Proof Repair across Type Equivalences

2021 PLDI-Talia Ringer、RanDair Porter、Nathaniel Yazdani

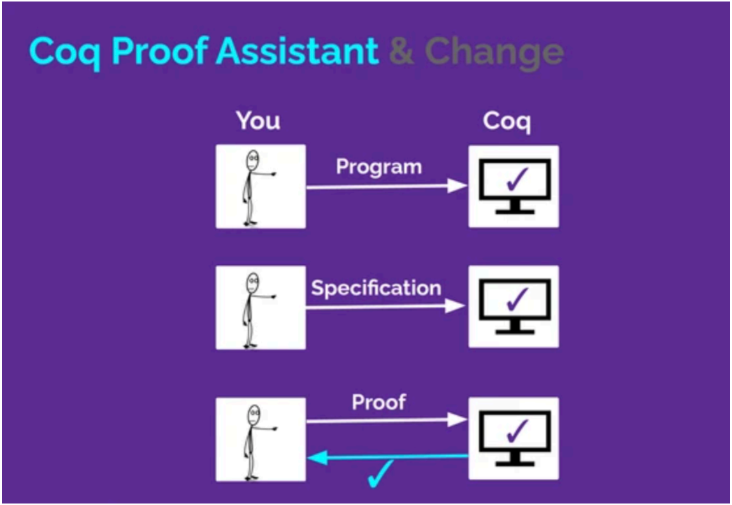
Speaker: Chi Feng

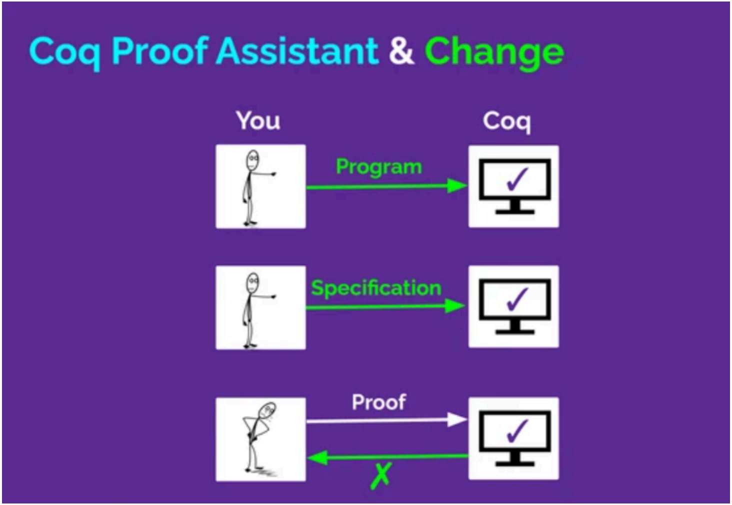
2022 年 4 月 8 日

SHANGHAI JIAO TONG UNIVERSITY

Catalogue

Catalogue


Coq Proof Assistant & Change
You
Coq
Program
Specification
Proof


Coq Proof Assistant & Change
You
Coq
Program
Specification
Proof

"There is **no reason to believe** that **verifying a modified program** is any easier than **verifying the original the first time around**."

Problem:
changes in datatypes.

Coq's Workflow

The typical proof engineering workflow in Coq is interactive:

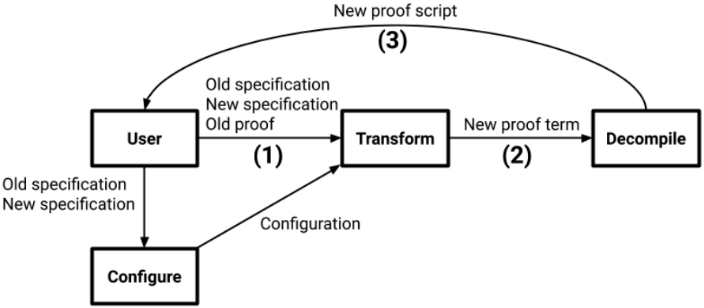
- **High-level**: tactics, proof scripts
- **Low-level**: proof term

Works at low-level, then builds back up to high-level.

Pumpkin Pi's Workflow

When the proof engineer invokes Pumpkin Pi:

- **Configure**
- **Transform**
- **Decompile**



Catalogue

A Simple Motivating Example

- Swap the two constructors of the list datatype.(left-old, right-new)

```
Inductive list (T : Type) : Type :=
| nil : list T
| cons : T → list T → list T.
```

```
Inductive list (T : Type) : Type :=
| cons : T → list T → list T
| nil : list T.
```

- The proof scripts of lemma rev_app_distr.

```
Lemma rev_app_distr {A} :
  ∀ (x y : list A), rev (x ++ y) = rev y ++ rev x.
Proof. (* by induction over x and y *)
  induction x as [| a l IHl].
  (* x nil: *) induction y as [| a l IHl].
  (* y nil: *) simpl. auto.
  (* y cons *) simpl. rewrite app_nil_r; auto.
  (* both cons: *) intro y. simpl.
  rewrite (IHl y). rewrite app_assoc; trivial.
Qed.
```

```
Proof. (* by induction over x and y *)
  intros x. induction x as [a l IHl| ]; intro y0.
  - (* both cons: *) simpl. rewrite IHl. simpl.
    rewrite app_assoc. auto.
  - (* x nil: *) induction y0 as [a l H| ].
    + (* y cons: *) simpl. rewrite app_nil_r. auto.
    + (* y nil: *) auto.
Qed.
```

Catalogue

Scope: Type Equivalences

- Given an equivalence between types A and B, Pumpkin Pi repairs functions and proofs defined over A to instead refer to B.
- Types A and B exists a type equivalence, those types are equivalent ($A \simeq B$).
- **Example:** two functions (top) form an equivalence (bottom).

```
swap T (l : Old.list T) : New.list T :=
  Old.list_rect T (fun (l : Old.list T) => New.list T)
    New.nil
    (fun t _ (IHl : New.list T) => New.cons T t IHl)
    l.

Lemma section: ∀ T (l : Old.list T),
  swap⁻¹ T (swap T l) = l.
Proof.
  intros T l. symmetry. induction l as [ |a l0 H].
  - auto.
  - simpl. rewrite ← H. auto.
Qed.
```

```
swap⁻¹ T (l : New.list T) : Old.list T :=
  New.list_rect T (fun (l : New.list T) => Old.list T)
    (fun t _ (IHl : Old.list T) => Old.cons T t IHl)
    Old.nil
    l.

Lemma retraction: ∀ T (l : New.list T),
  swap T (swap⁻¹ T l) = l.
Proof.
  intros T l. symmetry. induction l as [t l0 H| ].
  - simpl. rewrite ← H. auto.
  - auto.
Qed.
```

Scope: Type Equivalences

Two changes below are described equivalences:

- Factoring out Constructors.

```
Inductive I :=
| A : I
| B : I.
```

```
Inductive J :=
| makeJ : bool → J.
```

- Adding a Dependent Index.

```
Inductive list (T : Type) : Type :=
| nil : list T
| cons : T → list T → list T.
```

```
Inductive vector (T : Type) : nat → Type :=
| nil : vector T O
| cons : T → ∀ (n : nat), vector T n → vector T (S n).
```

Goal: Transport with a Twist

- The goal of Pumpkin Pi is to implement proof reuse: transport.
- **Transport**: takes a term t and produces a term t' (the same as t modulo an equivalence A $\simeq$ B).
- When transport across A $\simeq$ B takes t to t' , t and t' are equal up to transport across that equivalence ($t \equiv_{A \simeq B} t'$).

Catalogue

Transformation

- Pumpkin Pi's heart: a configurable proof term transformation for transporting proofs across equivalences.
- CoC: a variant of the lambda calculus with polymorphism and dependent types.
- CIC_ω: extends CoC with inductive types.
- Inductive types: are defined by constructors and eliminators.
- The syntax for CIC_ω with primitive eliminators.

$$\langle i\rangle \in \mathbb{N},\ \langle v\rangle \in \text{Vars},\ \langle s\rangle \in \{\,\text{Prop}, \text{Set}, \text{Type}\langle i\rangle\,\}$$

$$\langle t\rangle ::= \langle v\rangle \mid \langle s\rangle \mid \Pi\,(\langle v\rangle : \langle t\rangle)\,.\,\langle t\rangle \mid \lambda\,(\langle v\rangle : \langle t\rangle)\,.\,\langle t\rangle \mid \langle t\rangle\,\langle t\rangle \mid \text{Ind}\,(\langle v\rangle : \langle t\rangle)\{\langle t\rangle,\ldots,\langle t\rangle\} \mid \text{Constr}\,(\langle i\rangle, \langle t\rangle) \mid \text{Elim}(\langle t\rangle, \langle t\rangle)\{\langle t\rangle,\ldots,\langle t\rangle\}$$

Configuration

- Configuration corresponds to an equivalence A $\simeq$ B.
- Configuration helps the transformation achieve two goals:
  - Preserve the equivalence.
  - Produce well-typed terms.
- Configuration: ((DepConstr, DepElim), (Eta, Iota))

Preserving the Equivalence.

- For the list change in Page 10, the configuration uses the dependent constructors and eliminators(swap):

```
DepConstr(0, list T) : list T := Constr(0, list T).
DepConstr(1, list T) t l : list T :=
  Constr (1, list T) t l.

DepElim(l, P) { p_nil, p_cons } : P l :=
  Elim(l, P) { p_nil, p_cons }.
```

```
DepConstr(0, list T) : list T := Constr(1, list T).
DepConstr(1, list T) t l : list T :=
  Constr(0, list T) t l.

DepElim(l, P) { p_nil, p_cons } : P l :=
  Elim(l, P) { p_cons, p_nil }.
```

- For the type of vectors of some length(map):

$$\mathtt{packed\_vect\ T} := \Sigma(\mathtt{n} : \mathtt{nat}).\mathtt{vector\ T\ n}.$$

```
DepConstr(0, packed_vect) : packed_vect T :=
  ∃ (Constr(0, nat)) (Constr(0, vector T)).

DepElim(s, P) { f₀ f₁ } : P (∃ (π_l s) (π_r s)) :=
  Elim(π_r s, λ(n : nat)(v : vector T n).P (∃ n v)) {
    f₀,
    (λ(t : T)(n : nat)(v : vector T n).f₁ t (∃ n v))
  }.
```

Producing Well-Typed Terms.

- Two kinds of equality:
  - Hold by reduction (definitional equality).
  - Hold by proof (propositional equality).
- **Definitionally equal:** two terms t and t' of type T reduce to the same normal form.
- **Propositionally equal:** there is a proof that t = t' using the inductive equality type = at type T.
- Eta and Iota define η-expansion and ι-reduction of A and B.
  - η-**expansion:** describes how to expand a term to apply a constructor to an eliminator.
  - ι-**reduction:** describes how to reduce an elimination of a constructor.

Producing Well-Typed Terms.

- Unary numbers nat to binary numbers N:

```
Inductive positive :=
| xI : positive → positive
| xO : positive → positive
| xH : positive.
```

```
Inductive nat :=
| O : nat
| S : nat → nat.
```

```
Inductive N :=
| N0 : N
| Npos : positive → N.
```

nat

- ι-reduction for nat is definitional.

```
∀ P p0 pS n,
  DepElim(DepConstr(1, nat) n, P) { p0, pS } =
  pS n (DepElim(n, P) { p0, pS }).
```

- Iota is a rewrite by that proof by reflexivity.

```
∀ P p0 pS n (Q: P (DepConstr(1, nat) n) → s),
  Iota(1, nat, Q) :
    Q (pS n (DepElim(n, P) { p0, pS })) →
    Q (DepElim(DepConstr(1, nat) n, P) { p0, pS }).
```

Producing Well-Typed Terms.

N

- ι-reduction for N is propositional.

```
∀ P p0 pS n,
  DepElim(DepConstr(1, N) n, P) { p0, pS } =
  pS n (DepElim(n, P) { p0, pS }).
```

- Iota is a rewrite by the propositional equality.

```
∀ P p0 pS n (Q: P (DepConstr(1, N) n) → s),
  Iota(1, N, Q) :
    Q (pS n (DepElim(n, P) { p0, pS })) →
    Q (DepElim(DepConstr(1, N) n, P) { p0, pS }).
```

The Proof Term Transformation

- Transformation $\Gamma \vdash t \Uparrow t'$ for transporting terms across A $\simeq$ B with configuration:

$$\boxed{\Gamma \vdash t \Uparrow t'}$$

$$\textsc{Dep-Elim}\ \frac{\Gamma \vdash a \Uparrow b \qquad \Gamma \vdash p_a \Uparrow p_b \qquad \Gamma \vdash \vec{f_a} \Uparrow \vec{f_b}}{\Gamma \vdash \mathrm{DepElim}(a,\ p_a)\vec{f_a} \Uparrow \mathrm{DepElim}(b,\ p_b)\vec{f_b}}$$

$$\textsc{Dep-Constr}\ \frac{\Gamma \vdash \vec{t_a} \Uparrow \vec{t_b}}{\Gamma \vdash \mathrm{DepConstr}(j,\ A)\ \vec{t_a} \Uparrow \mathrm{DepConstr}(j,\ B)\ \vec{t_b}}$$

$$\textsc{Eta}\ \frac{}{\Gamma \vdash \mathrm{Eta}(A) \Uparrow \mathrm{Eta}(B)}$$

$$\textsc{Iota}\ \frac{\Gamma \vdash q_A \Uparrow q_B \qquad \Gamma \vdash \vec{t_A} \Uparrow \vec{t_B}}{\Gamma \vdash \mathrm{Iota}(j,\ A,\ q_A)\ \vec{t_A} \Uparrow \mathrm{Iota}(j,\ B,\ q_B)\ \vec{t_B}}$$

$$\textsc{Equivalence}\ \frac{}{\Gamma \vdash A \Uparrow B}$$

$$\textsc{Constr}\ \frac{\Gamma \vdash T \Uparrow T' \qquad \Gamma \vdash \vec{t} \Uparrow \vec{t'}}{\Gamma \vdash \mathrm{Constr}(j,\ T)\ \vec{t} \Uparrow \mathrm{Constr}(j,\ T')\ \vec{t'}}$$

$$\textsc{Ind}\ \frac{\Gamma \vdash T \Uparrow T' \qquad \Gamma \vdash \vec{C} \Uparrow \vec{C'}}{\Gamma \vdash \mathrm{Ind}(Ty : T)\vec{C} \Uparrow \mathrm{Ind}(Ty : T')\vec{C'}}$$

$$\textsc{App}\ \frac{\Gamma \vdash f \Uparrow f' \qquad \Gamma \vdash t \Uparrow t'}{\Gamma \vdash f t \Uparrow f' t'}$$

$$\textsc{Elim}\ \frac{\Gamma \vdash c \Uparrow c' \qquad \Gamma \vdash Q \Uparrow Q' \qquad \Gamma \vdash \vec{f} \Uparrow \vec{f'}}{\Gamma \vdash \mathrm{Elim}(c, Q)\vec{f} \Uparrow \mathrm{Elim}(c', Q')\vec{f'}}$$

$$\textsc{Lam}\ \frac{\Gamma \vdash t \Uparrow t' \qquad \Gamma \vdash T \Uparrow T' \qquad \Gamma,\ t : T \vdash b \Uparrow b'}{\Gamma \vdash \lambda(t : T).b \Uparrow \lambda(t' : T').b'}$$

$$\textsc{Prod}\ \frac{\Gamma \vdash t \Uparrow t' \qquad \Gamma \vdash T \Uparrow T' \qquad \Gamma,\ t : T \vdash b \Uparrow b'}{\Gamma \vdash \Pi(t : T).b \Uparrow \Pi(t' : T').b'}$$

$$\textsc{Var}\ \frac{v \in \mathrm{Vars}}{\Gamma \vdash v \Uparrow v}$$

- Pumpkin Pi uses unification to get real proof terms before transformation.

Unification

The list in Page 10 of the append function:

- unmodified
- unified with the configuration
- ported to the updated type
- reduced to the output.

```
(* 1: original term *)
λ (T : Type) (l m : Old.list T) .
 Elim(l, λ(l: Old.list T).Old.list T → Old.list T)) {
   (λ m . m),
   (λ t _ IHl m . Constr(1, Old.list T) t (IHl m))
 } m.

(* 2: after unifying with configuration *)
λ (T : Type) (l m : A) .
 DepElim(l, λ(l: A).A → A)) {
   (λ m . m)
   (λ t _ IHl m . DepConstr(1, A) t (IHl m))
 } m.
```

```
(* 4: reduced to final term *)
λ (T : Type) (l m : New.list T) .
 Elim(l, λ(l: New.list T).New.list T → New.list T)) {
   (λ t _ IHl m . Constr(0, New.list T) t (IHl m)),
   (λ m . m)
 } m.

(* 3: after transforming *)
λ (T : Type) (l m : B) .
 DepElim(l, λ(l: B).B → B)) {
   (λ m . m)
   (λ t _ IHl m . DepConstr(1, B) t (IHl m))
 } m.
```

Specifying Correct Configurations

- Many equivalences $\Longrightarrow$ a change, many configurations $\Longrightarrow$ an equivalence.
- f: the function that uses DepElim to eliminate A and DepConstr to construct B, (g : similar to f).
- These are the correctness criteria for the configuration(preserves equivalence coherently with equality).

section: $\forall$ (a : A), g (f a) = a.
retraction: $\forall$ (b : B), f (g b) = b.

constr_ok: $\forall\ j\ \vec{x_A}\ \vec{x_B},\ \vec{x_A} \equiv_{A\simeq B} \vec{x_B} \rightarrow$
  DepConstr(j, A) $\vec{x_A} \equiv_{A\simeq B}$ DepConstr(j, B) $\vec{x_B}$.

elim_ok: $\forall$ a b P_A P_B $\vec{f_A}$ $\vec{f_B}$,
  a $\equiv_{A\simeq B}$ b $\rightarrow$
  $\mathrm{P}_A \equiv_{(A\rightarrow s)\simeq(B\rightarrow s)} \mathrm{P}_B \rightarrow$
  $\forall\ j,\ \vec{f_A}[\mathrm{j}] \equiv_{\xi(A,P_A,j)\simeq\xi(B,P_B,j)} \vec{f_B}[\mathrm{j}] \rightarrow$
  DepElim(a, P_A) $\vec{f_A} \equiv_{(Pa)\simeq(Pb)}$ DepElim(b, P_B) $\vec{f_A}$.

Specifying Correct Configurations

- Equivalence: section、retraction、constr_ok、elim_ok.
- Correctness: completeness、soundness.

section: $\forall$ (a : A), g (f a) = a.
retraction: $\forall$ (b : B), f (g b) = b.

constr_ok: $\forall\ j\ \vec{x_A}\ \vec{x_B},\ \vec{x_A} \equiv_{A\simeq B} \vec{x_B} \rightarrow$
  DepConstr(j, A) $\vec{x_A} \equiv_{A\simeq B}$ DepConstr(j, B) $\vec{x_B}$.

elim_ok: $\forall$ a b P_A P_B $\vec{f_A}$ $\vec{f_B}$,
  a $\equiv_{A\simeq B}$ b $\rightarrow$
  $\mathrm{P}_A \equiv_{(A\rightarrow s)\simeq(B\rightarrow s)} \mathrm{P}_B \rightarrow$
  $\forall\ j,\ \vec{f_A}$[j] $\equiv_{\xi(A,P_A,j)\simeq\xi(B,P_B,j)}$ $\vec{f_B}$[j]$\rightarrow$
  DepElim(a, P_A) $\vec{f_A} \equiv_{(Pa)\simeq(Pb)}$ DepElim(b, P_B) $\vec{f_A}$.

Catalogue

Decompiling Proof Terms to Tactics

Decompile achieves this in two passes:

- Decompile proof terms to scripts using a predefined set of tactics.
- Improve on suggested tactics.

Basic Proof Scripts

- A mini decompiler: takes CIC_ω terms and produces tactics in a mini version of Ltac(Qtac).
- Qtac syntax:

$\langle v \rangle \in$ Vars, $\langle t \rangle \in \mathrm{CIC}_\omega$

$\langle p \rangle ::=$ intro $\langle v \rangle$ | rewrite $\langle t \rangle \langle t \rangle$ | symmetry | apply $\langle t \rangle$ | induction $\langle t \rangle \langle t \rangle \{ \langle p \rangle, \ldots, \langle p \rangle \}$ | split $\{ \langle p \rangle, \langle p \rangle \}$ | left | right | $\langle p \rangle . \langle p \rangle$

Decompiling Proof Terms to Tactics

- Qtac decompiler($\Gamma \vdash t \Longrightarrow p$) semantics:

$$\boxed{\Gamma \vdash t \Rightarrow p}$$

$$\textsc{Intro}\quad \frac{\Gamma,\ n : T \vdash b \Rightarrow p}{\Gamma \vdash \lambda(n : T).b \Rightarrow \text{intro}\ n.\ p}$$

$$\textsc{Symmetry}\quad \frac{\Gamma \vdash H \Rightarrow p}{\Gamma \vdash \texttt{eq\_sym}\ H \Rightarrow \text{symmetry}.\ p}$$

$$\textsc{Split}\quad \frac{\Gamma \vdash l \Rightarrow p \qquad \Gamma \vdash r \Rightarrow q}{\Gamma \vdash \text{Constr}(0,\ \wedge)\ l\ r \Rightarrow \text{split}\{p, q\}.}$$

$$\textsc{Left}\quad \frac{\Gamma \vdash H \Rightarrow p}{\Gamma \vdash \text{Constr}(0,\ \vee)\ H \Rightarrow \text{left}.\ p}$$

$$\textsc{Right}\quad \frac{\Gamma \vdash H \Rightarrow p}{\Gamma \vdash \text{Constr}(1,\ \vee)\ H \Rightarrow \text{right}.\ p}$$

$$\textsc{Rewrite}\quad \frac{\Gamma \vdash H_1 : x = y \qquad \Gamma \vdash H_2 \Rightarrow p}{\Gamma \vdash \text{Elim}(H_1,\ P)\{x,\ H_2,\ y\} \Rightarrow \text{symmetry}.\ \text{rewrite}\ P\ H_1.\ p}$$

$$\textsc{Induction}\quad \frac{\Gamma \vdash \vec{f} \Rightarrow \vec{p}}{\Gamma \vdash \text{Elim}(t,\ P)\ \vec{f} \Rightarrow \text{induction}\ P\ t\ \vec{p}}$$

$$\textsc{Apply}\quad \frac{\Gamma \vdash t \Rightarrow p}{\Gamma \vdash f t \Rightarrow \text{apply}\ f.\ p}$$

$$\textsc{Base}\quad \frac{}{\Gamma \vdash t \Rightarrow \text{apply}\ t}$$

Decompiling Proof Terms to Tactics

- Base: apply
- Recurse over the proof term and construct a proof script using a predefined set of tactics.
- The proof term for the base case of rev_app_distr (Page 10) alongside the proof script that Pumpkin Pi suggests.

```
fun (y0 : list A)1 =>
  list_rect2 _ _ (fun a l H2 =>
    eq_ind_r3 _ eq_refl4 (app_nil_r (rev l) (a::[]))3)
    eq_refl5
    y02

- intro y0.1 induction y0 as [a l H|].2
  + simpl. rewrite app_nil_r.3 auto.4
  + auto.5
```

Catalogue

Case Studies

Some changes using Pumpkin Pi:

Class	Config.	Examples	Sav.	Repair Tools	Search Tools
Algebraic Ornaments	Auto	List to Packed Vector, hs-to-coq [3]	☺	Pumpkin Pi, Devoid, UP	Pumpkin Pi, Devoid
		List to Packed Vector, Std. Library [16]	☺	Pumpkin Pi, Devoid, UP	Pumpkin Pi, Devoid
Unpack Sigma Types	Auto	Vector of Given Length, hs-to-coq [3]	☺	Pumpkin Pi, UP	Pumpkin Pi
Tuples & Records	Auto	Simple Records [13]	☺	Pumpkin Pi, UP	Pumpkin Pi
		Parameterized Records [17]	☺	Pumpkin Pi, UP	Pumpkin Pi
		Industrial Use [18]	☺	Pumpkin Pi, UP	Pumpkin Pi
Permute Constructors	Auto	List, Standard Library [1]	☺	Pumpkin Pi, UP	Pumpkin Pi
		Modifying PL, REPLica Benchmark [1]	☺	Pumpkin Pi, UP	Pumpkin Pi
		Large Ambiguous Enum [1]	☺	Pumpkin Pi, UP	Pumpkin Pi
Add new Constructors	Mixed	PL Extension, REPLica Benchmark [19]	☹	Pumpkin Pi	Pumpkin Pi (partial)
Factor Constructors	Manual	External Example [2]	☺	Pumpkin Pi, UP	None
Permute Hypotheses	Manual	External Example [20]	☹	Pumpkin Pi, UP	None
Change Ind. Structure	Manual	Unary to Binary, Classic Benchmark [5]	☺	Pumpkin Pi, Magaud	None
		Vector to Fin. Set, External Example [21]	☺	Pumpkin Pi	None

Catalogue

Conclusions

Pumpkin Pi-a proof repair tool for changes in datatypes:

- search procedures for equivalences
- a proof term transformation
- a proof term to tactic decompiler.

Reference

1 Ringer, T. , Porter, R. D. , Yazdani, N. , Leo, J. , Dan, G. . (2021). Proof repair across type equivalences. PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. ACM.

2 https://www.pldi21.org/poster_pldi.43.html

Thank You