EuroProofNet WG6 meeting 2025

Controlling computation in type theory, *locally*

Théo Winterhalter

INRIA Saclay

Computation in type theory ✨

Proofs by computation

```
refl : 2 + 2 = 4
```

Computation in type theory ✨

Proofs by computation

```
refl : 2 + 2 = 4
```

Proving equalities in a commutative ring done right in Coq

Grégoire, Mahboubi

2005

Computation in type theory ✨

Proofs by computation

```
refl : 2 + 2 = 4
```

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

Computation in type theory ✨

Proofs by computation

```
refl : 2 + 2 = 4
```

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

More equalities on the nose

Observational equality, now!
Altenkirch, McBride, Swierstra
2007

Computation in type theory ✨

Proofs by computation

```
refl : 2 + 2 = 4
```

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

More equalities on the nose

Observational equality, now!
Altenkirch, McBride, Swierstra
2007

LEAN

$$\frac{A : \mathtt{Prop} \qquad u, v : A}{u \equiv v}$$

Computation in type theory ✨

Proofs by computation

```
refl : 2 + 2 = 4
```

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

More equalities on the nose

Observational equality, now!
Altenkirch, McBride, Swierstra
2007

LEAN

$$\frac{A : \mathsf{Prop} \qquad u, v : A}{u \equiv v}$$

Definitional proof irrelevance without K
Gilbert, Cockx, Tabareau
2019
Agda ROCQ

and more!

Controlling and extending computation in ITPs

Coq modulo theory

Strub

2010

Controlling and extending computation in ITPs

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

Controlling and extending computation in ITPs

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

Controlling and extending computation in ITPs

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

Controlling unfolding in type theory
Gratzer, Sterling, Angiuli, Coquand, Birkedal
2022

Agda

cooltt

Controlling and extending computation in ITPs

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

Controlling unfolding in type theory
Gratzer, Sterling, Angiuli, Coquand, Birkedal
2022

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

Controlling and extending computation in ITPs

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

Agda

This one is local! 😎

Controlling unfolding in type theory
Gratzer, Sterling, Angiuli, Coquand, Birkedal
2022

😎 cooltt

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

ROCQ

Why locality matters

Example: exceptions

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

Agda · ROCQ

Computation rules must be assumed forever

```
lemma something_unrelated : …
```

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
defn 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 t f true ↦ t
rule ifte t f false ↦ t
```

Rules extend the trusted computing base

Why locality matters

Example: exceptions

```
symb raise : ∀ {A}, A
rule if raise then t else f ↦ raise
defn 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 t f true ↦ t
rule ifte t f false ↦ 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
defn 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 t f true ↦ t
rule ifte t f false ↦ t
```

Rules extend the trusted computing base

Uncaught mistake without a model
(somewhat mitigated in Agda)

Overall, not very modular
(or type-theoretic)

The elephant in the room: why not use extensional type theory?

Equality reflection

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

The elephant in the room: why not use extensional type theory?

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

The elephant in the room: why not use extensional type theory?

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

The elephant in the room:
why not use extensional type theory?

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!

My proposal

Prenex quantification over (directed) equations

```
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
```

My proposal

Prenex quantification over (directed) equations

```
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
```

My proposal

Prenex quantification over (directed) equations

```
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
```

My proposal

Prenex quantification over (directed) equations

```
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 → `ifte P t f := S.elim P (U.elim _ t) (U.elim _ f)`

My proposal

Prenex quantification over (directed) equations

```
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 (→ `ifte P t f := …`)

My proposal

Prenex quantification over (directed) equations

```
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
```

```
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 negb⟨ B : Bool ⟩ (b : B.bool) : B.bool :=
  B.ifte (λ _. B.bool) B.false B.true 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 (→ `ifte P t f := ...`)

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

Other examples include…

Hiding implementation details
while retaining computation

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

```
instance ShiftasMap
  shift := map suc
```

This way, `map` never appears in goals out of nowhere
useful for automatically generated functions (eg. Equations in Rocq)
for controlling unfolding (like in `cooltt`)
or for strictifications

Other examples include...

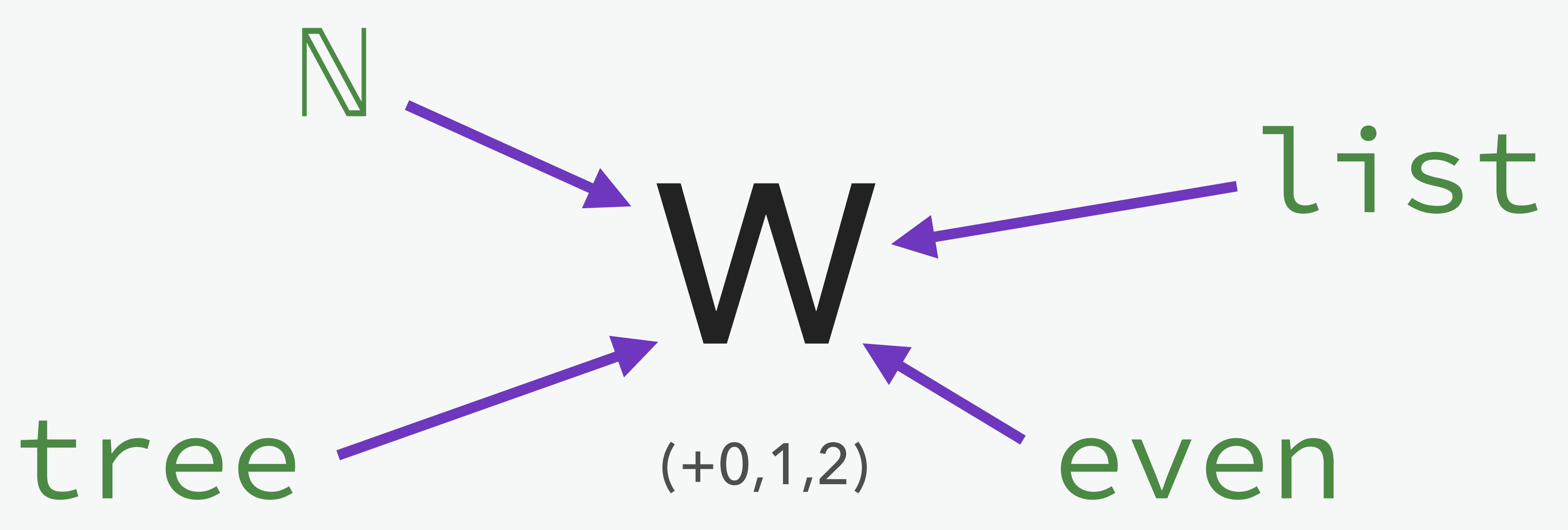
Hiding implementation details
while retaining computation

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

```
instance ShiftasMap
  shift := map suc
```

This way, `map` never appears in goals out of nowhere
useful for automatically generated functions (eg. Equations in Rocq)
for controlling unfolding (like in `cooltt`)
or for strictifications

Encode features using simpler ones



Why not W?
Hugunin
2021

Other examples include…

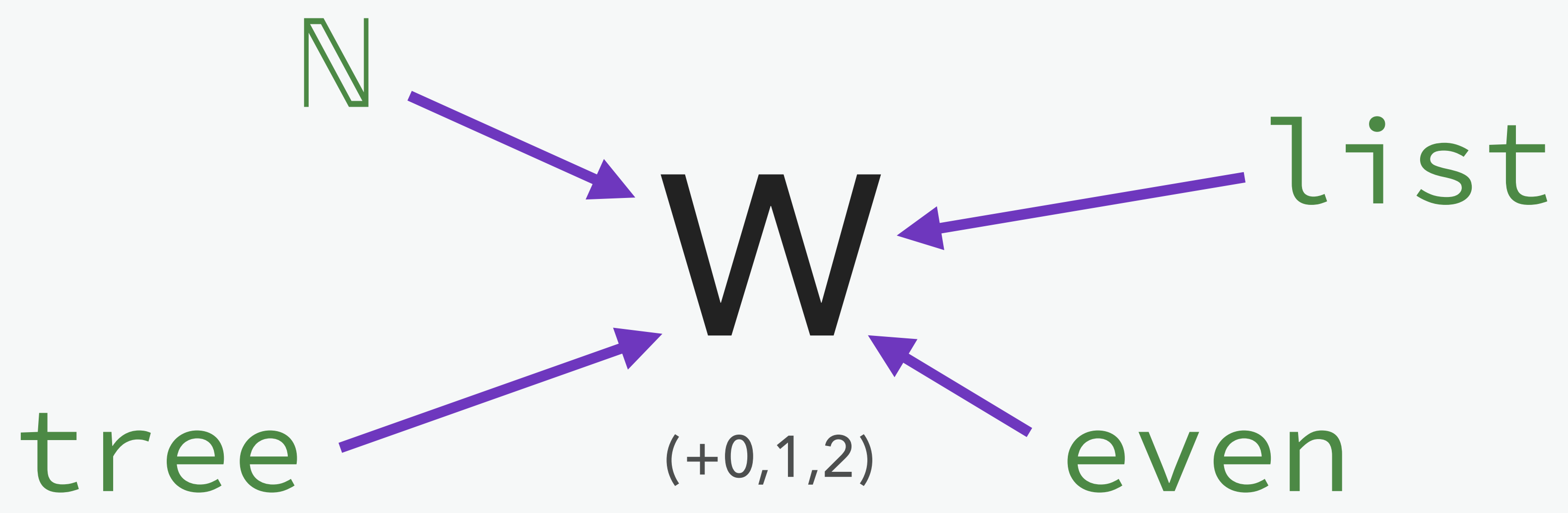
Hiding implementation details
while retaining computation

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

```
instance ShiftasMap
  shift := map suc
```

This way, map never appears in goals out of nowhere
useful for automatically generated functions (eg. Equations in Rocq)
for controlling unfolding (like in `cooltt`)
or for strictifications

Encode features using simpler ones



Why not W?
Hugunin
2021

Use effects locally, eg. exceptions

The type theory

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

The type theory

```
interface E⟨ Ξ’ ⟩ assumes Δ where R
```

```
def f⟨ Ξ’ ⟩ : A := t
```

Global environment

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

The type theory

```
interface E⟨ Ξ’ ⟩ assumes Δ where R
```

```
def f⟨ Ξ’ ⟩ : A := t
```

Global environment

reference in Σ

name

instance

```
M : E⟨ ξ ⟩
```

Extension environment

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

The type theory

```
interface E⟨ Ξ’ ⟩ assumes Δ where R
```

```
def f⟨ Ξ’ ⟩ : A := t
```

Global environment

reference in Σ

name

instance

```
M : E⟨ ξ ⟩
```

Extension environment

Basically regular MLTT

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

The type theory

```
interface E⟨ Ξ’ ⟩ assumes Δ where R
```

```
def f⟨ Ξ’ ⟩ : A := t
```

reference in Σ

name — instance

```
M : E⟨ ξ ⟩
```

Global environment — Extension environment — Basically regular MLTT

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

Computation rule (simplified)

```
(interface E⟨ Ξ’ ⟩ assumes Δ where R) ∈ Σ
  (M : E⟨ ξ ⟩) ∈ Ξ              (l ↦ r) ∈ R
─────────────────────────────────────────────
          Σ | Ξ | Γ ⊢ lξσ ≡ rξσ
```

The type theory

```
interface E⟨ Ξ’ ⟩ assumes Δ where R
```

```
def f⟨ Ξ’ ⟩ : A := t
```

reference in Σ

name — instance

```
M : E⟨ ξ ⟩
```

Global environment — Extension environment — Basically regular MLTT

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

Computation rule (simplified)

$$\frac{(\texttt{interface } E\langle\ \Xi'\ \rangle \texttt{ assumes } \Delta \texttt{ where } R) \in \Sigma \qquad (M : E\langle\ \xi\ \rangle) \in \Xi \qquad (l \mapsto r) \in R}{\Sigma \mid \Xi \mid \Gamma \vdash l\xi\sigma \equiv r\xi\sigma}$$

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 ETT

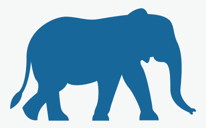
Meta-theory

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 ETT 

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

where κ interprets the definitions of Σ

Inlining

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

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

where κ interprets the definitions of Σ

removes all definitions
and unfolds them in extensions

Inlining

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

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

removes all definitions
and unfolds them in extensions

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

where κ interprets the definitions of Σ

removes all definitions
and unfolds them in extensions

with κ fixed (and abstract):

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

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

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

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

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

κ is then defined by induction on ⊢ Σ such that

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

Inlining

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

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

removes all definitions
and unfolds them in extensions

where κ interprets the definitions of Σ

with κ fixed (and abstract):

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

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

κ is then defined by induction on ⊢ Σ 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 $[\![\Sigma]\!] \mid [\![\Xi]\!]\langle \kappa \rangle \mid [\![\Gamma]\!]\langle \kappa \rangle \vdash [\![t]\!]\langle \kappa \rangle : [\![A]\!]\langle \kappa \rangle$

where κ interprets the definitions of Σ

removes all definitions and unfolds them in extensions

with κ fixed (and abstract):

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

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

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

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

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

κ is then defined by induction on ⊢ Σ such that

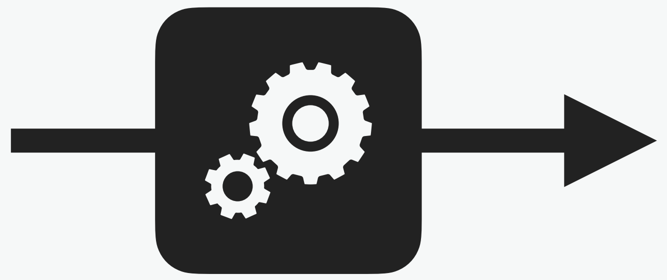
recursive call ok because t lives in an environment *smaller* than Σ

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

Conclusion

Conservative extension of MLTT with **local computation**

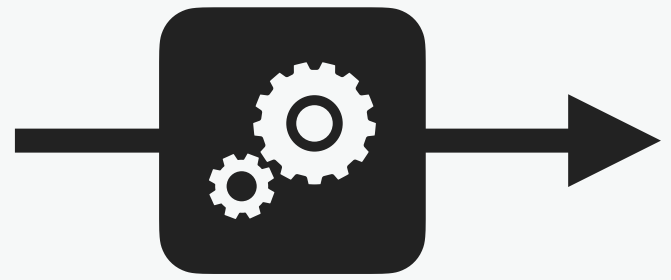
Conclusion



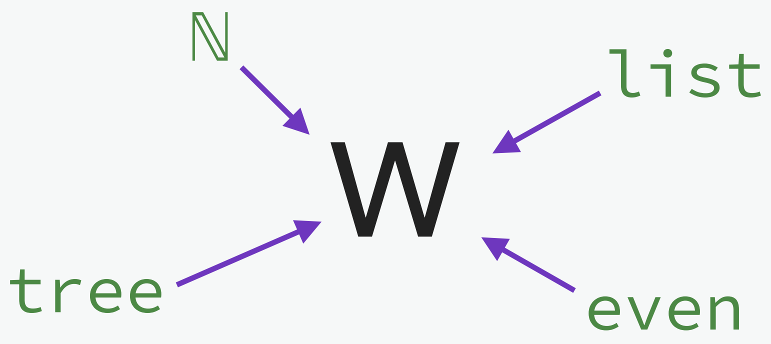
hide implementation details

Conservative extension of MLTT with **local computation**

Conclusion



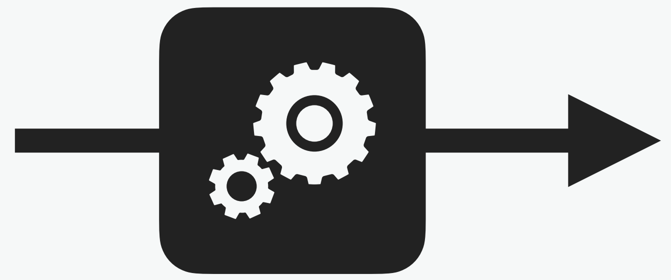
hide implementation details



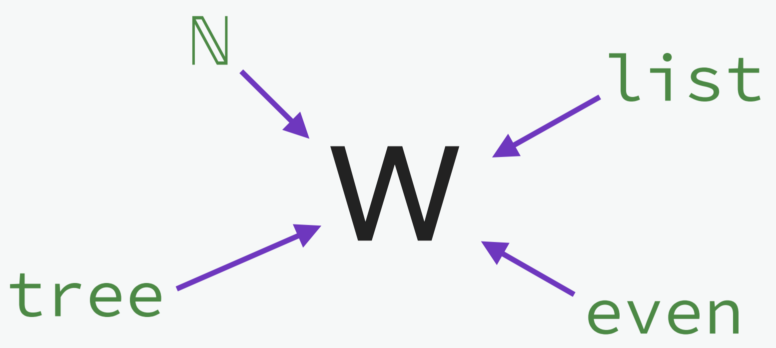
encode features

Conservative extension of MLTT with **local computation**

Conclusion



hide implementation details

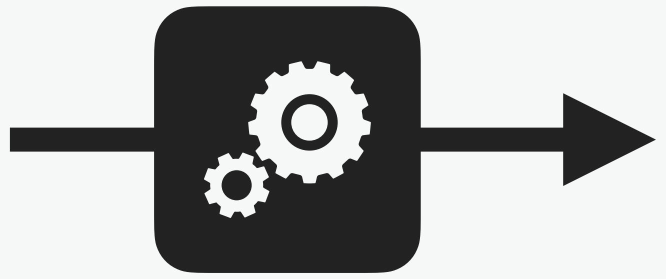


encode features

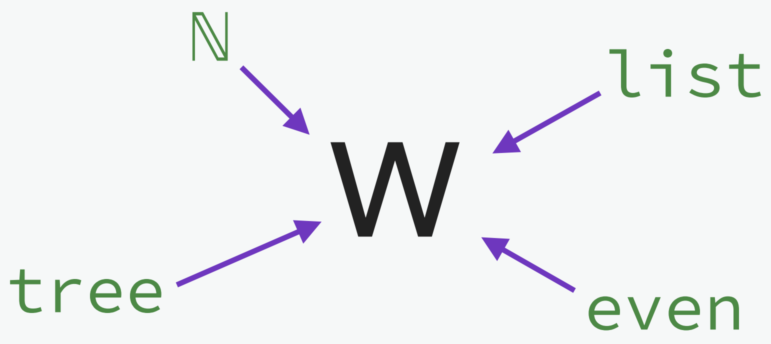
contained extensions (safer)

Conservative extension of MLTT with **local computation**

Conclusion



hide implementation details



encode features

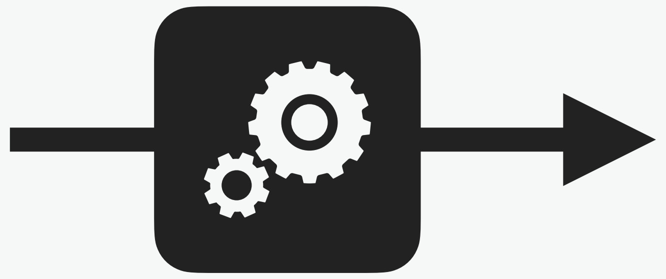
contained extensions (safer)

Conservative extension of MLTT with **local computation**

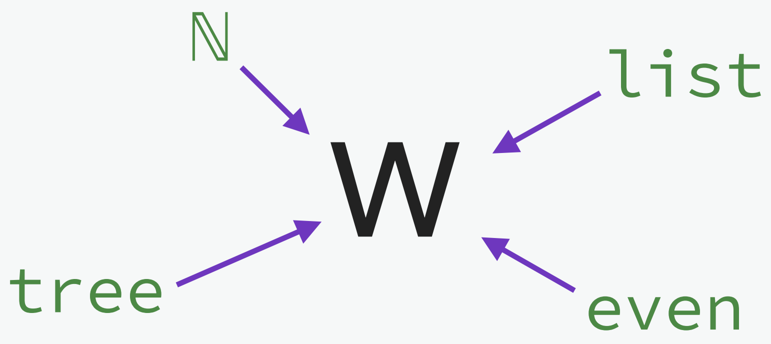
ROCQ APPROVED

/TheoWinterhalter/local-comp

Conclusion



hide implementation details



encode features

contained extensions (safer)

Conservative extension of MLTT with **local computation**

ROCQ APPROVED

/TheoWinterhalter/local-comp

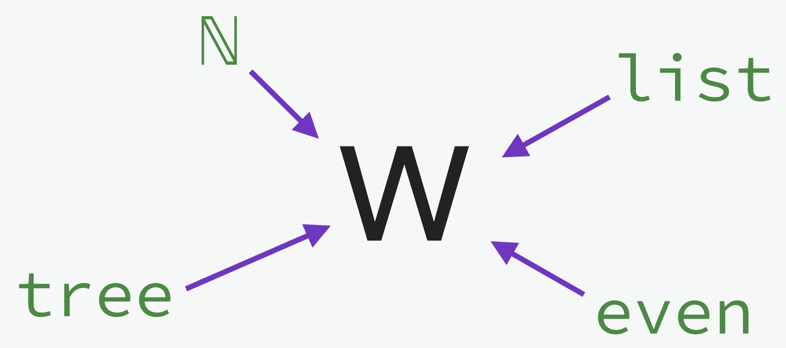
Perspectives

Agda ROCQ

Concrete implementation

Conclusion

hide implementation details



encode features

contained extensions (safer)

Conservative extension of MLTT with **local computation**

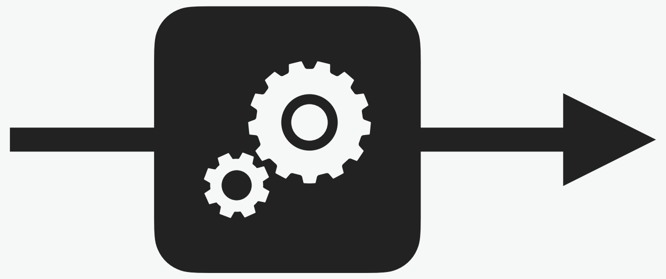
/TheoWinterhalter/local-comp

Perspectives

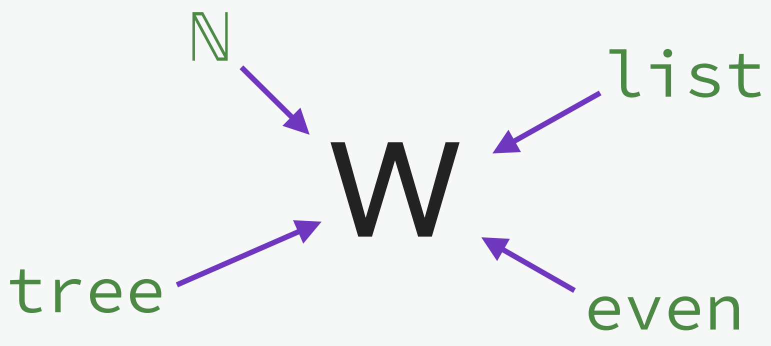
Concrete implementation

Decidability of type checking

Conclusion



hide implementation details



encode features

contained extensions (safer)

Conservative extension of MLTT with **local computation**

ROCQ APPROVED

/TheoWinterhalter/local-comp

Perspectives

Agda ROCQ

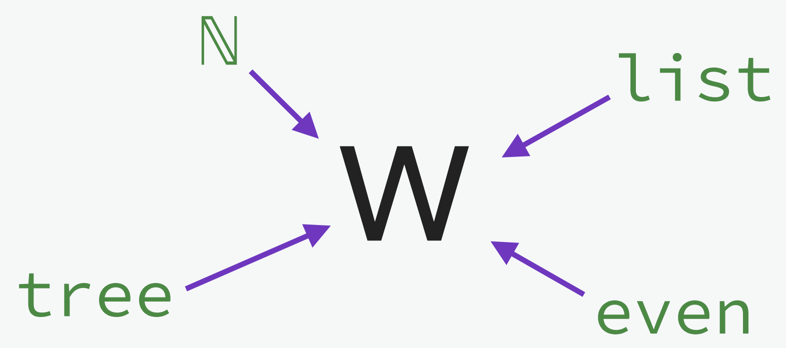
Concrete implementation

Decidability of type checking

Propositional instances

Conclusion

hide implementation details



encode features

contained extensions (safer)

Conservative extension of MLTT with **local computation**

ROCQ APPROVED

/TheoWinterhalter/local-comp

Perspectives

Agda ROCQ

Concrete implementation

Decidability of type checking

Propositional instances

Thank you!