CPP 2019

Eliminating **reflection** through *reflection*

Théo **Winterhalter**

joint work with

Matthieu **Sozeau**

Nicolas **Tabareau**

Different notions of equality

Conversion

Extends the notion of β-equality

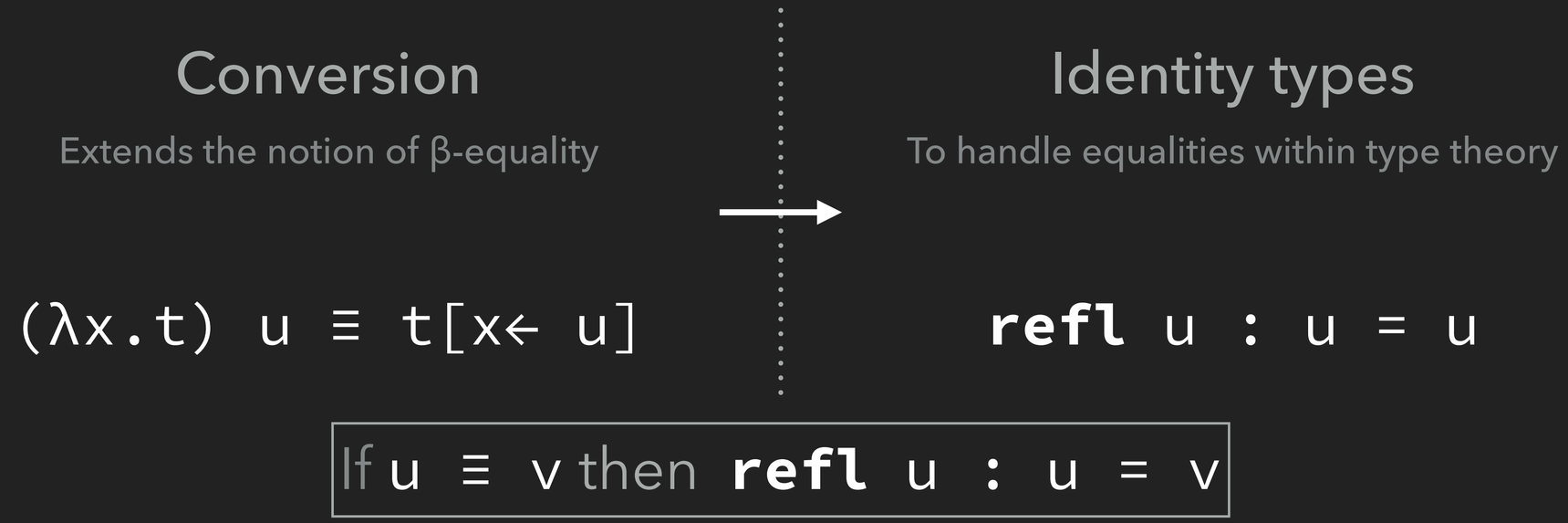
```
(λx.t) u ≡ t[x← u]
```

Identity types

To handle equalities within type theory

```
refl u : u = u
```

Different notions of equality



Reflection

Conversion

Extends the notion of β-equality

Identity types

To handle equalities within type theory

Reflection

Conversion

Extends the notion of β-equality

Identity types

To handle equalities within type theory

$$\frac{p : u = v}{u \equiv v}$$

Example

Example

```
vec_A : nat → Type
```

Example

```
vecA : nat → Type

[] : vecA 0
```

Example

```
vec_A : nat → Type

[] : vec_A 0

cons : ∀ n, A → vec_A n → vec_A (S n)
```

Example

```
vecA : nat → Type

[] : vecA 0

cons : ∀ n, A → vecA n → vecA (S n)

rev : ∀ {n m}, vecA n → vecA m → vecA (n + m)
```

Example

```
vecA : nat → Type

[] : vecA 0

cons : ∀ n, A → vecA n → vecA (S n)
```

```
rev : ∀ {n m}, vecA n → vecA m → vecA (n + m)

rev [] acc ≐ acc
```

Example

```
vecA : nat → Type

[] : vecA 0

cons : ∀ n, A → vecA n → vecA (S n)
```

```
rev : ∀ {n m}, vecA n → vecA m → vecA (n + m)

rev [] acc ≐ acc

rev (cons n a v) acc ≐ rev v (cons m a acc)
```

Example

```
vecA : nat → Type

[] : vecA 0

cons : ∀ n, A → vecA n → vecA (S n)
```

```
rev : ∀ {n m}, vecA n → vecA m → vecA (n + m)

rev [] acc ≐ acc

rev (cons n a v) acc ≐ rev v (cons m a acc)
```

vecA m

Example

```
vecA : nat → Type

[] : vecA 0

cons : ∀ n, A → vecA n → vecA (S n)
```

```
rev : ∀ {n m}, vecA n → vecA m → vecA (n + m)

rev [] acc ≐ acc

rev (cons n a v) acc ≐ rev v (cons m a acc)
```

```
vecA (S m)
```

Example

```
vecA : nat → Type

[] : vecA 0

cons : ∀ n, A → vecA n → vecA (S n)
```

```
rev : ∀ {n m}, vecA n → vecA m → vecA (n + m)

rev [] acc ≐ acc

rev (cons n a v) acc ≐ rev v (cons m a acc)
```

`vecA (n + S m)`

Example

```
vecA : nat → Type

[] : vecA 0

cons : ∀ n, A → vecA n → vecA (S n)
```

```
rev : ∀ {n m}, vecA n → vecA m → vecA (n + m)

rev [] acc ≐ acc

rev (cons n a v) acc ≐ rev v (cons m a acc)
```

expected: `vecA (S n + m)` ≠ `vecA (n + S m)`

Example

```
vecA : nat → Type

[] : vecA 0

cons : ∀ n, A → vecA n → vecA (S n)
```

```
rev : ∀ {n m}, vecA n → vecA m → vecA (n + m)

rev [] acc ≐ acc

rev (cons n a v) acc ≐ rev v (cons m a acc)
```

reflection ⇒ `vecA (S n + m) ≡ vecA (n + S m)`

Intensional VS Extensional

$$\frac{p : u = v}{u \equiv v}$$

ETT = ITT + reflection

Intensional VS Extensional

$$\frac{p : u = v}{u \equiv v}$$

ETT = ITT + reflection

What is the relation between the two?

Intensional VS Extensional

What is the relation between the two?

ETT is *conservative* over **ITT + K + funext**

1995 Martin Hofmann

Intensional VS Extensional

What is the relation between the two?

ETT is *conservative* over **ITT + K + funext**

1995 Martin Hofmann

Intensional VS Extensional

What is the relation between the two?

ETT is *conservative* over **ITT + K + funext**

1995 Martin Hofmann

```
K : ∀ A (x : A) (e : x = x), e = refl x
```

Intensional VS Extensional

What is the relation between the two?

ETT is *conservative* over **ITT + K + funext**

1995 Martin Hofmann

```
K : ∀ A (x : A) (e : x = x), e = refl x
```

```
funext : ∀ A B (f g : A → B), (∀ (x : A), f x = g x) → f = g
```

Intensional VS Extensional

What is the relation between the two?

ETT can be *translated* to **ITT + K + funext + ?**

2005 Nicolas Oury

```
K : ∀ A (x : A) (e : x = x), e = refl x
```

```
funext : ∀ A B (f g : A → B), (∀ (x : A), f x = g x) → f = g
```

Intensional VS Extensional

What is the relation between the two?

ETT can be *translated* to **ITT + K + funext + ?**

2005 Nicolas Oury

```
K : ∀ A (x : A) (e : x = x), e = refl x
```

```
funext : ∀ A B (f g : A → B), (∀ (x : A), f x = g x) → f = g
```

‘?’ : « heterogenous equality is a congruence for application »

Intensional VS Extensional

What is the relation between the two?

ETT can be *translated* to **ITT + K + funext**

TODAY

```
K : ∀ A (x : A) (e : x = x), e = refl x
```

```
funext : ∀ A B (f g : A → B), (∀ (x : A), f x = g x) → f = g
```

‘?’ : « heterogenous equality is a congruence for application »

Intensional VS Extensional

What is the relation between the two?

ETT can be *translated* to **ITT + K + funext**

TODAY

```
K : ∀ A (x : A) (e : x = x), e = refl x
```

```
funext : ∀ A B (f g : A → B), (∀ (x : A), f x = g x) → f = g
```

~~‘?’ : « heterogenous equality is a congruence for application »~~

Intensional VS Extensional

What is the relation between the two?

ETT can be *translated* to **ITT + K + funext**

TODAY

Intensional VS Extensional

What is the relation between the two?

ETT can be *translated* to **ITT + K + funext**

TODAY

- + Minimal (axiom-wise)
- + Constructive (formalised in Coq)
- + Computes (produces Coq terms)
- + « Usable » (Coq plugin proof of concept)

Fundamental difference

Oury

Hofmann / us

Fundamental difference

Oury

Minimal annotations

```
λ(x : A).t

t u
```

Hofmann / us

Fundamental difference

Oury

Minimal annotations

```
λ(x : A).t

t u
```

Hofmann / us

Fully annotated terms

```
λ(x : A).B.t

t @(x:A).B u
```

Fundamental difference

Oury

Minimal annotations

$$\lambda(x : A).t$$

$$t\ u$$

Hofmann / us

Fully annotated terms

$$\lambda(x : A).B.t$$

$$t\ @^{(x:A).B}\ u$$

Blocked β-reduction

$$(\lambda(x : A).B.t)\ @^{(x:A).B}\ u \equiv t[x := u]$$

Fundamental difference

Oury

Minimal annotations

$$\lambda(x : A).t$$

$$t\ u$$

Free β-reduction

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

Hofmann / us

Fully annotated terms

$$\lambda(x : A).B.t$$

$$t\ @^{(x:A).B}\ u$$

Blocked β-reduction

$$(\lambda(x : A).B.t)\ @^{(x:A).B}\ u \equiv t[x := u]$$

Fundamental difference

Oury

Free β-reduction

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

Hofmann / us

Blocked β-reduction

$$(\lambda(x : A).B.t)\ @^{(x:A).B}\ u \equiv t[x := u]$$

Fundamental difference

Oury

Free β-reduction

```
(λ(x : A).x) u
  ≡ x[x := u]
```

Hofmann / us

Blocked β-reduction

```
(λ(x : A).B.t) @(x:A).B u
      ≡ t[x := u]
```

Fundamental difference

Oury

Free β-reduction

$$(\lambda(x : A).x)\ u \equiv u$$

Hofmann / us

Blocked β-reduction

$$(\lambda(x : A).B.t)\ @^{(x:A).B}\ u \equiv t[x := u]$$

Fundamental difference

Oury

Free β-reduction

$$(\lambda(x : \mathsf{nat}).x)\ 0 \equiv 0$$

Hofmann / us

Blocked β-reduction

$$(\lambda(x : A).B.t)\ @^{(x:A).B}\ u \equiv t[x := u]$$

Fundamental difference

Oury

Free β-reduction

$$\frac{(\lambda(x : \mathsf{nat}).x)\ 0 \equiv 0}{\mathsf{nat} \to \mathsf{nat}}$$

Hofmann / us

Blocked β-reduction

$$(\lambda(x : A).B.t)\ @^{(x:A).B}\ u \equiv t[x := u]$$

Fundamental difference

Oury

Free β-reduction

```
(λ(x : nat).x) 0 ≡ 0
```

```
nat → nat
≡ nat → bool
```

under consistent context

Hofmann / us

Blocked β-reduction

```
(λ(x : A).B.t) @(x:A).B u
        ≡ t[x := u]
```

Fundamental difference

Oury

Free β-reduction

$$\frac{(\lambda(x : \mathsf{nat}).x)\ 0 \equiv 0}{\mathsf{bool}}$$

Hofmann / us

Blocked β-reduction

$$(\lambda(x : A).B.t)\ @^{(x:A).B}\ u \equiv t[x := u]$$

Fundamental difference

Oury

Free β-reduction

$$\frac{(\lambda(x : \mathtt{nat}).x)\ 0}{\mathtt{bool}} \equiv \frac{0}{\mathtt{nat}}$$

Hofmann / us

Blocked β-reduction

$$(\lambda(x : A).B.t)\ @^{(x:A).B}\ u \equiv t[x := u]$$

Fundamental difference

Oury

Free β-reduction

$$\frac{(\lambda(x : \mathsf{nat}).x)\ 0}{\mathsf{bool}} \equiv \frac{0}{\mathsf{nat}}$$

$\mathsf{bool} \neq \mathsf{nat}$

under consistent context

Hofmann / us

Blocked β-reduction

$$(\lambda(x : A).B.t)\ @^{(x:A).B}\ u \equiv t[x := u]$$

Fundamental difference

Oury

Free β-reduction

$$\frac{(\lambda(x : \mathsf{nat}).x)\ 0}{\mathsf{bool}} \equiv \frac{0}{\mathsf{nat}}$$

$$\mathsf{bool} \neq \mathsf{nat}$$

No Uniqueness of type

OR

No Subject reduction

Hofmann / us

Blocked β-reduction

$$(\lambda(x : A).B.t)\ @^{(x:A).B}\ u \equiv t[x := u]$$

Fundamental difference

Oury

Free β-reduction

$$\frac{(\lambda(x : \mathsf{nat}).x)\ 0 \equiv 0}{\mathsf{bool} \neq \mathsf{nat}}$$

No Uniqueness of type

OR

No Subject reduction

Hofmann / us

Blocked β-reduction

$$(\lambda(x : A).B.t)\ @^{(x:A).B}\ u \equiv t[x := u]$$

Uniqueness of type

$\Gamma \vdash t : A$ and $\Gamma \vdash t : B$

$\Rightarrow \Gamma \vdash A \equiv B$

Principle of the translation

ETT

ITT

Principle of the translation

ETT

Typing derivation

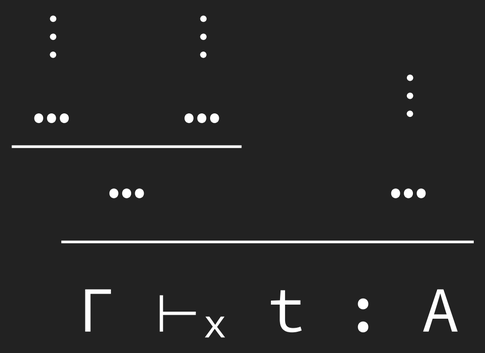
ITT

Well typed term

Principle of the translation

ETT

Typing derivation



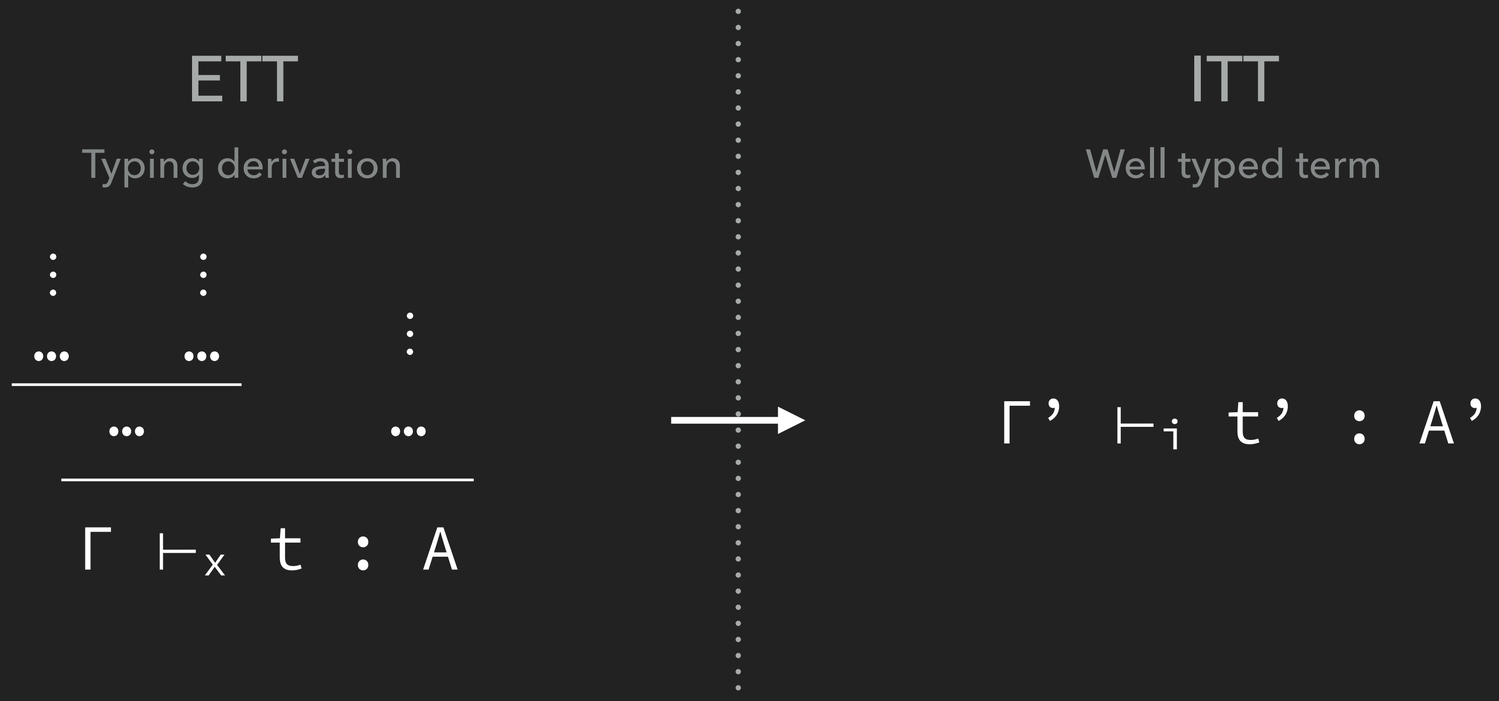
$\Gamma \vdash_x t : A$

ITT

Well typed term

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

Principle of the translation



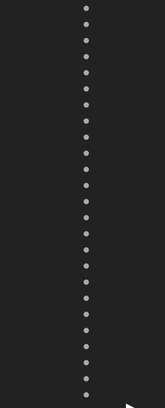
Idea: *Conversion* is translated to *transport*.

Principle of the translation

ETT

Typing derivation

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



ITT

Well typed term

$$\Gamma' \vdash_i t' : B'$$

$$\Gamma' \vdash_i e : B' = A'$$

$$\Downarrow$$

$$\Gamma' \vdash_i \mathbf{transp}(e,t') : A'$$

Idea: *Conversion* is translated to *transport*.

Principle of the translation

ETT

Typing derivation

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

ITT

Well typed term

$$\Gamma' \vdash_i t' : B'$$
$$\Gamma' \vdash_i e : B' = A'$$
$$\Downarrow$$
$$\Gamma' \vdash_i \mathbf{transp}(e,t') : A'$$

Idea: *Conversion* is translated to *transport*.

⇒ Coherence problems

Heterogenous equality

$$a \ {}_{A=B} \ b$$

Heterogenous equality

$$a \ {}_{A}{=}_{B} \ b$$

$$\doteq \sum (p : A = B), \mathbf{transp}(p,a) = b$$

Terms up to transport

$$\frac{t \sqsubset t'}{t \sqsubset \mathbf{transp}(e,t')}$$

Terms up to transport

$$\frac{t \sqsubset t'}{t \sqsubset \mathbf{transp}(e,t')} \qquad \frac{t \sqsubset t' \quad A \sqsubset A' \quad B \sqsubset B' \quad u \sqsubset u'}{t\ @^{(x:A).B}\ u \sqsubset t'\ @^{(x:A').B'}\ u'}$$

Terms up to transport

$$\frac{t \sqsubset t'}{t \sqsubset \mathbf{transp}(e,t')} \qquad \frac{t \sqsubset t' \quad A \sqsubset A' \quad B \sqsubset B' \quad u \sqsubset u'}{t\ @^{(x:A).B}\ u \sqsubset t'\ @^{(x:A').B'}\ u'} \qquad \dots$$

Terms up to transport

$$\frac{t \sqsubset t'}{t \sqsubset \mathbf{transp}(e,t')} \qquad \frac{t \sqsubset t' \quad A \sqsubset A' \quad B \sqsubset B' \quad u \sqsubset u'}{t\ @^{(x:A).B}\ u \sqsubset t'\ @^{(x:A').B'}\ u'} \qquad \dots$$

Invariant

`t` is **translated** to `t'` with $t \sqsubset t'$

Terms up to transport

$$\frac{t \sqsubset t'}{t \sqsubset \mathbf{transp}(e,t')} \qquad \frac{t \sqsubset t' \quad A \sqsubset A' \quad B \sqsubset B' \quad u \sqsubset u'}{t\ @^{(x:A).B}\ u \sqsubset t'\ @^{(x:A').B'}\ u'} \qquad \dots$$

Invariant

t is **translated** to t' with $t \sqsubset t'$

Fundamental lemma

Given Γ and $t \sqsupset . \sqsubset t'$, there exists a term p such that
if $\Gamma \vdash_i t : A$ and $\Gamma \vdash_i t' : B$ then $\Gamma \vdash_x p : t\ {}_{A=B}\ t'$.

Translation

$$\text{if } \vdash_x \Gamma \text{ then } \sum \Gamma^t \sqsupset \Gamma, \vdash_i \Gamma^t$$

Translation

$$\texttt{if}\ \vdash_x \Gamma\ \texttt{then}\ \sum \Gamma^t \supseteq \Gamma,\ \vdash_i \Gamma^t$$

$$\texttt{if}\ \Gamma \vdash_x t : A\ \texttt{then}$$
$$\forall\ \Gamma^t \supseteq \Gamma,\ \vdash_i \Gamma^t \to \sum\ (t^t \supseteq t)\ (A^t \supseteq A),\ \Gamma^t \vdash_i t^t : A^t$$

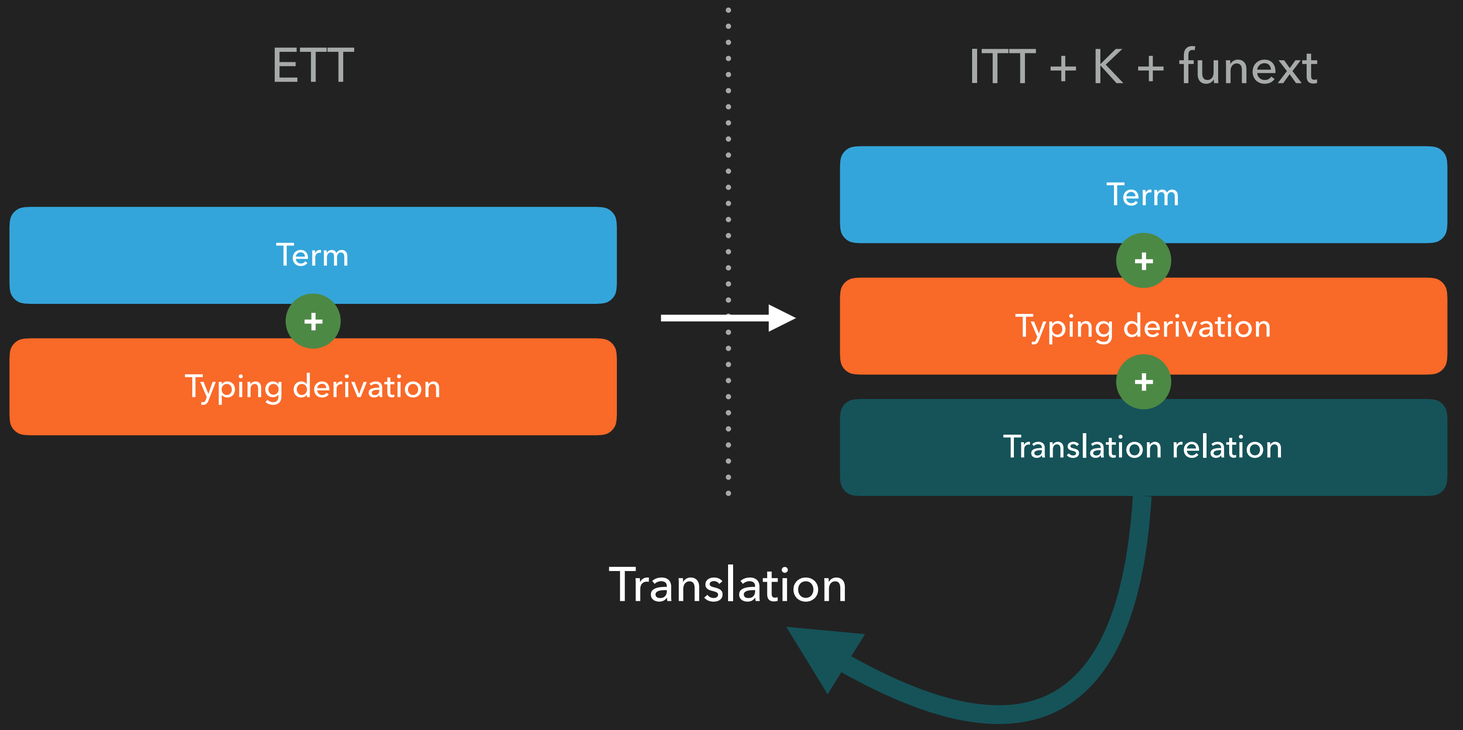
Translation

$$\text{if } \vdash_x \Gamma \text{ then } \sum \Gamma^t \sqsupseteq \Gamma,\ \vdash_i \Gamma^t$$

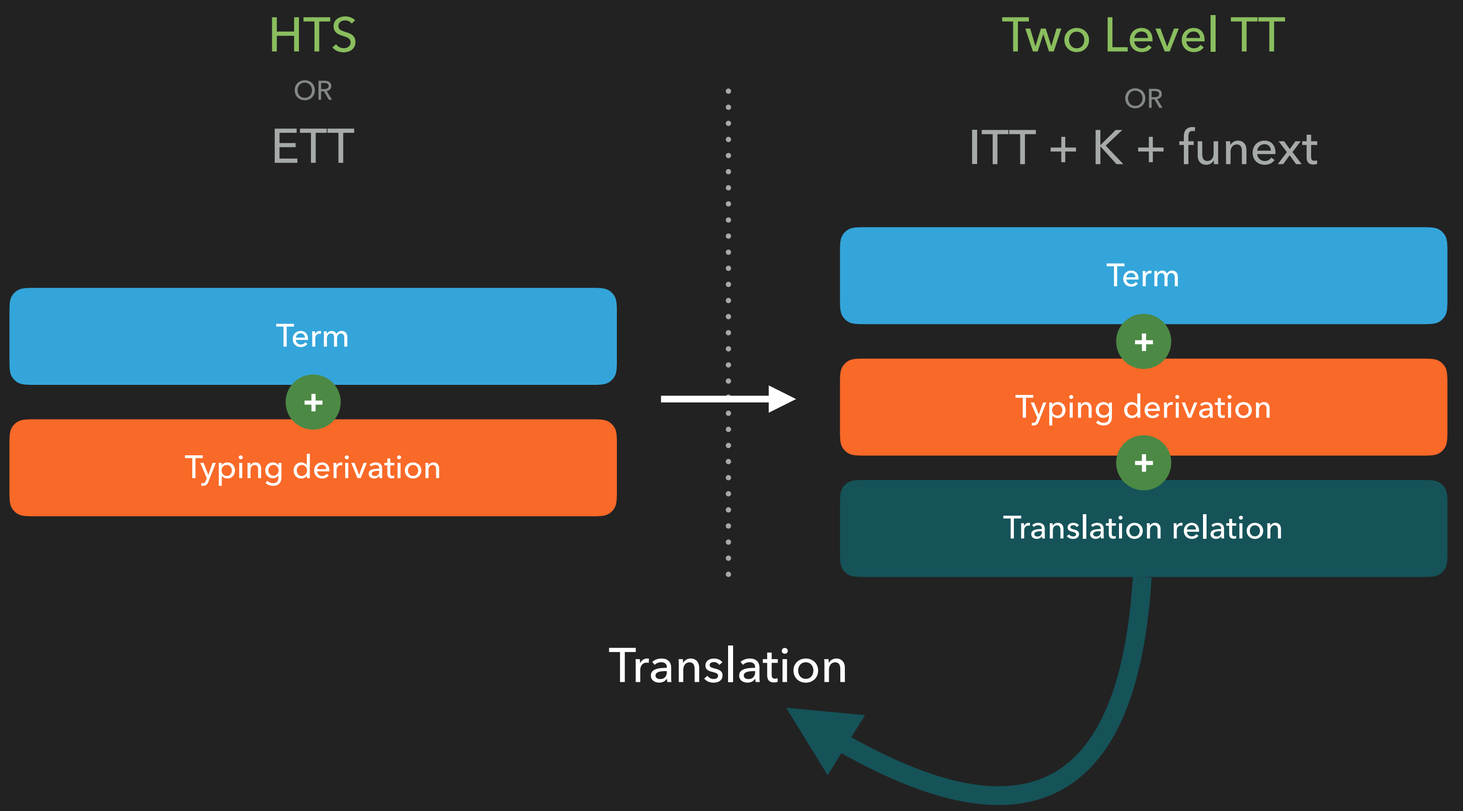
$$\text{if } \Gamma \vdash_x t : A \text{ then}$$
$$\forall\ \Gamma^t \sqsupseteq \Gamma,\ \vdash_i \Gamma^t \to \sum\ (t^t \sqsupseteq t)\ (A^t \sqsupseteq A),\ \Gamma^t \vdash_i t^t : A^t$$

$$\text{if } \Gamma \vdash_x t \equiv u : A \text{ then}$$
$$\forall\ \Gamma^t \sqsupseteq \Gamma,\ \vdash_i \Gamma^t \to\ \sum\ (t^t \sqsupseteq t)\ (A^t \sqsupseteq A)\ (u^t \sqsupseteq u)\ (A^s \sqsupseteq A)\ p,$$
$$\Gamma^t \vdash_i p : t^t\ {}_{A^t = A^s}\ u^t$$

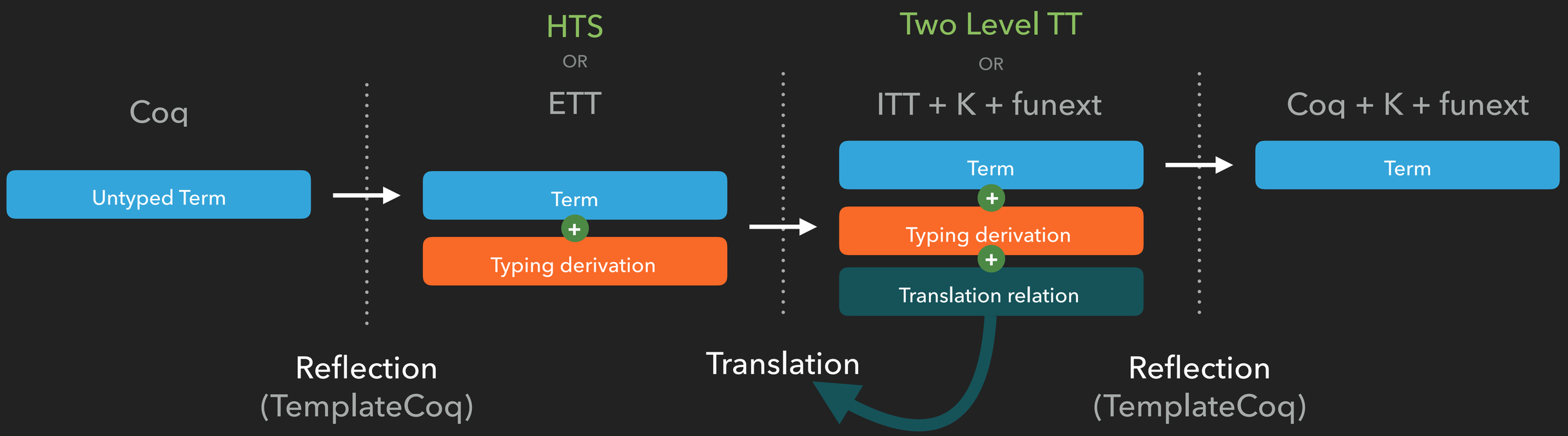
Conclusion


ETT
Term
+
Typing derivation
ITT + K + funext
Term
+
Typing derivation
+
Translation relation
Translation

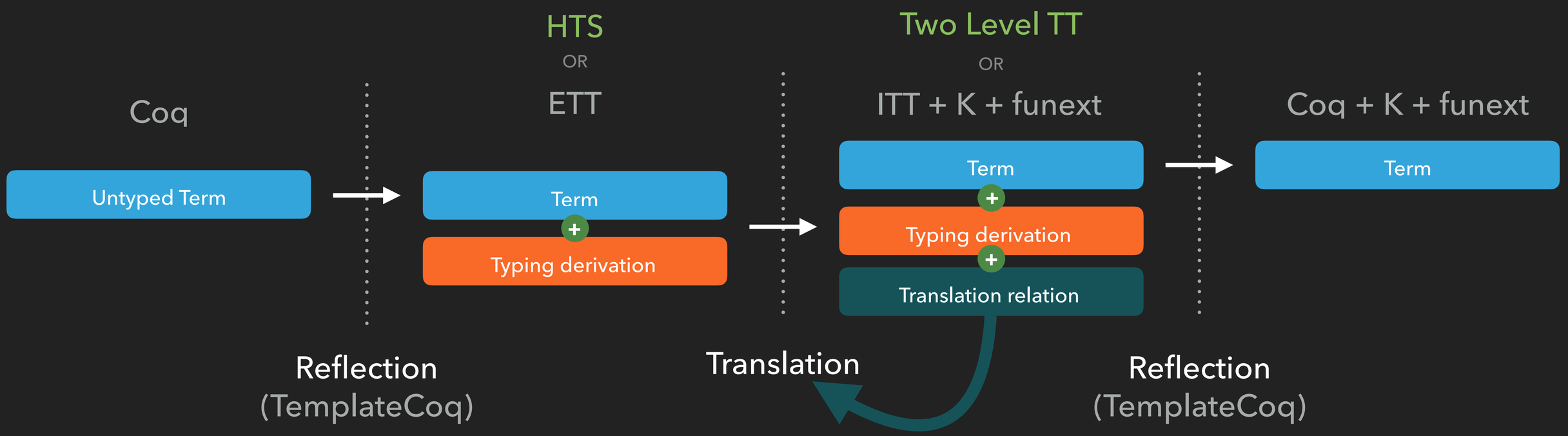
Conclusion


HTS
OR
ETT
Term
+
Typing derivation
Two Level TT
OR
ITT + K + funext
Term
+
Typing derivation
+
Translation relation
Translation

Conclusion


HTS
OR
ETT
Two Level TT
OR
ITT + K + funext
Coq
Coq + K + funext
Untyped Term
Term
+
Typing derivation
Term
+
Typing derivation
+
Translation relation
Term
Reflection
(TemplateCoq)
Translation
Reflection
(TemplateCoq)

Conclusion



https://github.com/TheoWinterhalter/**ett-to-itt**

```
From Template Require Import All.
From Translation Require Import Quotes plugin.
Import MonadNotation.

(*! EXAMPLE 1 *)
(*
   Our first example is the identity with a coercion.
1  As you can see, the definition fails in ITT/Coq.
1  We thus use the notation {! _ !} that allows a term of type A
1  to be given in place of any other type B.
1  This is ignored by the plugin and as such it allows us to write ETT
1  terms directly in Coq.
1 *)
1Fail Definition pseudoid (A B : Type) (e : A = B) (x : A) : B := x.
1Definition pseudoid (A B : Type) (e : A = B) (x : A) : B := {! x !}.
1
1Run TemplateProgram (Translate ε "pseudoid").
2Print pseudoidᵗ.
2
2
2
2
2
2
2
2
2
2
3
3
3
3
```

U:--- *goals* All (1,0) (Coq Goals)

```
From Template Require Import All.
From Translation Require Import Quotes plugin.
Import MonadNotation.

(*! EXAMPLE 1 *)
(*
   Our first example is the identity with a coercion.
   As you can see, the definition fails in ITT/Coq.
   We thus use the notation {! _ !} that allows a term of type A
   to be given in place of any other type B.
   This is ignored by the plugin and as such it allows us to write ETT
   terms directly in Coq.
 *)
Fail Definition pseudoid (A B : Type) (e : A = B) (x : A) : B := x.
Definition pseudoid (A B : Type) (e : A = B) (x : A) : B := {! x !}.

Run TemplateProgram (Translate ε "pseudoid").
Print pseudoidᵗ.
```

U:--- *goals* All (1,0) (Coq Goals)

```
The command has indeed failed
with message: (diff)
In environment
A : Type
B : Type
e : A = B
x : A
The term "x" has type "A"
while it is expected to have
type
"B".
```

```
From Template Require Import All.
From Translation Require Import Quotes plugin.
Import MonadNotation.

(*! EXAMPLE 1 *)
(*
   Our first example is the identity with a coercion.
   As you can see, the definition fails in ITT/Coq.
   We thus use the notation {! _ !} that allows a term of type A
   to be given in place of any other type B.
   This is ignored by the plugin and as such it allows us to write ETT
   terms directly in Coq.
 *)
Fail Definition pseudoid (A B : Type) (e : A = B) (x : A) : B := x.
Definition pseudoid (A B : Type) (e : A = B) (x : A) : B := {! x !}.

Run TemplateProgram (Translate ε "pseudoid").
Print pseudoidᵗ.
```

U:--- *goals* All (1,0) (Coq Goals)

```
pseudoid is defined
```

```
From Template Require Import All.
From Translation Require Import Quotes plugin.
Import MonadNotation.

(*! EXAMPLE 1 *)
(*
   Our first example is the identity with a coercion.
   As you can see, the definition fails in ITT/Coq.
   We thus use the notation {! _ !} that allows a term of type A
   to be given in place of any other type B.
   This is ignored by the plugin and as such it allows us to write ETT
   terms directly in Coq.
 *)
Fail Definition pseudoid (A B : Type) (e : A = B) (x : A) : B := x.
Definition pseudoid (A B : Type) (e : A = B) (x : A) : B := {! x !}.

Time Run TemplateProgram (Translate ε "pseudoid").
Print pseudoidᵗ.
```

```
U:---  *goals*        All (1,0)      (Coq Goals)
pseudoid_obligation_0 has type-checked,
generating 1 obligation
Solving obligations automatically...
pseudoid_obligation_0_obligation_1 is
defined
No more obligations remaining
pseudoid_obligation_0 is defined
"Successfully generated pseudoidᵗ"

<infomsg>
Finished transaction in 0.446 secs
(0.419u,0.022s) (successful)
</infomsg>
```

```
From Template Require Import All.
From Translation Require Import Quotes plugin.
Import MonadNotation.

(*! EXAMPLE 1 *)
(*
   Our first example is the identity with a coercion.
   As you can see, the definition fails in ITT/Coq.
   We thus use the notation {! _ !} that allows a term of type A
   to be given in place of any other type B.
   This is ignored by the plugin and as such it allows us to write ET
T
   terms directly in Coq.
 *)
Fail Definition pseudoid (A B : Type) (e : A = B) (x : A) : B := x.
Definition pseudoid (A B : Type) (e : A = B) (x : A) : B := {! x !}.

Time Run TemplateProgram (Translate ε "pseudoid").
Print pseudoidᵗ.
```

U:--- *goals* All (1,0) (Coq Goals)

```
pseudoidᵗ =
λ (A B : Type) (e : A = B) (x : A) ⇒
transport (pseudoid_obligation_0 A B e x) x
     : ∀ A B : Type, A = B → A → B

Argument scopes are [type_scope type_scope _ _]
```

```
   Inductive types.

   For this we take a look at the type of vectors.
   In order to translate an element of vec A n, we first need to add
   vec (and nat) to the context.
 *)
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons : A → ∀ n, vec A n → vec A (S n).

Arguments vnil {_}.
Arguments vcons {_} _ _ _.

Definition vv := vcons 1 _ vnil.
Time Run TemplateProgram (
      θ ← TranslateConstant ε "nat" ;;
      θ ← TranslateConstant θ "vec" ;;
      Translate θ "vv"
).
Print vvᵗ.
```

```
U:---  *goals*        All (1,0)      (Coq Goals)
```

```
   Inductive types.

   For this we take a look at the type of vectors.
   In order to translate an element of vec A n, we first need to add
   vec (and nat) to the context.
 *)
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons : A → ∀ n, vec A n → vec A (S n).

Arguments vnil {_}.
Arguments vcons {_} _ _ _.

Definition vv := vcons 1 _ vnil.
Time Run TemplateProgram (
      θ ← TranslateConstant ε "nat" ;;
      θ ← TranslateConstant θ "vec" ;;
      Translate θ "vv"
).
Print vvᵗ.
```

```
U:---  *goals*        All (1,0)     (Coq Goals)
"Successfully generated vvᵗ"

<infomsg>
Finished transaction in 2.579 secs
(2.497u,0.061s) (successful)
</infomsg>
```

```
(*
   Inductive types.

   For this we take a look at the type of vectors.
   In order to translate an element of vec A n, we first need to add
   vec (and nat) to the context.
 *)
Inductive vec A : ℕ → Type :=
| vnil : vec A 0
| vcons : A → ∀ n, vec A n → vec A (S n).

Arguments vnil {_}.
Arguments vcons {_} _ _ _.

Definition vv := vcons 1 _ vnil.
Time Run TemplateProgram (
      θ ← TranslateConstant ε "nat" ;;
      θ ← TranslateConstant θ "vec" ;;
      Translate θ "vv"
).
Print vvᵗ.
```

U:--- *goals* All (1,0) (Coq Goals)

```
vvᵗ = vcons 1 0 vnil
     : vec ℕ 1
```

```
(*
   Reversal of vectors.

   Our plugin doesn't handle fixpoint and pattern-matching so we need
   to write our function with eliminators.
   Same as before we need to add the eliminator (as well as addition
on natural
   numbers) to the context.
*)

Fail Definition vrev {A n m} (v : vec A n) (acc : vec A m) : vec A (n
  + m) :=
  vec_rect A (λ n _ ⇒ ∀ m, vec A m → vec A (n + m))
           (λ m acc ⇒ acc) (λ a n _ rv m acc ⇒ rv _ (vcons a m acc))
           n v m acc.

Definition vrev {A n m} (v : vec A n) (acc : vec A m) : vec A (n + m)
  :=
  vec_rect A (λ n _ ⇒ ∀ m, vec A m → vec A (n + m))
           (λ m acc ⇒ acc) (λ a n _ rv m acc ⇒ {! rv _ (vcons a m acc
) !})
           n v m acc.

Time Run TemplateProgram (
      θ ← TranslateConstant ε "nat" ;;
      θ ← TranslateConstant θ "vec" ;;
      θ ← TranslateConstant θ "Nat.add" ;;
      θ ← TranslateConstant θ "vec_rect" ;;
      Translate θ "vrev"
).
Print vrevᵗ.
```

U:--- *goals* All (1,0) (Coq Goals)

```
vvᵗ = vcons 1 0 vnil
     : vec ℕ 1
```

```
  Reversal of vectors.

  Our plugin doesn't handle fixpoint and pattern-matching so we need
  to write our function with eliminators.
  Same as before we need to add the eliminator (as well as addition on natural
  numbers) to the context.
*)

Fail Definition vrev {A n m} (v : vec A n) (acc : vec A m) : vec A (n + m) :=
  vec_rect A (λ n _ ⇒ ∀ m, vec A m → vec A (n + m))
           (λ m acc ⇒ acc) (λ a n _ rv m acc ⇒ rv _ (vcons a m acc))
           n v m acc.

Definition vrev {A n m} (v : vec A n) (acc : vec A m) : vec A (n + m) :=
  vec_rect A (λ n _ ⇒ ∀ m, vec A m → vec A (n + m))
           (λ m acc ⇒ acc) (λ a n _ rv m acc ⇒ {! rv _ (vcons a m acc) !})
           n v m acc.

Time Run TemplateProgram (
     θ ← TranslateConstant ε "nat" ;;
     θ ← TranslateConstant θ "vec" ;;
     θ ← TranslateConstant θ "Nat.add" ;;
     θ ← TranslateConstant θ "vec_rect" ;;
     Translate θ "vrev"
).
Print vrevᵗ.
```

```
U:---  *goals*       All (1,0)      (Coq Goals)
The command has indeed failed with
message: (diff)
In environment
A : Type
n : ℕ
m : ℕ
v : vec A n
acc : vec A m
a : A
n0 : ℕ
v0 : vec A n0
rv :
∀ m : ℕ, vec A m → vec A (n0 + m)
m0 : ℕ
acc0 : vec A m0
The term "rv (S m0) (vcons a m0 acc0)"
has type "vec A (n0 + S m0)"
while it is expected to have type
 "vec A (S n0 + m0)".
```

```
(*
   Reversal of vectors.

   Our plugin doesn't handle fixpoint and pattern-matching so we need
   to write our function with eliminators.
   Same as before we need to add the eliminator (as well as addition on natural
   numbers) to the context.
 *)

Fail Definition vrev {A n m} (v : vec A n) (acc : vec A m) : vec A (n + m) :=
  vec_rect A (λ n _ ⇒ ∀ m, vec A m → vec A (n + m))
           (λ m acc ⇒ acc) (λ a n _ rv m acc ⇒ rv _ (vcons a m acc))
           n v m acc.

Definition vrev {A n m} (v : vec A n) (acc : vec A m) : vec A (n + m) :=
  vec_rect A (λ n _ ⇒ ∀ m, vec A m → vec A (n + m))
           (λ m acc ⇒ acc) (λ a n _ rv m acc ⇒ {! rv _ (vcons a m acc) !})
           n v m acc.

Time Run TemplateProgram (
      θ ← TranslateConstant ε "nat" ;;
      θ ← TranslateConstant θ "vec" ;;
      θ ← TranslateConstant θ "Nat.add" ;;
      θ ← TranslateConstant θ "vec_rect" ;;
      Translate θ "vrev"
).
Print vrevᵗ.
```

U:--- *goals* All (1,0) (Coq Goals)

vrev is defined

```
   numbers) to the context.
*)

Fail Definition vrev {A n m} (v : vec A n) (acc : vec A m) : vec A (n + m) :=
  vec_rect A (λ n _ ⇒ ∀ m, vec A m → vec A (n + m))
           (λ m acc ⇒ acc) (λ a n _ rv m acc ⇒ rv _ (vcons a m acc))
           n v m acc.

Definition vrev {A n m} (v : vec A n) (acc : vec A m) : vec A (n + m) :=
  vec_rect A (λ n _ ⇒ ∀ m, vec A m → vec A (n + m))
           (λ m acc ⇒ acc) (λ a n _ rv m acc ⇒ {! rv _ (vcons a m acc) !})
           n v m acc.

Time Run TemplateProgram (
      θ ← TranslateConstant ε "nat" ;;
      θ ← TranslateConstant θ "vec" ;;
      θ ← TranslateConstant θ "Nat.add" ;;
      θ ← TranslateConstant θ "vec_rect" ;;
      Translate θ "vrev"
).
Print vrevᵗ.
```

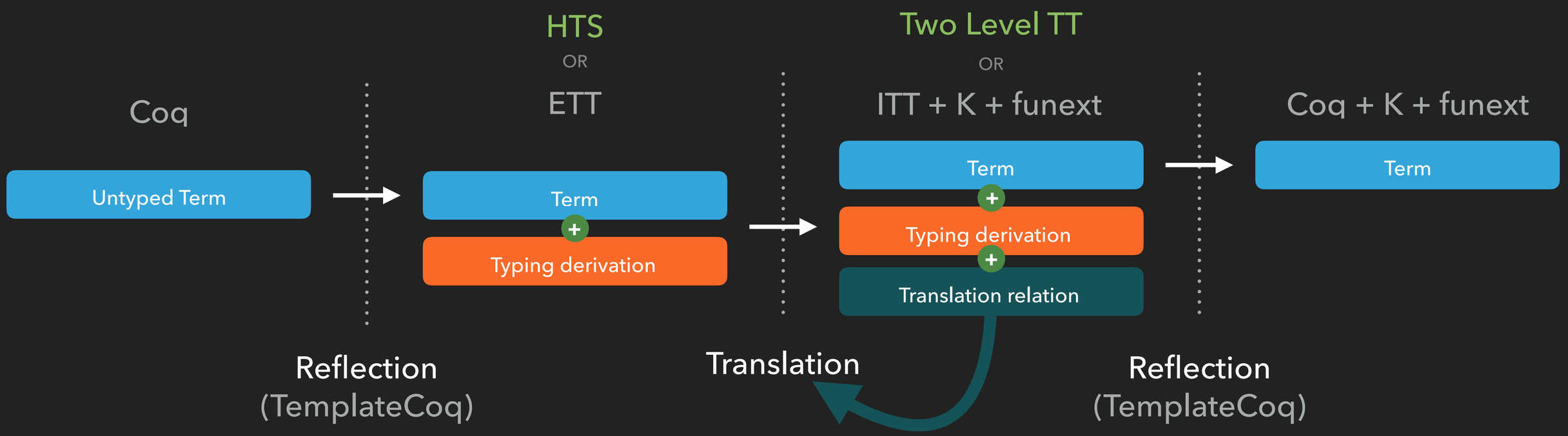
```
U:---  *goals*       All (1,0)     (Coq Goals)
vrev_obligation_0_obligation_1 is defined
No more obligations remaining
vrev_obligation_0 is defined
vrev_obligation_1 has type-checked, generating 1 obligation
Solving obligations automatically...
vrev_obligation_1_obligation_1 is defined
No more obligations remaining
vrev_obligation_1 is defined
vrev_obligation_2 has type-checked, generating 1 obligation
Solving obligations automatically...
vrev_obligation_2_obligation_1 is defined
No more obligations remaining
vrev_obligation_2 is defined
vrev_obligation_3 has type-checked, generating 1 obligation
Solving obligations automatically...
vrev_obligation_3_obligation_1 is defined
No more obligations remaining
vrev_obligation_3 is defined
vrev_obligation_4 has type-checked, generating 1 obligation
Solving obligations automatically...
vrev_obligation_4_obligation_1 is defined
No more obligations remaining
vrev_obligation_4 is defined
"Successfully generated vrevᵗ"

<infomsg>
Finished transaction in 117.92 secs (113.791u,1.98s)
```

```
v : vec A n) (acc : vec A m) :
 vec A (n + m) :=
  vec_rect A (λ n _ ⇒ ∀ m, vec
 A m → vec A (n + m))
           (λ m acc ⇒ acc) (λ
a n _ rv m acc ⇒ rv _ (vcons a
 m acc))
           n v m acc.

Definition vrev {A n m} (v : v
ec A n) (acc : vec A m) : vec
A (n + m) :=
  vec_rect A (λ n _ ⇒ ∀ m, vec
 A m → vec A (n + m))
           (λ m acc ⇒ acc) (λ
a n _ rv m acc ⇒ {! rv _ (vcon
s a m acc) !})
           n v m acc.

Time Run TemplateProgram (
      Θ ← TranslateConstant ε "
nat" ;;
      Θ ← TranslateConstant Θ
"vec" ;;
      Θ ← TranslateConstant Θ
"Nat.add" ;;
      Θ ← TranslateConstant Θ
"vec_rect" ;;
      Translate Θ "vrev"
).
Print vrev^t.
```

U:--- *goals* All (1,0) (Coq Goals)

```
vrev^t =
λ (A : Type) (n m : ℕ) (v : vec A n) (acc : vec A m) ⇒
vec_rect A (λ (n₀ : ℕ) (_ : vec A n₀) ⇒ ∀ m₀ : ℕ, vec A m₀ → vec A (n₀ + m₀))
  (λ (m₀ : ℕ) (acc₀ : vec A m₀) ⇒ acc₀)
  (λ (a : A) (n₀ : ℕ) (v₀ : vec A n₀)
     (rv : (λ (n₁ : ℕ) (_ : vec A n₁) ⇒ ∀ m₀ : ℕ, vec A m₀ → vec A (n₁ + m₀))
            n₀ v₀) (m₀ : ℕ) (acc₀ : vec A m₀) ⇒
   transport (vrev_obligation_3 A n m v acc a n₀ v₀ rv m₀ acc₀)
     (rv (S m₀) (vcons a m₀ acc₀))) n v m acc
     : ∀ (A : Type) (n m : ℕ), vec A n → vec A m → vec A (n + m)

Argument scopes are [type_scope nat_scope nat_scope _ _]
```

Conclusion



https://github.com/TheoWinterhalter/**ett-to-itt**