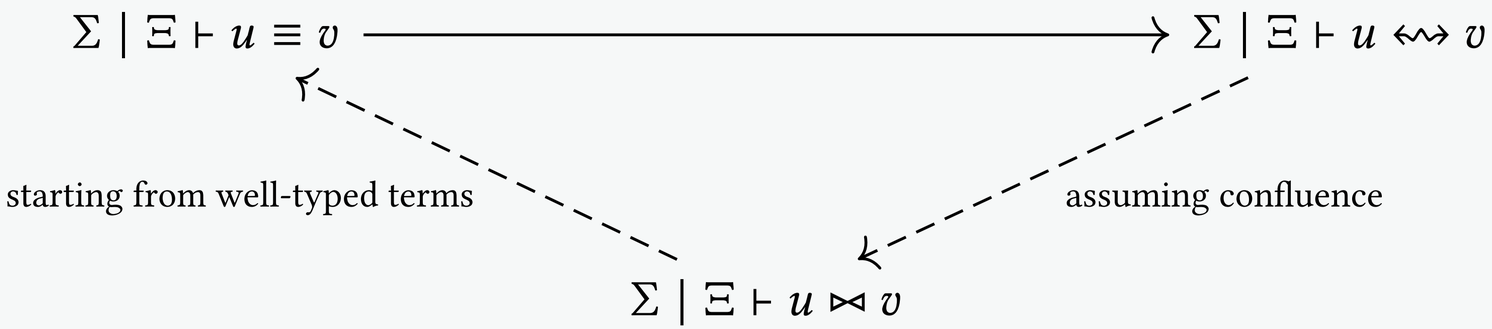
Lemma If $\Sigma \mid \Xi \ltimes u$ and $\Sigma \mid \Xi \vdash u \rightsquigarrow^* v$ then $\Sigma \mid \Xi \vdash u \equiv v$

assuming rules preserve $\ltimes$

Theorem If $\Sigma \mid \Xi \ltimes u$ and $\Sigma \mid \Xi \ltimes v$ then conversion and joinability are equivalent for u and v

Checking conversion

Characterising conversion with reduction



Injectivity of Π

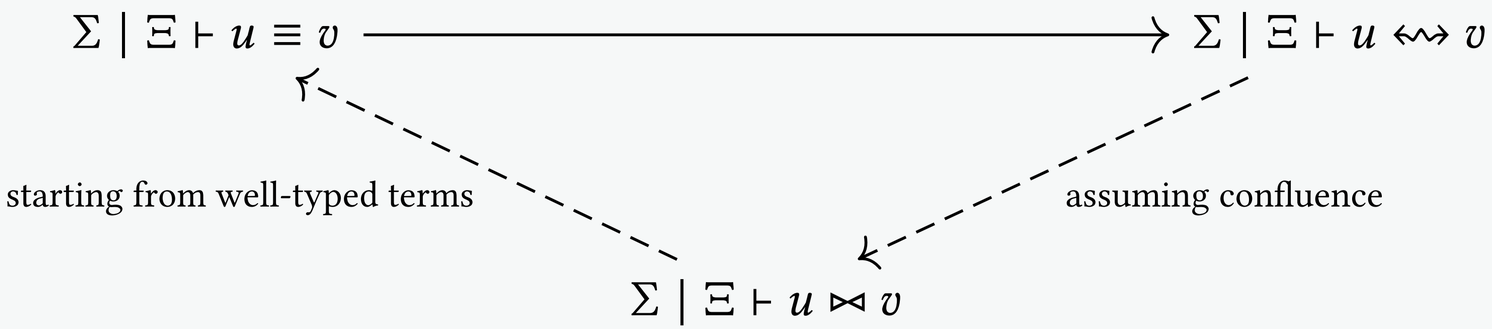
If `Σ | Ξ | Γ ⊢ Π (x : A). B ≡ Π (x : C). D` then
`Σ | Ξ | Γ ⊢ A ≡ C` and `Σ | Ξ | Γ, x : A ⊢ B ≡ D`

Subject reduction

If `Σ | Ξ | Γ ⊢ u : T` and `Σ | Ξ ⊢ u ⇝ v` then `Σ | Ξ | Γ ⊢ v : T`

Checking conversion

Characterising conversion with reduction



Injectivity of Π

If `Σ | Ξ | Γ ⊢ Π (x : A). B ≡ Π (x : C). D` then
`Σ | Ξ | Γ ⊢ A ≡ C` and `Σ | Ξ | Γ, x : A ⊢ B ≡ D`

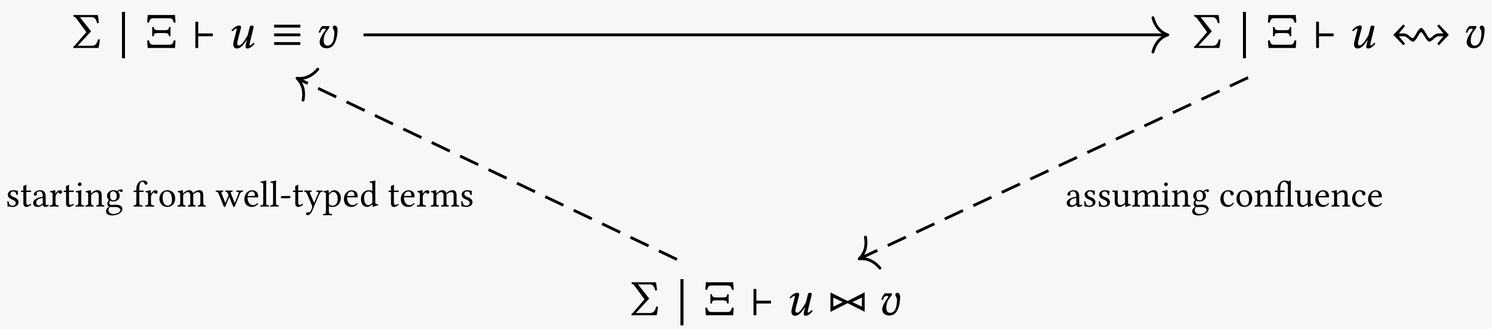
Subject reduction

If `Σ | Ξ | Γ ⊢ u : T` and `Σ | Ξ ⊢ u ⇝ v` then `Σ | Ξ | Γ ⊢ v : T`

assuming rules preserve $\bowtie$ and typing + confluence

Checking conversion

Characterising conversion with reduction



Injectivity of Π

If `Σ | Ξ | Γ ⊢ Π (x : A). B ≡ Π (x : C). D` then
`Σ | Ξ | Γ ⊢ A ≡ C` and `Σ | Ξ | Γ, x : A ⊢ B ≡ D`

Subject reduction

If `Σ | Ξ | Γ ⊢ u : T` and `Σ | Ξ ⊢ u ⇝ v` then `Σ | Ξ | Γ ⊢ v : T`

assuming rules preserve $\bowtie$ and typing + confluence

it holds with no rules at top-level (so you can wreak havoc locally without creating trouble globally)

How to check these conditions

Modular confluence criterion

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

Cockx, Tabareau, Winterhalter

2021

How to check these conditions

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

How to check these conditions

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!

How to check these conditions

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

How to check these conditions

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 rewriting community!

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.
```

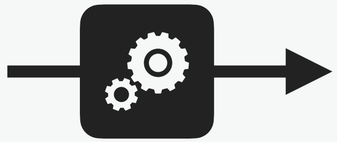
```
Module UsePlus (P : Plus).
  Import P.

  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**

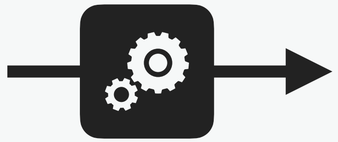
Conclusion



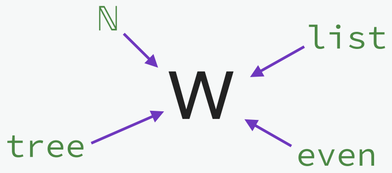
hide implementation details

Conservative extension of MLTT with **local computation**

Conclusion



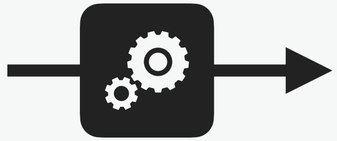
hide implementation details



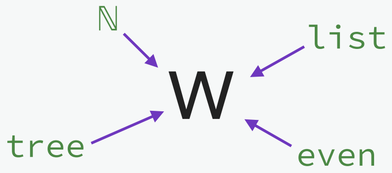
encode features

Conservative extension of MLTT with **local computation**

Conclusion



hide implementation details

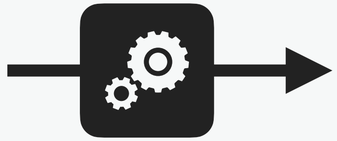


encode features

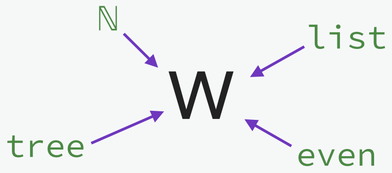
contained extensions (safer)

Conservative extension of MLTT with **local computation**

Conclusion



hide implementation details



encode features

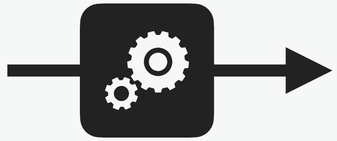
contained extensions (safer)

Conservative extension of MLTT with **local computation**

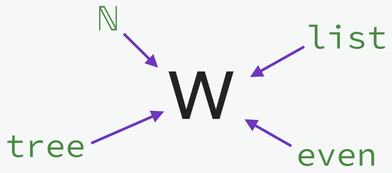
ROCQ APPROVED

/TheoWinterhalter/local-comp

Conclusion



hide implementation details



encode features

contained extensions (safer)

Conservative extension of MLTT with **local computation**

ROCQ APPROVED

/TheoWinterhalter/local-comp

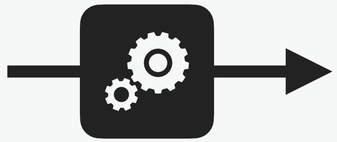
Perspectives (ongoing)

ROCQ

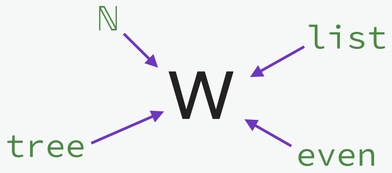
Concrete implementations

Agda

Conclusion



hide implementation details



encode features

contained extensions (safer)

Conservative extension of MLTT with **local computation**

ROCQ APPROVED

/TheoWinterhalter/local-comp

Perspectives (ongoing)

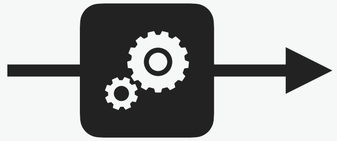
ROCQ

Concrete implementations

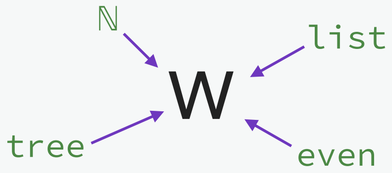
Agda

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

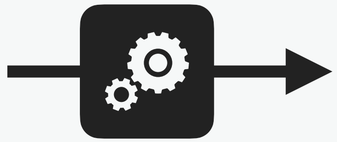
Concrete implementations

Agda

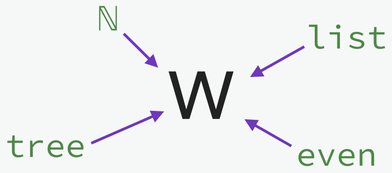
Propositional instances

Local sorts / sort polymorphism

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

Local sorts / sort polymorphism

Thank you!