EuroProofNet WG1+2+4 meeting

Dependent Ghosts have a reflection for free

Théo Winterhalter

What's up with vectors?

```
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons (a : A) n (v : vec A n) : vec A (S n)
```

type-based invariant

What’s up with vectors?

```
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons (a : A) n (v : vec A n) : vec A (S n)
```

type-based invariant

```
rev : ∀ n m. vec A n → vec A m → vec A (n + m)
rev 0 m vnil acc := acc
rev (S k) m (vcons a k v) acc := rev k (S m) v (vcons a m acc)
```

What's up with vectors?

```
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons (a : A) n (v : vec A n) : vec A (S n)
```

type-based invariant

```
rev : ∀ n m. vec A n → vec A m → vec A (n + m)
rev 0 m vnil acc := acc
rev (S k) m (vcons a k v) acc := rev k (S m) v (vcons a m acc)
```

actually a type mismatch!

`vec A (S k + m)` vs `vec A (k + S m)`

but we really wish they would be equal…

What's up with vectors?

```
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons (a : A) n (v : vec A n) : vec A (S n)
```

type-based invariant

```
rev : ∀ n m. vec A n → vec A m → vec A (n + m)
rev 0 m vnil acc := acc
rev (S k) m (vcons a k v) acc := rev k (S m) v (vcons a m acc)
```

actually a type mismatch!

`vec A (S k + m)` vs `vec A (k + S m)`

but we really wish they would be equal…

What's up with vectors?

```
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons (a : A) n (v : vec A n) : vec A (S n)
```

```
rev : ∀ n m. vec A n → vec A m → vec A (n + m)
rev 0 m vnil acc := acc
rev (S k) m (vcons a k v) acc := rev k (S m) v (vcons a m acc)
```

```
type ‘a vec =
| Vnil
| Vcons of ‘a * nat * ‘a vec
```

What's up with vectors?

```
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons (a : A) n (v : vec A n) : vec A (S n)
```

```
rev : ∀ n m. vec A n → vec A m → vec A (n + m)
rev 0 m vnil acc := acc
rev (S k) m (vcons a k v) acc := rev k (S m) v (vcons a m acc)
```

```
type ‘a vec =
| Vnil
| Vcons of ‘a * nat * ‘a vec
```

```
val rev : nat → nat → ‘a vec → ‘a vec → ‘a vec
let rev _ m v acc =
  match v with
  | Vnil → acc
  | Vcons (a,k,w) → Obj.magic (rev k (S m) w (Vcons (a,m,acc))
```

What's up with vectors?

```
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons (a : A) n (v : vec A n) : vec A (S n)
```

```
rev : ∀ n m. vec A n → vec A m → vec A (n + m)
rev 0 m vnil acc := acc
rev (S k) m (vcons a k v) acc := rev k (S m) v (vcons a m acc)
```

```
type ‘a vec =
| Vnil
| Vcons of ‘a * nat * ‘a vec
```

we should have lists!

```
val rev : nat → nat → ‘a vec → ‘a vec → ‘a vec
let rev _ m v acc =
  match v with
  | Vnil → acc
  | Vcons (a,k,w) → Obj.magic (rev k (S m) w (Vcons (a,m,acc))
```

What's up with vectors?

```
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons (a : A) n (v : vec A n) : vec A (S n)
```

```
rev : ∀ n m. vec A n → vec A m → vec A (n + m)
rev 0 m vnil acc := acc
rev (S k) m (vcons a k v) acc := rev k (S m) v (vcons a m acc)
```

```
type ‘a vec =
| Vnil
| Vcons of ‘a * nat * ‘a vec
```

we should have lists!

```
val rev : nat → nat → ‘a vec → ‘a vec → ‘a vec
let rev _ m v acc =
  match v with
  | Vnil → acc
  | Vcons (a,k,w) → Obj.magic (rev k (S m) w (Vcons (a,m,acc))
```

The problem is always in the n of `vec A n`...

Using ghost types…

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

we mark the index as erased

```
type ‘a vec =
| Vnil
| Vcons of ‘a * ‘a vec
```

erased is removed at extraction

Using ghost types…

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

we mark the index as erased

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

```
type 'a vec =
| Vnil
| Vcons of 'a * 'a vec
```

erased is removed at extraction

Using ghost types…

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

we mark the index as erased

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

Eliminator `reveal` cannot land in `Type`
only in `Ghost` and `Prop`

```
type 'a vec =
| Vnil
| Vcons of 'a * 'a vec
```

erased is removed at extraction

Using ghost types...

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

we mark the index as erased

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

```
type 'a vec =
| Vnil
| Vcons of 'a * 'a vec
```

Eliminator `reveal` cannot land in `Type`
only in `Ghost` and `Prop`

erased is removed at extraction

```
gS : erased ℕ → erased ℕ
gS n := reveal n as x in hide (S x)
```

Using ghost types…

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

we mark the index as erased

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

```
type ‘a vec =
| Vnil
| Vcons of ‘a * ‘a vec
```

Eliminator `reveal` cannot land in `Type`
only in `Ghost` and `Prop`

erased is removed at extraction

```
gS : erased ℕ → erased ℕ
gS n := reveal n as x in hide (S x)
```

but…

```
erased bool → bool
```

only contains constant functions

…and ghost reflection

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

we mark the index as erased

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

```
A : Ghost     u, v : A     e : u = v
────────────────────────────────────
               u ≡ v
```

Eliminator reveal cannot land in Type
only in Ghost and Prop

Propositionally equal inhabitants of ghosts
are definitionally equal

…and ghost reflection

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

we mark the index as erased

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

```
A : Ghost      u, v : A      e : u = v
──────────────────────────────────────
                u ≡ v
```

Eliminator reveal cannot land in Type
only in Ghost and Prop

Propositionally equal inhabitants of ghosts
are definitionally equal

```
rev : ∀ {n m}. vec A n → vec A m → vec A (n +' m)
rev vnil acc := acc
rev (vcons a k v) acc := rev v (vcons a m acc)
```

...and ghost reflection

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

we mark the index as erased

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

```
A : Ghost     u, v : A     e : u = v
─────────────────────────────────────
               u ≡ v
```

Eliminator `reveal` cannot land in `Type` only in `Ghost` and `Prop`

Propositionally equal inhabitants of ghosts are definitionally equal

```
rev : ∀ {n m}. vec A n → vec A m → vec A (n +’ m)
rev vnil acc := acc
rev (vcons a k v) acc := rev v (vcons a m acc)
```

ok because `vec A (gS k +’ m) ≡ vec A (k +’ gS m)`

Wait, couldn't this just be (S)Prop?

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

Eliminator `reveal` cannot go to `Type`
only to `Ghost` and `Prop`

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Propositionally equal inhabitants of ghosts
are definitionally equal

Wait, couldn't this just be (S)Prop?

```
Inductive squash (A : Type) : Prop :=
| sq (a : A) : squash A
```

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Eliminator `reveal` cannot go to `Type`
only to `Ghost` and `Prop`

Propositionally equal inhabitants of ghosts
are definitionally equal

Wait, couldn't this just be (S)Prop?

```
Inductive squash (A : Type) : Prop :=
| sq (a : A) : squash A
```

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

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Eliminator `reveal` cannot go to `Type`
only to `Ghost` and `Prop`

Propositionally equal inhabitants of ghosts
are definitionally equal

Wait, couldn't this just be (S)Prop?

```
Inductive squash (A : Type) : Prop :=
| sq (a : A) : squash A
```

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

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

$$\frac{A : \mathrm{Ghost} \qquad u,\ v : A \qquad e : u = v}{u \equiv v}$$

Eliminator `reveal` cannot go to `Type`
only to `Ghost` and `Prop`

Propositionally equal inhabitants of ghosts
are definitionally equal

Not if we want to distinguish the two types

`vec A (hide 0)` and `vec A (gS n)` *(eg to build head and `tail` functions)*

Wait, couldn't this just be (S)Prop?

```
Inductive squash (A : Type) : Prop :=
| sq (a : A) : squash A
```

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

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

$$\frac{A : \mathtt{Ghost} \qquad u,\ v : A \qquad e : u = v}{u \equiv v}$$

Eliminator `reveal` cannot go to `Type`
only to `Ghost` and `Prop`

Propositionally equal inhabitants of ghosts
are definitionally equal

Not if we want to distinguish the two types

`vec A (hide 0)` and `vec A (gS n)` *(eg* to build head and `tail` functions)

and thus `hide 0` and `gS n`

Wait, couldn't this just be (S)Prop?

```
Inductive squash (A : Type) : Prop :=
| sq (a : A) : squash A
```

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

```
Inductive erased (A : Type) : Ghost :=
| hide (a : A) : erased A
```

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Eliminator `reveal` cannot go to `Type`
only to `Ghost` and `Prop`

Propositionally equal inhabitants of ghosts
are definitionally equal

this is a problem though!

Not if we want to distinguish the two types

`vec A (hide 0)` and `vec A (gS n)` *(eg* to build head and `tail` functions)

and thus `hide 0` and `gS n`

Reveal proposition

$$\frac{\texttt{e : erased A} \qquad \texttt{f : A → Prop}}{\texttt{Reveal e f : Prop}}$$

```
Reveal (hide t) f ↔ f t
```

Reveal proposition

$$\frac{e : \mathsf{erased}\ A \qquad f : A \to \mathsf{Prop}}{\mathsf{Reveal}\ e\ f : \mathsf{Prop}}$$

```
Reveal (hide t) f ↔ f t
```

We get a discriminator: `D (hide 0) ↔ ⊤` `D (gS n) ↔ ⊥`

```
D : erased ℕ → Prop
D n := Reveal n (λx. match x with 0 => ⊤ | _ => ⊥ end)
```

How do we justify this?

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Ghost reflection

How do we justify this?

$$\frac{A : \texttt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Ghost reflection

↓

$$\frac{A : \texttt{Ghost} \qquad e : u =_A v \qquad t : P\ u}{\texttt{cast}\ e\ P\ t : P\ v}$$

Ghost casts

How do we justify this?

$$\frac{\texttt{A : Ghost} \qquad \texttt{u, v : A} \qquad \texttt{e : u = v}}{\texttt{u} \equiv \texttt{v}}$$

Ghost reflection

translating *derivations*
replacing conversion rule by casts
like for ETT to ITT [Oury 2005 ; WST 2019]

$$\frac{\texttt{A : Ghost} \qquad \texttt{e : u} =_{\texttt{A}} \texttt{v} \qquad \texttt{t : P u}}{\texttt{cast e P t : P v}}$$

Ghost casts

How do we justify this?

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Ghost reflection

translating *derivations*
replacing conversion rule by casts
like for ETT to ITT [Oury 2005 ; WST 2019]

$$\frac{A : \mathtt{Ghost} \qquad e : u =_A v \qquad t : P\ u}{\mathtt{cast}\ e\ P\ t : P\ v}$$

Ghost casts

$$\mathtt{cast}\ e\ P\ t \equiv t$$

ignored for conversion

How do we justify this?

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Ghost reflection

translating *derivations*
replacing conversion rule by casts
like for ETT to ITT [Oury 2005 ; WST 2019]

How do we justify this?

$$\frac{A : \mathtt{Ghost} \qquad e : u =_A v \qquad t : P\ u}{\mathtt{cast}\ e\ P\ t : P\ v}$$

Ghost casts

$$\mathtt{cast}\ e\ P\ t \equiv t$$

ignored for conversion

How do we justify this?

$$\frac{A : \mathtt{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Ghost reflection

translating *derivations*
replacing conversion rule by casts
like for ETT to ITT [Oury 2005 ; WST 2019]

How do we justify this?

$$\frac{A : \mathtt{Ghost} \qquad e : u =_A v \qquad t : P\ u}{\mathtt{cast}\ e\ P\ t : P\ v}$$

Ghost casts

$$\mathtt{cast}\ e\ P\ t \equiv t$$

ignored for conversion

MLTT/CIC with (S)Prop

How do we justify this?

$$\frac{A : \mathsf{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Ghost reflection

translating *derivations*
replacing conversion rule by casts
like for ETT to ITT [Oury 2005 ; WST 2019]

How do we justify this?

$$\frac{A : \mathsf{Ghost} \qquad e : u =_A v \qquad t : P\ u}{\mathsf{cast}\ e\ P\ t : P\ v}$$

Ghost casts

$$\mathsf{cast}\ e\ P\ t \equiv t$$

ignored for conversion

parametricity translation
taking inspiration from exceptional type theory
[Pédrot Tabareau 2018]

MLTT/CIC with (S)Prop

How do we justify this?

$$\frac{A : \mathsf{Ghost} \qquad u, v : A \qquad e : u = v}{u \equiv v}$$

Ghost reflection

translating derivations
replacing conversion rule by casts
like for ETT to ITT [Oury 2005 ; WST 2019]

GTT

How do we justify this?

$$\frac{A : \mathsf{Ghost} \qquad e : u =_A v \qquad t : P\ u}{\mathsf{cast}\ e\ P\ t : P\ v}$$

Ghost casts

$$\mathsf{cast}\ e\ P\ t \equiv t$$

ignored for conversion

interesting on its own!

parametricity translation
taking inspiration from exceptional type theory
[Pédrot Tabareau 2018]

MLTT/CIC with (S)Prop

Erasure

Translation getting rid of all ghosts and proofs

$$[\lambda(x^{\mathbb{T}} : A).\ t]_\varepsilon := \lambda(x : [\![A]\!]_\varepsilon).\ [t]_\varepsilon$$

$$[\lambda(x^{\mathbb{G}} : A).\ t]_\varepsilon := [t]_\varepsilon$$

Erasure

Translation getting rid of all ghosts and proofs

$[\lambda(x^{\mathbb{T}} : A).\ t]_\varepsilon := \lambda(x : [\![A]\!]_\varepsilon).\ [t]_\varepsilon$

$[f^{\mathbb{T}}\ u^{\mathbb{T}}]_\varepsilon := [f]_\varepsilon\ [u]_\varepsilon$

$[\lambda(x^{\mathbb{G}} : A).\ t]_\varepsilon := [t]_\varepsilon$

$[f^{\mathbb{T}}\ u^{\mathbb{G}}]_\varepsilon := [f]_\varepsilon$

Erasure

Translation getting rid of all ghosts and proofs

$$[\lambda(x^{\mathbb{T}} : A).\ t]_\varepsilon := \lambda(x : [\![A]\!]_\varepsilon).\ [t]_\varepsilon$$

$$[\lambda(x^{\mathbb{G}} : A).\ t]_\varepsilon := [t]_\varepsilon$$

$$[f^{\mathbb{T}}\ u^{\mathbb{T}}]_\varepsilon := [f]_\varepsilon\ [u]_\varepsilon$$

$$[f^{\mathbb{T}}\ u^{\mathbb{G}}]_\varepsilon := [f]_\varepsilon$$

$$[\mathtt{cast}\ e\ P\ t]_\varepsilon := [t]_\varepsilon$$

Erasure

Translation getting rid of all ghosts and proofs

$[\lambda(x^{\mathbb{T}} : A).\ t]_\varepsilon := \lambda(x : [\![A]\!]_\varepsilon).\ [t]_\varepsilon$

$[\lambda(x^{\mathbb{G}} : A).\ t]_\varepsilon := [t]_\varepsilon$

$[f^{\mathbb{T}}\ u^{\mathbb{T}}]_\varepsilon := [f]_\varepsilon\ [u]_\varepsilon$

$[f^{\mathbb{T}}\ u^{\mathbb{G}}]_\varepsilon := [f]_\varepsilon$

$[\mathsf{cast}\ e\ P\ t]_\varepsilon := [t]_\varepsilon$

$\mathsf{exfalso}^{\mathbb{T}}\ (A : \mathsf{Type})\ (p : \bot) : A$

$[\mathsf{exfalso}^{\mathbb{T}}\ A\ p]_\varepsilon := ??$

Erasure

Translation getting rid of all ghosts and proofs

$[\lambda(x^{\mathbb{T}} : A).\ t]_\varepsilon := \lambda(x : [\![A]\!]_\varepsilon).\ [t]_\varepsilon$

$[\lambda(x^{\mathbb{G}} : A).\ t]_\varepsilon := [t]_\varepsilon$

$[f^{\mathbb{T}}\ u^{\mathbb{T}}]_\varepsilon := [f]_\varepsilon\ [u]_\varepsilon$

$[f^{\mathbb{T}}\ u^{\mathbb{G}}]_\varepsilon := [f]_\varepsilon$

$[\mathsf{cast}\ e\ P\ t]_\varepsilon := [t]_\varepsilon$

$\mathsf{exfalso}^{\mathbb{T}}\ (A : \mathsf{Type})\ (p : \bot) : A$

$[\mathsf{exfalso}^{\mathbb{T}}\ A\ p]_\varepsilon := ??$

we get no $\bot$ but we need some $[\![A]\!]_\varepsilon$

Erasure

Translation getting rid of all ghosts and proofs

$[\lambda(x^{\mathbb{T}} : A).\ t]_\varepsilon := \lambda(x : [\![A]\!]_\varepsilon).\ [t]_\varepsilon$

$[\lambda(x^{\mathbb{G}} : A).\ t]_\varepsilon := [t]_\varepsilon$

$[f^{\mathbb{T}}\ u^{\mathbb{T}}]_\varepsilon := [f]_\varepsilon\ [u]_\varepsilon$

$[f^{\mathbb{T}}\ u^{\mathbb{G}}]_\varepsilon := [f]_\varepsilon$

$[\mathtt{cast}\ e\ P\ t]_\varepsilon := [t]_\varepsilon$

$\mathtt{exfalso}^{\mathbb{T}}\ (A : \mathtt{Type})\ (p : \bot) : A$

$[\mathtt{exfalso}^{\mathbb{T}}\ A\ p]_\varepsilon :=$ “$\mathtt{raise}\ [\![A]\!]_\varepsilon$”

we get no $\bot$ but we need some $[\![A]\!]_\varepsilon$

Erasure

Translation getting rid of all ghosts and proofs

$[\lambda(x^{\mathbb{T}} : A).\ t]_\varepsilon := \lambda(x : [\![A]\!]_\varepsilon).\ [t]_\varepsilon$

$[\lambda(x^{\mathbb{G}} : A).\ t]_\varepsilon := [t]_\varepsilon$

$[f^{\mathbb{T}}\ u^{\mathbb{T}}]_\varepsilon := [f]_\varepsilon\ [u]_\varepsilon$

$[f^{\mathbb{T}}\ u^{\mathbb{G}}]_\varepsilon := [f]_\varepsilon$

$[\mathrm{cast}\ e\ P\ t]_\varepsilon := [t]_\varepsilon$

$\mathrm{exfalso}^{\mathbb{T}}\ (A : \mathrm{Type})\ (p : \bot) : A$

$[\mathrm{exfalso}^{\mathbb{T}}\ A\ p]_\varepsilon := [\![A]\!]_\varnothing$

we get no $\bot$ but we need some $[\![A]\!]_\varepsilon$

$A : \mathrm{Type} \longrightarrow [\![A]\!]_\varepsilon : \mathrm{Type}$, $[\![A]\!]_\varnothing : [\![A]\!]_\varepsilon$

Example

Booleans

source

```
Inductive bool :=
| true
| false
```

Example

Booleans

source

```
Inductive bool :=
| true
| false
```

erasure

```
Inductive bool• :=
| true•
| false•
| bool∅
```

Example

Booleans

source

```
Inductive bool :=
| true
| false
```

erasure

```
Inductive bool• :=
| true•
| false•
| bool∅
```

parametricity in Prop [Keller Lasson 2012]

```
Inductive boolP : bool• → Prop :=
| trueP : boolP true•
| falseP : boolP false•
```

predicate guaranteeing no exceptions raised at top-level

Example

Booleans

source

```
Inductive bool :=
| true
| false
```

erasure

```
Inductive bool• :=
| true•
| false•
| bool∅
```

⚠ limits large elimination

parametricity in Prop [Keller Lasson 2012]

```
Inductive boolP : bool• → Prop :=
| trueP : boolP true•
| falseP : boolP false•
```

predicate guaranteeing no exceptions raised at top-level

Example

Booleans

source

```
Inductive bool :=
| true
| false
```

erasure

```
Inductive bool• :=
| true•
| false•
| bool∅
```

⚠ limits large elimination

parametricity in Prop [Keller Lasson 2012]

```
Inductive boolP : bool• → Prop :=
| trueP : boolP true•
| falseP : boolP false•
```

predicate guaranteeing no exceptions raised at top-level

Free theorem:

```
erased bool → bool
```

only contains constant functions

Example

Vectors

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

Example

Vectors

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

```
Inductive vec• (A : ty) :=
| vnil•
| vcons• (a : El A) (v : vec• A)
| vec∅
```

Example

Vectors

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

```
Inductive vec• (A : ty) :=
| vnil•
| vcons• (a : El A) (v : vec• A)
| vec∅
```

```
Inductive vecP (A : ty) (AP : El A → Prop) : ∀ n (nP : ℕP n), vec• A → Prop :=
| vnilP : vecP A AP 0• 0P vnil•
| vconsP a (aP : AP a) n nP v : vecP A AP n nP v → vecP A AP (S• n) (SP n nP) (vcons• a v)
```

Example

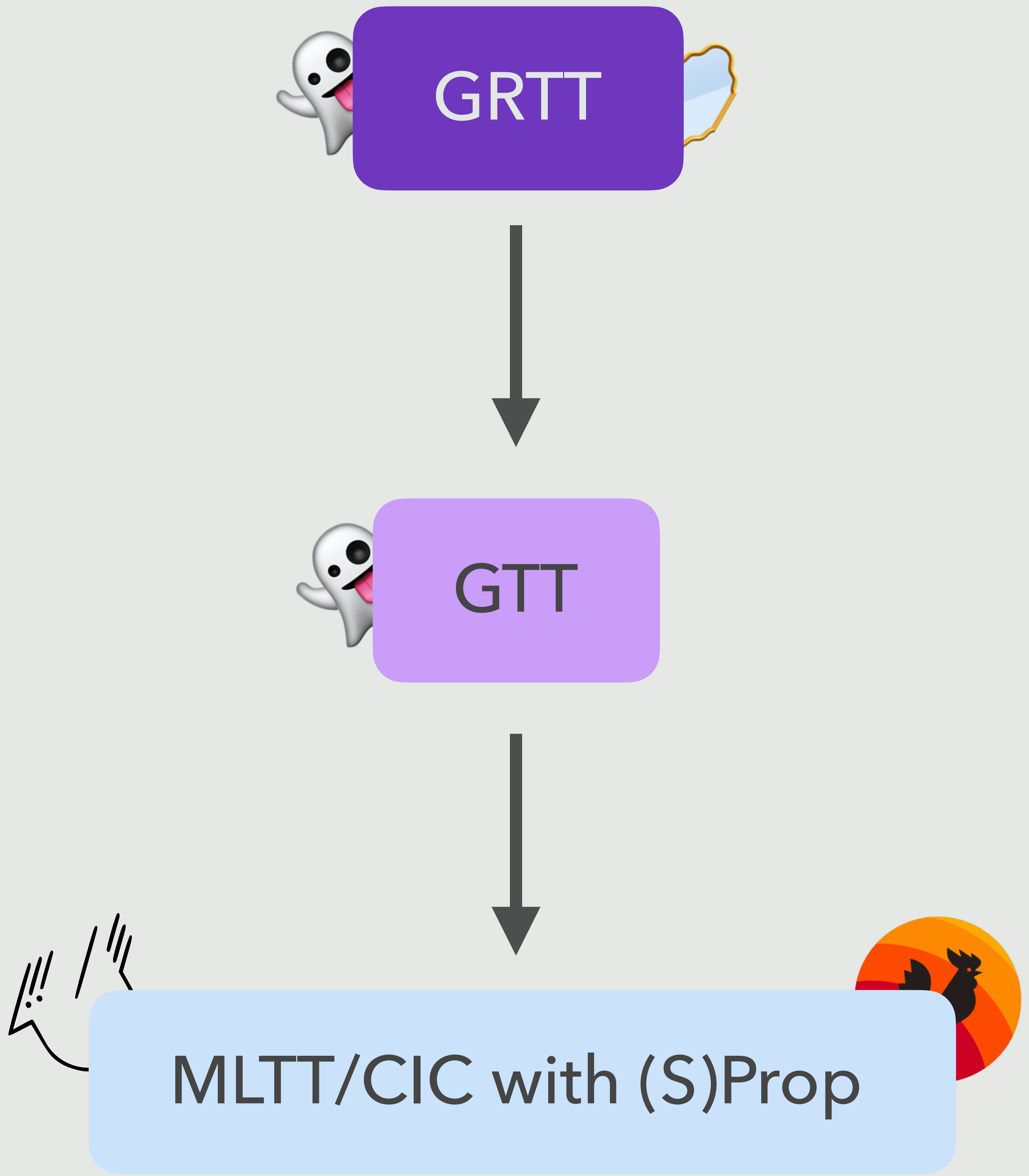
Vectors

```
Inductive vec A : erased ℕ → Type :=
| vnil : vec A (hide 0)
| vcons (a : A) n (v : vec A n) : vec A (gS n)
```

```
Inductive vec• (A : ty) :=
| vnil•
| vcons• (a : El A) (v : vec• A)
| vec∅
```

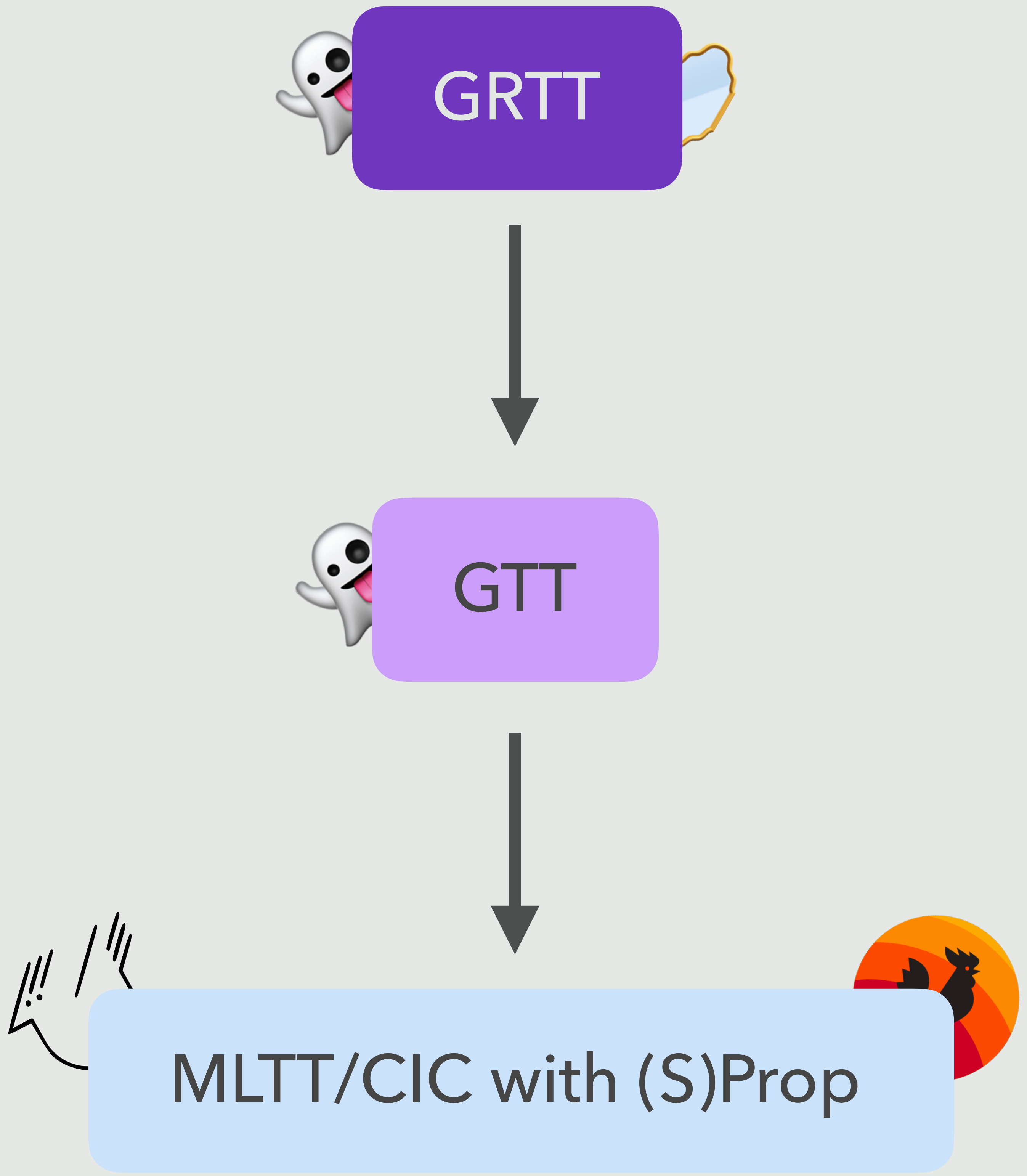
```
Inductive vecₚ (A : ty) (AP : El A → Prop) : ∀ n (nP : ℕₚ n), vec• A → Prop :=
| vnilₚ : vecₚ A AP 0• 0ₚ vnil•
| vconsₚ a (aP : AP a) n nP v : vecₚ A AP n nP v → vecₚ A AP (S• n) (Sₚ n nP) (vcons• a v)
```

+ extra oddities in the paper


GRTT
GTT
MLTT/CIC with (S)Prop

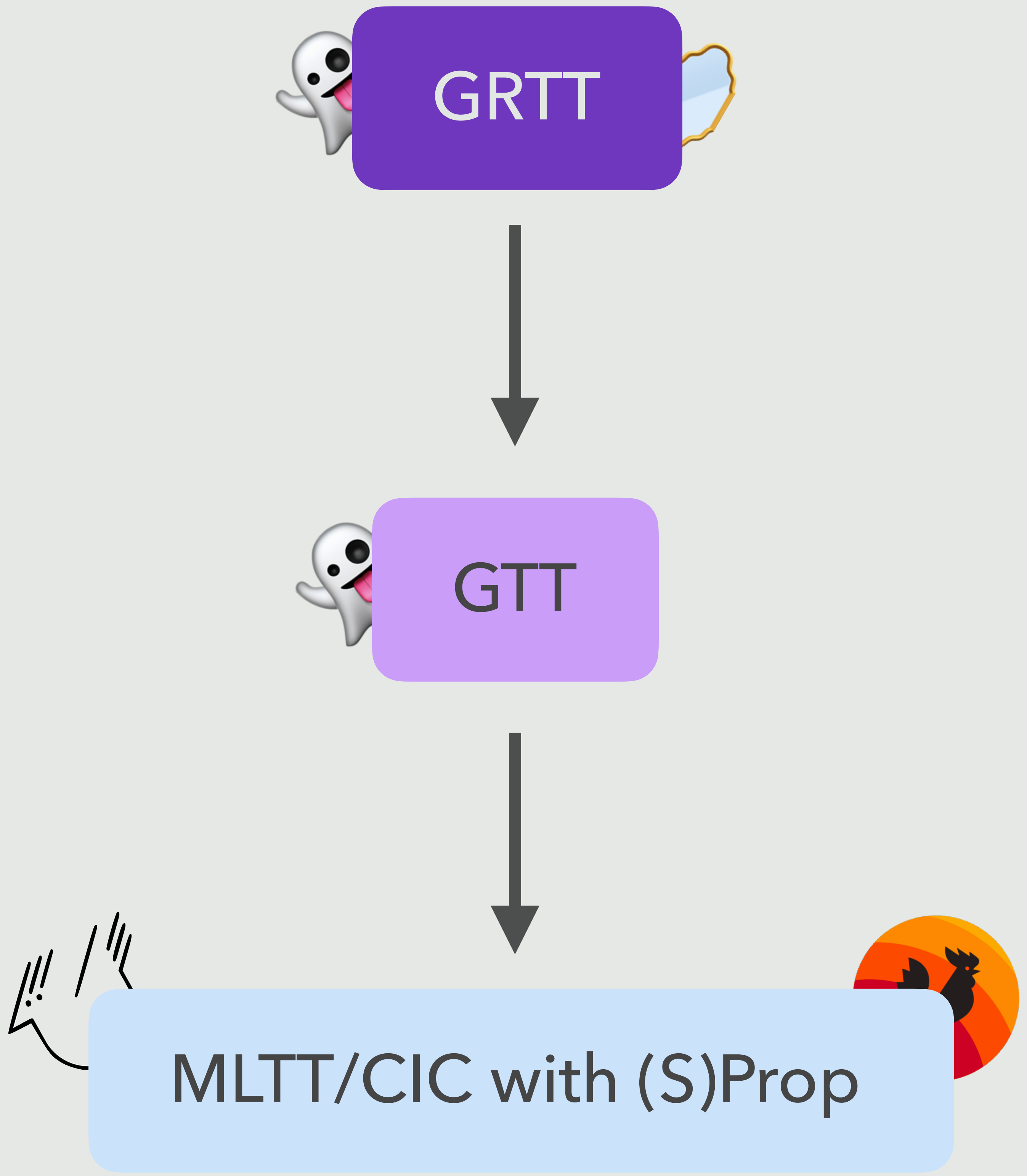
Meta-theory

conservativity
consistency
type former discrimination
free theorems



Meta-theory

conservativity
consistency
type former discrimination
free theorems

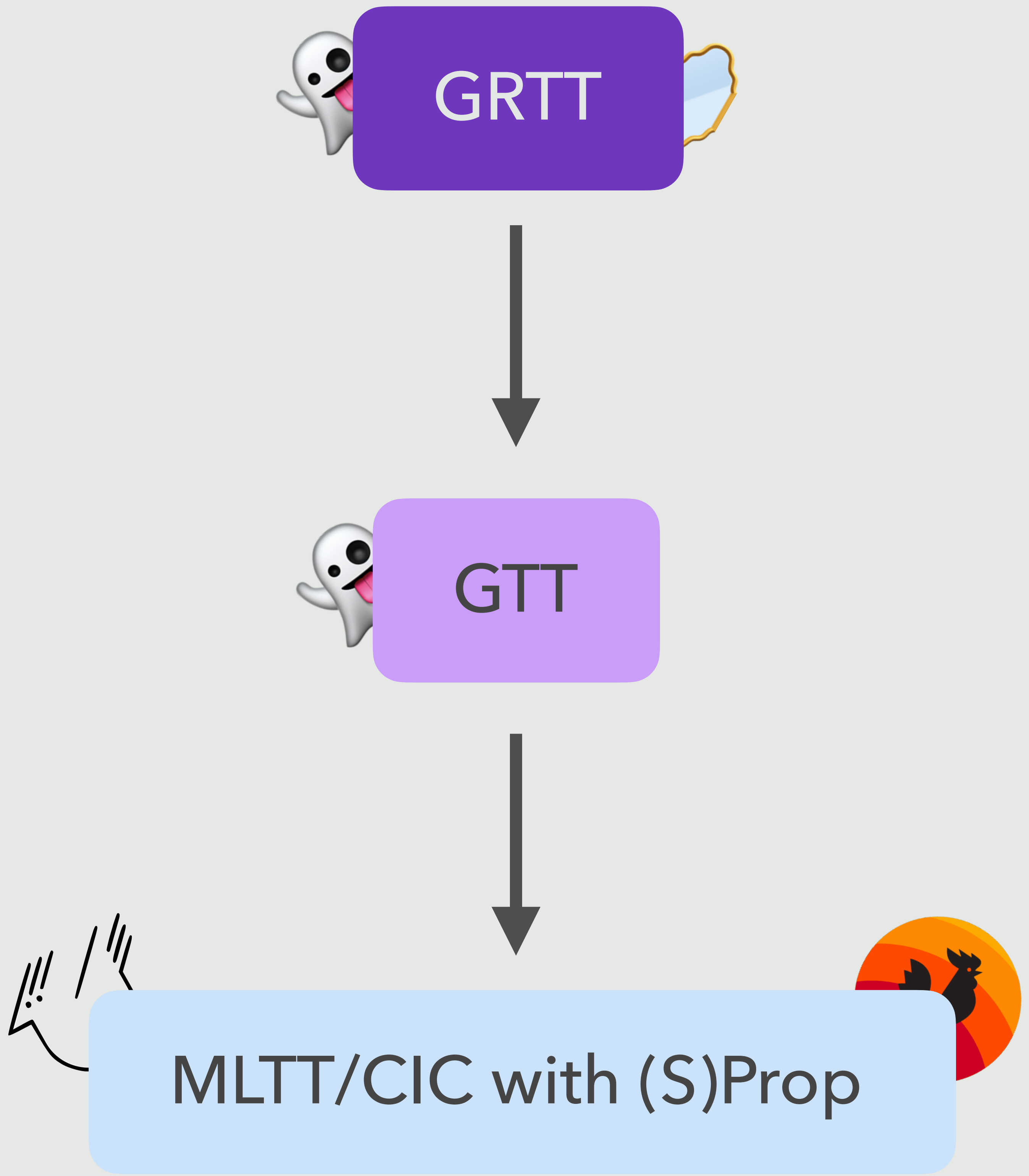


Perspectives

general inductives
subject reduction
termination
decidability (for GTT only)
meta-theory of F*

Meta-theory

conservativity
consistency
type former discrimination
free theorems



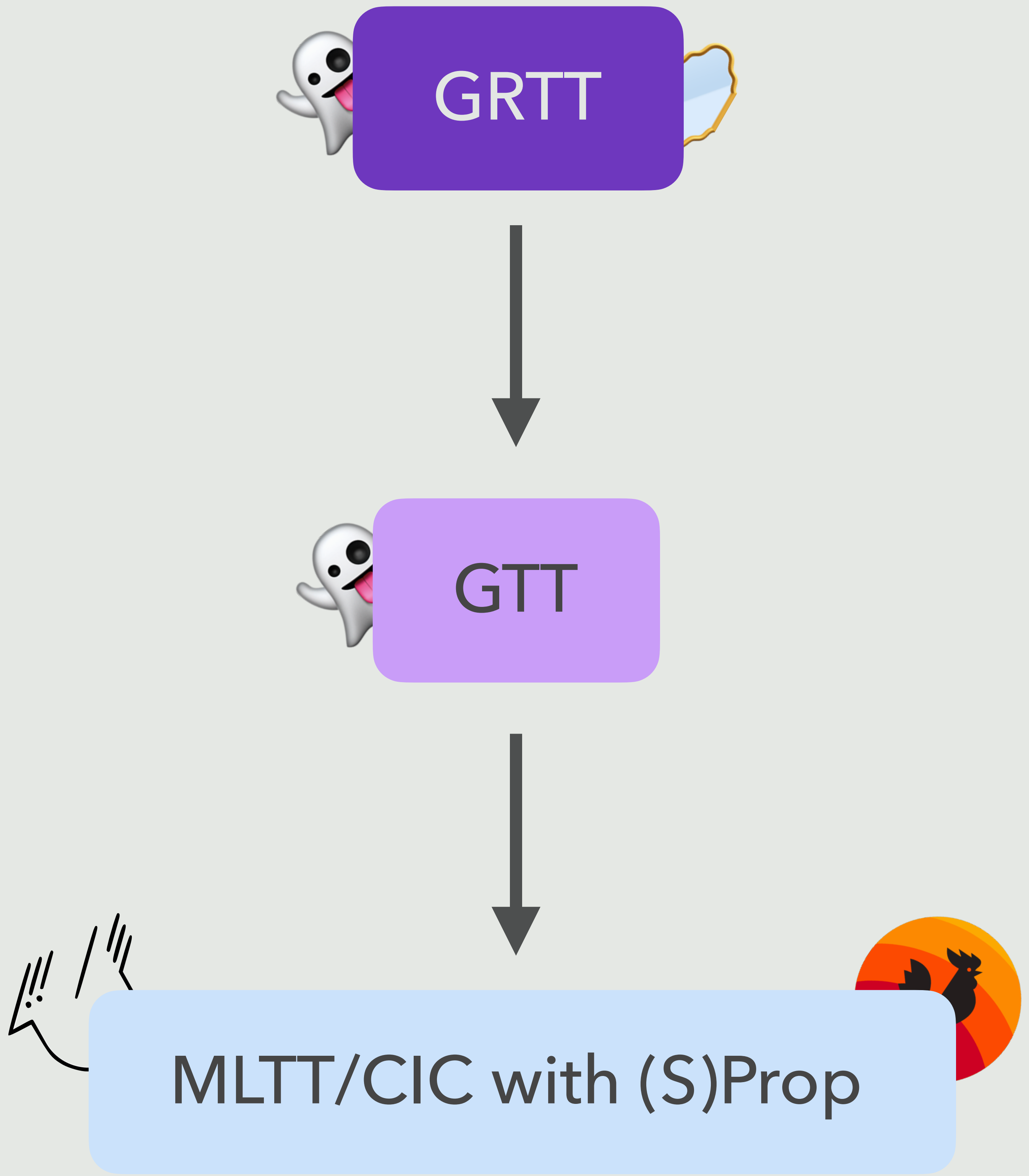
Perspectives

general inductives
subject reduction
termination
decidability (for GTT only)
meta-theory of F*

ongoing work by **Ewen Broudin--Caradec**

Meta-theory

conservativity
consistency
type former discrimination
free theorems



Perspectives

general inductives
subject reduction
termination
decidability (for GTT only)
meta-theory of F*

ongoing work by **Ewen Broudin--Caradec**

some tricks in the formalisation to handle contexts of varying size

Autosubst 2 very useful but had to rewrite automation

/TheoWinterhalter/ghost-reflection