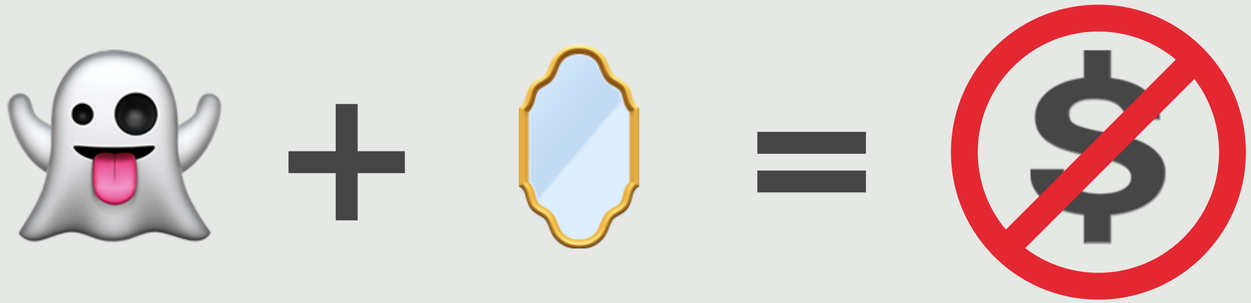
chocola

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)
```

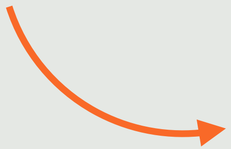
```
type 'a vec =
| Vnil
| Vcons of 'a * nat * 'a vec
```

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)))
```

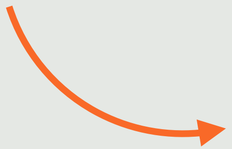
What's up with vectors?

```
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)))
```

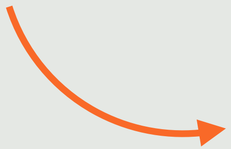
What's up with vectors?

```
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!



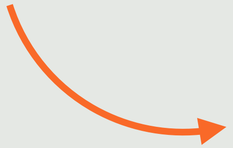
```
val rev : nat → nat → 'a vec → 'a vec → 'a vec
let rev _ m v acc =
  match v with
  | Vnil → acc
  | Vcons (a,k,w) → Obj.magic (rev k (S m) w (Vcons (a,m,acc)))
```

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
```

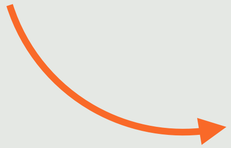
erased is removed at extraction

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
```

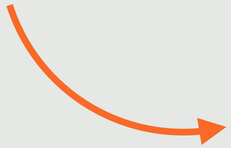
erased is removed at extraction

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
```



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

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

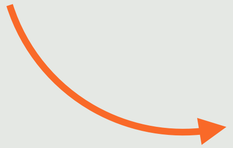
erased is removed at extraction

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

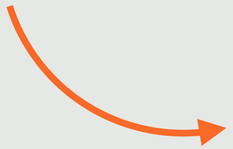
```
gS : erased ℕ → erased ℕ
gS n := reveal n as x in hide (S x)
```

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
```

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

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

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
```

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

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
```

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

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`

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

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
```

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

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
```

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

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

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

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
```

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

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

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

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
```

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

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

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

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
```

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

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

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

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

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

```
Reveal (hide t) f ↔ f t
```

Reveal proposition

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

```
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?

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

Ghost reflection

How do we justify this?

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

Ghost reflection

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

Ghost casts

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]

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

Ghost casts

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]

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

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

Ghost casts

ignored for conversion

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

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]

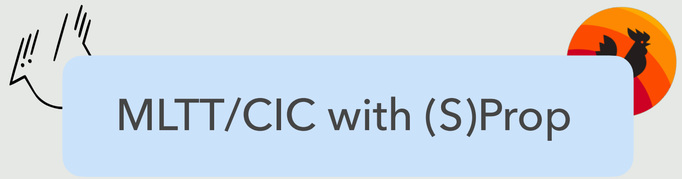
How do we justify this?

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

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

Ghost casts

ignored for conversion



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?

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

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

Ghost casts

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?

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

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

Ghost casts

ignored for conversion

interesting on its own!

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]

How do we justify this?

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

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

Ghost casts

ignored for conversion

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

Eliminating reflection

General case: Extenstional and intensional type theory

ETT = ITT + $\dfrac{e : u = v}{u \equiv v}$ 

Equality reflection

Propositionally equal **terms**

are definitionally equal

Eliminating reflection

General case: Extenstional and intensional type theory

ETT = ITT + $\dfrac{e : u = v}{u \equiv v}$ 

Equality reflection
Propositionally equal **terms**
are definitionally equal

 Key idea for ETT → ITT

Use transports everywhere
equality reflection is needed

Eliminating reflection

General case: ETT → ITT

We want $[\![.]\!]$ such that

ETT

$$\Gamma \vdash_x t : A \quad \text{implies} \quad [\![\Gamma]\!] \vdash_i [\![t]\!] : [\![A]\!]$$

$$\Gamma \vdash_x u \equiv v : A \quad \text{implies} \quad [\![\Gamma]\!] \vdash_i p : [\![u]\!] =_{[\![A]\!]} [\![v]\!]$$

ITT

Eliminating reflection

General case: ETT → ITT

We want $[\![.]\!]$ such that

ETT

$$\Gamma \vdash_x t : A \quad \text{implies} \quad [\![\Gamma]\!] \vdash_i [\![t]\!] : [\![A]\!]$$

$$\Gamma \vdash_x u \equiv v : A \quad \text{implies} \quad [\![\Gamma]\!] \vdash_i p : [\![u]\!] =_{[\![A]\!]} [\![v]\!]$$

ITT

Let's look at the **conversion** rule

$$\frac{\Gamma \vdash_x t : A \qquad \Gamma \vdash_x A \equiv B}{\Gamma \vdash_x t : B}$$

Eliminating reflection

General case: ETT → ITT

We want $[\![.]\!]$ such that

ETT

$$\Gamma \vdash_x t : A \quad \text{implies} \quad [\![\Gamma]\!] \vdash_i [\![t]\!] : [\![A]\!]$$

$$\Gamma \vdash_x u \equiv v : A \quad \text{implies} \quad [\![\Gamma]\!] \vdash_i p : [\![u]\!] =_{[\![A]\!]} [\![v]\!]$$

ITT

Let's look at the **conversion** rule

$$\frac{\Gamma \vdash_x t : A \qquad \Gamma \vdash_x A \equiv B}{\Gamma \vdash_x t : B}$$

$$[\![\Gamma]\!] \vdash_i [\![t]\!] : [\![A]\!]$$

Eliminating reflection

General case: ETT → ITT

We want $[\![.]\!]$ such that

ETT

$$\Gamma \vdash_x t : A \quad \text{implies} \quad [\![\Gamma]\!] \vdash_i [\![t]\!] : [\![A]\!]$$

$$\Gamma \vdash_x u \equiv v : A \quad \text{implies} \quad [\![\Gamma]\!] \vdash_i p : [\![u]\!] =_{[\![A]\!]} [\![v]\!]$$

ITT

Let's look at the **conversion** rule

$$\frac{\Gamma \vdash_x t : A \qquad \Gamma \vdash_x A \equiv B}{\Gamma \vdash_x t : B} \quad \dashrightarrow \quad \begin{array}{l} [\![\Gamma]\!] \vdash_i [\![t]\!] : [\![A]\!] \\ [\![\Gamma]\!] \vdash_i p : [\![A]\!] = [\![B]\!] \end{array}$$

Eliminating reflection

General case: ETT → ITT

We want $[\![.]\!]$ such that

ETT

$$\Gamma \vdash_x t : A \quad \text{implies} \quad [\![\Gamma]\!] \vdash_i [\![t]\!] : [\![A]\!]$$

$$\Gamma \vdash_x u \equiv v : A \quad \text{implies} \quad [\![\Gamma]\!] \vdash_i p : [\![u]\!] =_{[\![A]\!]} [\![v]\!]$$

ITT

Let's look at the **conversion** rule

$$\frac{\Gamma \vdash_x t : A \qquad \Gamma \vdash_x A \equiv B}{\Gamma \vdash_x t : B}$$

$$\dashrightarrow$$

$$[\![\Gamma]\!] \vdash_i [\![t]\!] : [\![A]\!]$$

$$[\![\Gamma]\!] \vdash_i p : [\![A]\!] = [\![B]\!]$$

so $[\![\Gamma]\!] \vdash_i \mathtt{transp}\ p\ [\![t]\!] : [\![B]\!]$

Eliminating reflection

General case: ETT → ITT

We want $[\![.]\!]$ such that

ETT

$$\Gamma \vdash_x t : A \quad \text{implies} \quad [\![\Gamma]\!] \vdash_i [\![t]\!] : [\![A]\!]$$

$$\Gamma \vdash_x u \equiv v : A \quad \text{implies} \quad [\![\Gamma]\!] \vdash_i p : [\![u]\!] =_{[\![A]\!]} [\![v]\!]$$

ITT

Let's look at the **conversion** rule

$$\frac{\Gamma \vdash_x t : A \qquad \Gamma \vdash_x A \equiv B}{\Gamma \vdash_x t : B}$$

$$[\![\Gamma]\!] \vdash_i [\![t]\!] : [\![A]\!]$$

$$[\![\Gamma]\!] \vdash_i p : [\![A]\!] = [\![B]\!]$$

so $[\![\Gamma]\!] \vdash_i \mathtt{transp}\ p\ [\![t]\!] : [\![B]\!]$

We wanted $[\![t]\!]$!

Eliminating reflection

General case: ETT → ITT

We want $[\![.]\!]$ such that

ETT

$$\Gamma \vdash_x t : A \quad \text{implies} \quad [\![\Gamma]\!] \vdash_i [\![t]\!] : [\![A]\!]$$

$$\Gamma \vdash_x u \equiv v : A \quad \text{implies} \quad [\![\Gamma]\!] \vdash_i p : [\![u]\!] =_{[\![A]\!]} [\![v]\!]$$

ITT

$[\![t]\!]$ should be a *class of terms* with

$$t' \in [\![t]\!] \quad \text{implies} \quad \texttt{transp}\ p\ t' \in [\![t]\!]$$

Eliminating reflection

General case: ETT → ITT

We want $[\![.]\!]$ such that

ETT

$$\Gamma \vdash_x t : A \quad \text{implies} \quad \Gamma' \vdash_i t' : A'$$

$$\Gamma \vdash_x u \equiv v : A \quad \text{implies} \quad \Gamma' \vdash_i p : u' =_{A'} v'$$

ITT

for some $\Gamma' \in [\![\Gamma]\!]$, $A' \in [\![A]\!]$, …

$[\![t]\!]$ should be a *class of terms* with

$$t' \in [\![t]\!] \quad \text{implies} \quad \mathtt{transp}\ p\ t' \in [\![t]\!]$$

Eliminating reflection

General case: ETT → ITT

We want $[\![.]\!]$ such that

ETT

$\Gamma \vdash_{\mathbf{x}} t : A$ implies $\Gamma' \vdash_{\mathbf{i}} t' : A'$

$\Gamma \vdash_{\mathbf{x}} u \equiv v : A$ implies $\Gamma' \vdash_{\mathbf{i}} p : u' =_{A'} v'$

ITT

for some $\Gamma' \in [\![\Gamma]\!]$, $A' \in [\![A]\!]$, …

conversion rule again:

$$\frac{\Gamma \vdash_x t : A \qquad \Gamma \vdash_x A \equiv B}{\Gamma \vdash_x t : B}$$

⇢

$\Gamma' \vdash_i t' : A'$

$\Gamma'' \vdash_i p : A'' =_T B'$

Eliminating reflection

General case: ETT → ITT

We want $[\![.]\!]$ such that

ETT

$\Gamma \vdash_x t : A$ implies $[\![\Gamma]\!] \vdash_i [\![t]\!] : [\![A]\!]$

$\Gamma \vdash_x u \equiv v : A$ implies $[\![\Gamma]\!] \vdash_i p : [\![u]\!] =_{[\![A]\!]} [\![v]\!]$

ITT

for some $[\![\Gamma]\!] \in [\![\Gamma]\!]$, $[\![A]\!] \in [\![A]\!]$, …

conversion rule again:

$$\frac{\Gamma \vdash_x t : A \qquad \Gamma \vdash_x A \equiv B}{\Gamma \vdash_x t : B}$$

$\Rightarrow$

$$\Gamma' \vdash_i t' : A'$$

$$\Gamma'' \vdash_i p : A'' =_{T'} B'$$

$T' \in [\![\mathrm{Type}]\!]$

Eliminating reflection

General case: ETT → ITT

We want $[\![.]\!]$ such that

ETT

$\Gamma \vdash_x t : A$ implies $\Gamma' \vdash_i t' : A'$

$\Gamma \vdash_x u \equiv v : A$ implies $\Gamma' \vdash_i p : u' =_{A'} v'$

ITT

for some $\Gamma' \in [\![\Gamma]\!]$, $A' \in [\![A]\!]$, …

conversion rule again:

$$\frac{\Gamma \vdash_x t : A \qquad \Gamma \vdash_x A \equiv B}{\Gamma \vdash_x t : B}$$

$\Gamma' \vdash_i t' : A'$

$\Gamma'' \vdash_i p : A'' =_{T'} B'$

$T' \in [\![\mathsf{Type}]\!]$

we need to relate two translations of the same object (at possibly two different types)

Eliminating reflection

General case: ETT → ITT

We want $[\![.]\!]$ such that

ETT

$$\Gamma \vdash_x t : A \quad \text{implies} \quad \Gamma' \vdash_i t' : A'$$

$$\Gamma \vdash_x u \equiv v : A \quad \text{implies} \quad \Gamma' \vdash_i p : u' =_{A'} v'$$

ITT

for some $\Gamma' \in [\![\Gamma]\!], A' \in [\![A]\!], \ldots$

Fundamental lemma

If $t', t'' \in [\![t]\!]$ then
t' and t'' are heterogeneously equal

Eliminating reflection

General case: ETT → ITT

We want $[\![.]\!]$ such that

ETT

$$\Gamma \vdash_x t : A \quad \text{implies} \quad \Gamma' \vdash_i t' : A'$$

$$\Gamma \vdash_x u \equiv v : A \quad \text{implies} \quad \Gamma' \vdash_i p : u' =_{A'} v'$$

ITT

for some $\Gamma' \in [\![\Gamma]\!], A' \in [\![A]\!], \ldots$

Fundamental lemma

If $t', t'' \in [\![t]\!]$ then
t' and t'' are heterogeneously equal

Then we need UIP and FunExt...
and we end up with enormous terms!

Eliminating reflection

Everything is better with ghosts!

GRTT

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

Ghost reflection

GTT

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

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

ignored for conversion

Ghost casts

Eliminating reflection

Everything is better with ghosts!

GRTT

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

Ghost reflection

GTT

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

Ghost casts

Two translations of the same term only differ in casts and are thus definitionally equal

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

ignored for conversion

Eliminating reflection

Everything is better with ghosts!

We define ⟦.⟧ a class of terms such that

GRTT

$$\Gamma \vdash_x t : A \quad \text{implies} \quad \Gamma' \vdash_g t' : A'$$

$$\Gamma \vdash_x u \equiv v : A \quad \text{implies}^* \quad \Gamma' \vdash_g u' \equiv v' : A'$$

GTT

for some $\Gamma' \in \llbracket \Gamma \rrbracket$, $A' \in \llbracket A \rrbracket$, …

Eliminating reflection

Everything is better with ghosts!

We define ⟦.⟧ a class of terms such that

GRTT

$$\Gamma \vdash_x t : A \quad \text{implies} \quad \Gamma' \vdash_g t' : A'$$

$$\Gamma \vdash_x u \equiv v : A \quad \text{implies}^* \quad \Gamma' \vdash_g u' \equiv v' : A'$$

GTT

for some $\Gamma' \in \llbracket \Gamma \rrbracket$, $A' \in \llbracket A \rrbracket$, …

But there's a catch! Ghost reflection only applies at the top-level.

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

Eliminating reflection

Everything is better with ghosts!

We define ⟦.⟧ a class of terms such that

GRTT

$$\Gamma \vdash_x t : A \quad \text{implies} \quad \Gamma' \vdash_g t' : A'$$

$$\Gamma \vdash_x u \equiv v : A \quad \text{implies}^* \quad \Gamma' \vdash_g u' \equiv v' : A'$$

GTT

for some $\Gamma' \in \llbracket \Gamma \rrbracket$, $A' \in \llbracket A \rrbracket$, …

But there's a catch! Ghost reflection only applies at the top-level.

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

In other words, no going under binders…
Conjecture. Assuming FunExt for ghosts should be enough.

Consequences of the translation

Conservativity

If $\vdash_x$ `A : Type` and $\vdash_x$ `t : A`

then there exists $\vdash_g$ `t' : A` with `t' ∈ ⟦t⟧`

Consequences of the translation

Conservativity

If $\vdash_x$ `A : Type` and $\vdash_x$ `t : A`

then there exists $\vdash_g$ `t' : A` with `t' ∈ ⟦t⟧`

Proof using the fact that `A` is definitionally equal to any of its translations `A'`

Consequences of the translation

Conservativity

If $\vdash_x$ `A : Type` and $\vdash_x$ `t : A`

then there exists $\vdash_g$ `t' : A` with `t' ∈ ⟦t⟧`

Proof using the fact that `A` is definitionally equal to any of its translations `A'`

Relative consistency

If $\vdash_x$ `t : ⊥`

then there exists $\vdash_g$ `t' : ⊥`

Consequences of the translation

Conservativity

If $\vdash_x$ `A : Type` and $\vdash_x$ `t : A`

then there exists $\vdash_g$ `t' : A` with `t' ∈ ⟦t⟧`

Proof using the fact that `A` is definitionally equal to any of its translations `A'`

Relative consistency

If $\vdash_x$ `t : ⊥`

then there exists $\vdash_g$ `t' : ⊥`

Proof using conservativity

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?

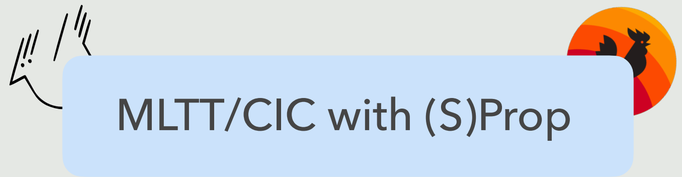
$$\frac{A : \mathsf{Ghost} \qquad e : u =_A v \qquad t : P\ u}{\mathtt{cast}\ e\ P\ t : P\ v}$$

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

Ghost casts

ignored for conversion

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



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

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

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

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

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

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

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

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

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

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

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

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

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

Erasure

Translation getting rid of all ghosts and proofs

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

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

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

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

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

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

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

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

Erasure

Translation getting rid of all ghosts and proofs

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

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

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

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

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

$\mathtt{exfalso}^{\mathbb{T}}\ (A : \mathtt{Type})\ (p : \bot) : A$ $\quad [\mathtt{exfalso}^{\mathbb{T}}\ A\ p]_\varepsilon := \text{“}\mathtt{raise}\ [\![A]\!]_\varepsilon\text{”}$ we get no $\bot$ but we need some $[\![A]\!]_\varepsilon$

Erasure

Translation getting rid of all ghosts and proofs

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

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

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

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

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

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

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

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

$$A : \mathtt{Type} \longrightarrow \begin{array}{l} [\![A]\!]_\varepsilon : \mathtt{Type} \\ [\![A]\!]_\varnothing : [\![A]\!]_\varepsilon \end{array}$$

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

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

```
∀ (f : erased bool → bool) (P : bool → Prop),
  P (f (hide true)) → P (f (hide false))
```

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

```
∀ (f : erased bool → bool) (P : bool → Prop),
  P (f (hide true)) → P (f (hide false))
```

is justified in the model by

```
∀ (fe : bool•) (fP : ∀ b, boolP b → boolP fe)
  (P : bool• → unit) (PP : ∀ b, boolP b → Prop),
  PP fe (fP true• trueP) → PP fe (fP false• falseP)
```

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

```
∀ (f : erased bool → bool) (P : bool → Prop),
  P (f (hide true)) → P (f (hide false))
```

is justified in the model by

```
∀ (fe : bool•) (fP : ∀ b, boolP b → boolP fe)
  (P : bool• → unit) (PP : ∀ b, boolP b → Prop),
  PP fe (fP true• trueP) → PP fe (fP false• falseP)
```

proof irrelevance

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

```
∀ (f : erased bool → bool) (P : bool → Prop),
  P (f (hide true)) → P (f (hide false))
```

is justified in the model by

```
∀ (fe : bool•) (fP : ∀ b, boolP b → boolP fe)
  (P : bool• → unit) (PP : ∀ b, boolP b → Prop),
  PP fe (fP true• trueP) → PP fe (fP false• falseP)
```

the key is that it is erased to a constant

proof irrelevance

Vectors

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

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⌀
```

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)
```

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)
```

```
⟦vec A n⟧P := vecP [A]ε ⟦A⟧P ⟦n⟧v ⟦n⟧P
```

we need **revival** to recover n

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)
```

```
⟦vec A n⟧P := vecP [A]ε ⟦A⟧P ⟦n⟧v ⟦n⟧P
```

we need **revival** to recover n

Computation for the **eliminator**

```
vec-elim vnil P z s ≡ z
```

Unusual

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)
```

```
⟦vec A n⟧P := vecP [A]ε ⟦A⟧P ⟦n⟧v ⟦n⟧P
```

we need **revival** to recover n

Computation for the **eliminator**

```
vec-elim vnil P z s ≡ z
```

```
vec-elim (vcons a n v) P z s ≡ s a (glength v) v (vec-elim v P z s)
```

Revival

Revival essentially gets back what was erased

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{G}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_V : [\![A]\!]_\varepsilon$ in CC

(undefined otherwise)

Revival

Revival essentially gets back what was erased

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{G}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_V : [\![A]\!]_\varepsilon$ in CC

(undefined otherwise)

Here we use modes: $\mathbb{T}$, $\mathbb{G}$, and $\mathbb{P}$ that are syntactically determined

$\Gamma \vdash t :: \mathbb{G}$ approximates
$\Gamma \vdash t : A :$ `Ghost` for some A

Revival

Revival essentially gets back what was erased

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{G}$ in GTT
then $[\![\Gamma]\!]_{\mathbb{P}} \vdash [\![t]\!]_{\mathbb{V}} : [\![A]\!]_{\varepsilon}$ in CC

(undefined otherwise)

Here we use modes: $\mathbb{T}$, $\mathbb{G}$, and $\mathbb{P}$ that are syntactically determined

$\Gamma \vdash t :: \mathbb{G}$ approximates
$\Gamma \vdash t : A :$ `Ghost` for some A

$[\![\texttt{hide}\ t]\!]_{\mathbb{V}} := [t]_{\varepsilon}$

Revival

Revival essentially gets back what was erased

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{G}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_V : [\![A]\!]_\varepsilon$ in CC

(undefined otherwise)

Here we use modes: $\mathbb{T}$, $\mathbb{G}$, and $\mathbb{P}$ that are syntactically determined

$\Gamma \vdash t :: \mathbb{G}$ approximates $\Gamma \vdash t : A : \texttt{Ghost}$ for some A

$[\![\texttt{hide}\ t]\!]_V := [t]_\varepsilon$

$[\![\texttt{reveal}\ e\ P\ f^{\mathbb{G}}]\!]_V := [\![f]\!]_V\ [\![e]\!]_V$

Revival

Revival essentially gets back what was erased

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{G}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_V : [\![A]\!]_\varepsilon$ in CC

(undefined otherwise)

Here we use modes: $\mathbb{T}$, $\mathbb{G}$, and $\mathbb{P}$ that are syntactically determined

$\Gamma \vdash t :: \mathbb{G}$ approximates
$\Gamma \vdash t : A :$ `Ghost` for some A

$$[\![\texttt{hide}\ t]\!]_V := [t]_\varepsilon \qquad [\![\texttt{reveal}\ e\ P\ f^{\mathbb{G}}]\!]_V := [\![f]\!]_V\ [\![e]\!]_V$$

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

Summary of the model

revival

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{G}$ in GTT

then $[\![\Gamma]\!]_P \vdash [\![t]\!]_v : [\![A]\!]_\varepsilon$ in CC

Summary of the model

erasure

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{T}$ in GTT
then $[\![\Gamma]\!]_\varepsilon \vdash [t]_\varepsilon : [\![A]\!]_\varepsilon$ in CC

revival

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{G}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_V : [\![A]\!]_\varepsilon$ in CC

Summary of the model

erasure

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{T}$ in GTT
then $[\![\Gamma]\!]_\varepsilon \vdash [t]_\varepsilon : [\![A]\!]_\varepsilon$ in CC

revival

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{G}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_V : [\![A]\!]_\varepsilon$ in CC

parametricity

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{T}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_P : [\![A]\!]_P\ [t]_\varepsilon$ in CC

Summary of the model

erasure

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{T}$ in GTT
then $[\![\Gamma]\!]_\varepsilon \vdash [t]_\varepsilon : [\![A]\!]_\varepsilon$ in CC

revival

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{G}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_V : [\![A]\!]_\varepsilon$ in CC

parametricity

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{T}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_P : [\![A]\!]_P\ [t]_\varepsilon$ in CC

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{G}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_P : [\![A]\!]_P\ [\![t]\!]_V$ in CC

Summary of the model

erasure

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{T}$ in GTT
then $[\![\Gamma]\!]_\varepsilon \vdash [\![t]\!]_\varepsilon : [\![A]\!]_\varepsilon$ in CC

revival

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{G}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_V : [\![A]\!]_\varepsilon$ in CC

parametricity

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{T}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_P : [\![A]\!]_P \ [\![t]\!]_\varepsilon$ in CC

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{G}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_P : [\![A]\!]_P \ [\![t]\!]_V$ in CC

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{P}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_P : [\![A]\!]_P$ in CC

Summary of the model

erasure

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{T}$ in GTT
then $[\![\Gamma]\!]_\varepsilon \vdash [\![t]\!]_\varepsilon : [\![A]\!]_\varepsilon$ in CC

revival

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{G}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_V : [\![A]\!]_\varepsilon$ in CC

parametricity

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{T}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_P : [\![A]\!]_P \; [\![t]\!]_\varepsilon$ in CC

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{G}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_P : [\![A]\!]_P \; [\![t]\!]_V$ in CC

If $\Gamma \vdash t : A$ and $\Gamma \vdash t :: \mathbb{P}$ in GTT
then $[\![\Gamma]\!]_P \vdash [\![t]\!]_P : [\![A]\!]_P$ in CC

Propositions essentially preserved (including $\bot$)
Hence consistency

Meta-theory

Joint work with Ewen Broudin--Caradec

Injectivity of Π types

If `Γ ⊢ ∀ (x : A). B ≡ ∀ (x : A'). B'`

then `Γ ⊢ A ≡ A'` and `Γ, x : A ⊢ B ≡ B'`

Meta-theory

Joint work with Ewen Broudin--Caradec

Injectivity of Π types

If `Γ ⊢ ∀ (x : A). B ≡ ∀ (x : A'). B'`

then `Γ ⊢ A ≡ A'` and `Γ, x : A ⊢ B ≡ B'`

violated by ETT!

Meta-theory

Joint work with Ewen Broudin--Caradec

Injectivity of Π types

If `Γ ⊢ ∀ (x : A). B ≡ ∀ (x : A'). B'`

then `Γ ⊢ A ≡ A'` and `Γ, x : A ⊢ B ≡ B'`

violated by ETT!

Proof by characterising definitional equality
by a confluent rewriting system

Meta-theory

Joint work with Ewen Broudin--Caradec

Injectivity of Π types

If `Γ ⊢ ∀ (x : A). B ≡ ∀ (x : A'). B'`

then `Γ ⊢ A ≡ A'` and `Γ, x : A ⊢ B ≡ B'`

violated by ETT!

Proof by characterising definitional equality
by a confluent rewriting system

We in fact can discriminate type formers
unlike F* for which it can cause trouble

Meta-theory

Joint work with Ewen Broudin--Caradec

Injectivity of Π types

If `Γ ⊢ ∀ (x : A). B ≡ ∀ (x : A'). B'`

then `Γ ⊢ A ≡ A'` and `Γ, x : A ⊢ B ≡ B'`

violated by ETT!

Proof by characterising definitional equality
by a confluent rewriting system

We in fact can discriminate type formers
unlike F* for which it can cause trouble

Uniqueness of type

If `Γ ⊢ t : A` and `Γ ⊢ t : B`

then `Γ ⊢ A ≡ B` (up to universe levels)

Meta-theory

Joint work with Ewen Broudin--Caradec

Injectivity of Π types

If `Γ ⊢ ∀ (x : A). B ≡ ∀ (x : A'). B'`

then `Γ ⊢ A ≡ A'` and `Γ, x : A ⊢ B ≡ B'`

violated by ETT!

Proof by characterising definitional equality by a confluent rewriting system

We in fact can discriminate type formers
unlike F* for which it can cause trouble

Uniqueness of type

If `Γ ⊢ t : A` and `Γ ⊢ t : B`

then `Γ ⊢ A ≡ B` (up to universe levels)

Proof by inversion of typing, using injectivity of Π for the application case

Meta-theory

Joint work with Ewen Broudin--Caradec

Injectivity of Π types

If `Γ ⊢ ∀ (x : A). B ≡ ∀ (x : A'). B'`

then `Γ ⊢ A ≡ A'` and `Γ, x : A ⊢ B ≡ B'`

violated by ETT!

Proof by characterising definitional equality by a confluent rewriting system

We in fact can discriminate type formers
unlike F* for which it can cause trouble

Uniqueness of type

If `Γ ⊢ t : A` and `Γ ⊢ t : B`

then `Γ ⊢ A ≡ B` (up to universe levels)

Proof by inversion of typing, using injectivity of Π for the application case

Mode coherence

If `Γ ⊢ t : A` and `Γ ⊢ A : Sort m`

then `Γ ⊢ t :: m`

where

```
Sort 𝕋 = Type
Sort 𝔾 = Ghost
Sort ℙ = Prop
```

Meta-theory

Joint work with Ewen Broudin--Caradec

Injectivity of Π types

If `Γ ⊢ ∀ (x : A). B ≡ ∀ (x : A'). B'`

then `Γ ⊢ A ≡ A'` and `Γ, x : A ⊢ B ≡ B'`

violated by ETT!

Proof by characterising definitional equality by a confluent rewriting system

We in fact can discriminate type formers
unlike F* for which it can cause trouble

Uniqueness of type

If `Γ ⊢ t : A` and `Γ ⊢ t : B`

then `Γ ⊢ A ≡ B` (up to universe levels)

Proof by inversion of typing, using injectivity of Π for the application case

Mode coherence

If `Γ ⊢ t : A` and `Γ ⊢ A : Sort m`

then `Γ ⊢ t :: m`

where

```
Sort 𝕋 = Type
Sort 𝔾 = Ghost
Sort ℙ = Prop
```

Proof by validity (if `Γ ⊢ t : A` then `Γ ⊢ A : Sort m` for some `m` with `Γ ⊢ t :: m`) and uniqueness of type together with injectivity of sorts (proven like for Π types)

Perspectives

Overcome the need for parametricity in Prop

Perspectives

Overcome the need for parametricity in Prop

general large elimination

Perspectives

Overcome the need for parametricity in Prop

general large elimination

`Prop : Ghost` or even `Type : Ghost` (currently done in F*)
removing the need for the special `Reveal`

Perspectives

Overcome the need for parametricity in Prop

general large elimination

`Prop : Ghost` or even `Type : Ghost` (currently done in F*)
removing the need for the special `Reveal`

Better understand the theory for inductive types

Perspectives

Overcome the need for parametricity in Prop	general large elimination `Prop : Ghost` or even `Type : Ghost` (currently done in F*) removing the need for the special `Reveal`
Better understand the theory for inductive types	fix (or understand better) the computation rule for `vec-elim`

Perspectives

Overcome the need for parametricity in Prop

general large elimination

`Prop : Ghost` or even `Type : Ghost` (currently done in F*)
removing the need for the special `Reveal`

Better understand the theory for inductive types

fix (or understand better) the computation rule for `vec-elim`

Internalise free theorems

Perspectives

Overcome the need for parametricity in Prop

general large elimination

`Prop : Ghost` or even `Type : Ghost` (currently done in F*)
removing the need for the special `Reveal`

Better understand the theory for inductive types

fix (or understand better) the computation rule for `vec-elim`

Internalise free theorems

Decidability of typing

Perspectives

Overcome the need for parametricity in Prop

general large elimination

`Prop : Ghost` or even `Type : Ghost` (currently done in F*)
removing the need for the special `Reveal`

Better understand the theory for inductive types

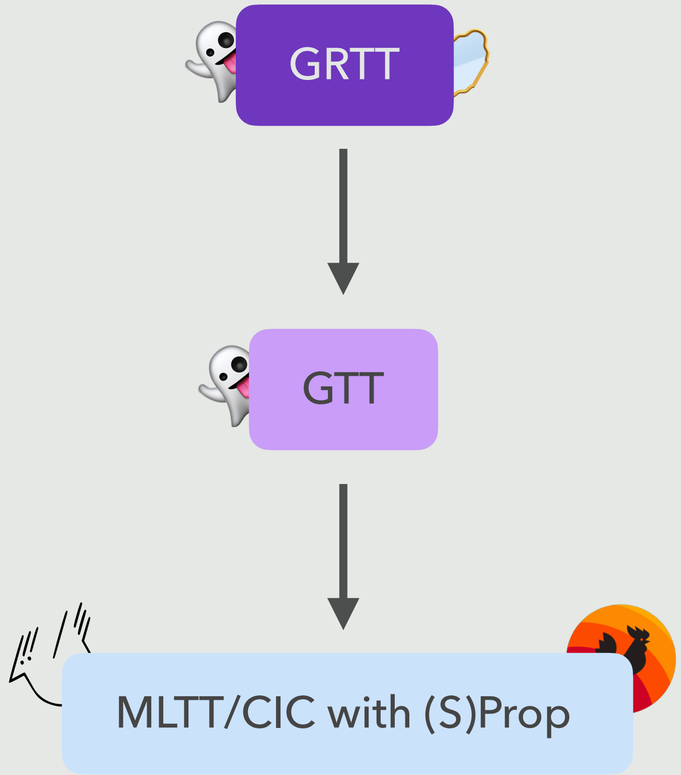
fix (or understand better) the computation rule for `vec-elim`

Internalise free theorems

Decidability of typing

Integration in Coq and MetaCoq

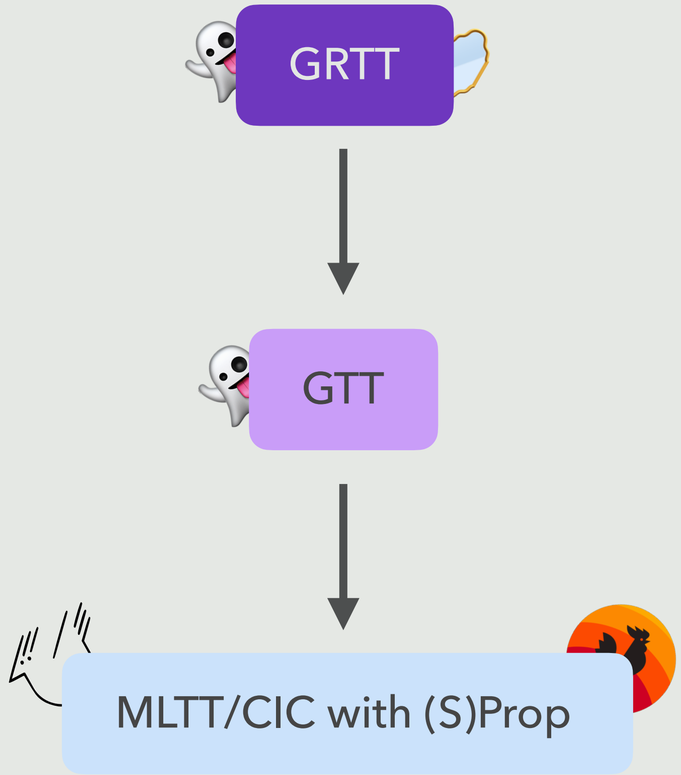
Conclusion



Conclusion

Meta-theory

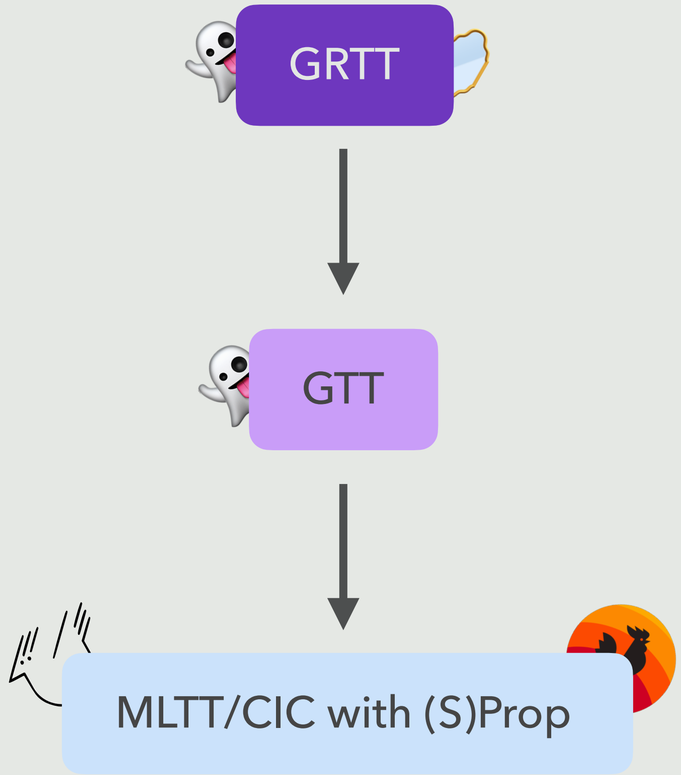
conservativity
consistency
type former discrimination
injectivity of type formers
free theorems



Conclusion

Meta-theory

conservativity
consistency
type former discrimination
injectivity of type formers
free theorems



Formalisation in Coq

from scratch using Autosubst 2
improving its automation

trick to handle contexts of varying size

/TheoWinterhalter/ghost-reflection