Compositional pre-processing for automated reasoning in dependent type theory

Valentin Blot [1 2], Denis Cousineau [4], Enzo Crance [2 3 4], Louise Dubois de Prisque [1 2], Chantal Keller [1], Assia Mahboubi [2 3], Pierre Vial [1 2] (France)

1 – LMF, Université Paris-Saclay
2 – Inria
3 – LS2N, Nantes Université
4 – Mitsubishi Electric R&D Centre Europe

CPP 2023
Boston, Massachusetts, USA
Tuesday 17 January 2023

The zoo of automation tactics

automation tactics

oracle-based
+ reconstruction

lia

[Besson, 2006]

first order solvers

hammer

[Czajka & Kaliszyk, 2018]

SMT solvers

SMTCoq

[Armand et al., 2011]

autarkic

itauto

[Besson, 2021]

decision procedures

ring

[Grégoire & Mahboubi, 2005]

proof search and heuristics

crush, **sauto**, **firstorder**

[Chlipala, 2013] [Czajka & Kaliszyk, 2018] [Corbineau, 2004]

The zoo of automation tactics

```
forall (A : Type) (l l' : list A),
  length (rev (l ++ l')) = length l + length l'
```

```
hammer.
```

The zoo of automation tactics

```
forall (A : Type) (l l' : list A) (a : A),
  length (rev (l ++ (a :: l'))) = length l + length l' + 1
```

```
Fail hammer.
```

The zoo of automation tactics

```
forall (A : Type) (l l' : list A) (a : A),
  length (rev (l ++ (a :: l'))) = length l + length l' + 1
```

```
Fail hammer.
```

```
forall (x : Z), x + 1 = 1 + x
```

```
smt.
```

(SMTCoq)

The zoo of automation tactics

```
forall (A : Type) (l l' : list A) (a : A),
  length (rev (l ++ (a :: l'))) = length l + length l' + 1
```

```
Fail hammer.
```

```
forall (x : int), x + 1 = 1 + x
```

```
Fail smt.
```

(SMTCoq)

The zoo of automation tactics

```
forall (x : int), x + 1 = 1 + x
                         lia.
```

The zoo of automation tactics

```
forall (f : int → int) (x : int), f (x + 1) = f (1 + x)
Fail lia.
```

The zoo of automation tactics

```
forall (f : int → int) (x : int), f (x + 1) = f (1 + x)
Fail lia.
```

```
forall (f : int → int) (x : int), f (x + 1) = f (1 + x)
smt.
```

(itauto)

The zoo of automation tactics

```
forall (f : int → int) (x : int), f (x + 1) = f (1 + x)
```

```
Fail lia.
```

```
forall (f : int → int) (x : int),
  (f (x + 1) == f (1 + x)) = true
```

```
Fail smt.
```

(itauto)

A compositional pre-processing toolbox

- aligning Coq goals with the logic of ATPs

- bypassing implementation details (data structures, notations, etc)

Contributions

- collection of general, small-scale, compositional pre-processing tactics
- an implemented strategy of orchestration (snipe)

$$t ::= n \mid t\ t' \mid \lambda\ t$$

$$\uparrow_c^d t$$

[Sakaguchi, 2020]

```
Inductive term : Type
```

```
Fixpoint shift (d c : nat) (t : term) : term
```

```
Lemma shift_add (d d' c c' : nat) (t : term) :
  c ⩽? c' → c' ⩽? c + d → shift d' c' (shift d c t) = shift (d' + d) c t.
Proof. elim: t d d' c c'; snipe. Qed.
```

$$\uparrow_{c'}^{d'} \uparrow_c^d t = \uparrow_c^{d'+d} t$$

```
Lemma shift_shift_distr (d d' c c' : nat) (t : term) :
  c' ⩽? c → shift d' c' (shift d c t) = shift d (d' + c) (shift d' c' t).
Proof. elim: t d d' c c'; snipe. Qed.
```

$$\uparrow_{c'}^{d'} \uparrow_c^d t = \uparrow_{d'+c}^{d} \uparrow_{c'}^{d'} t$$

Panorama of pre-processing tactics

Inductive types

Handling symbols

Going first order

Inductive types

objective: **interpret inductive types and predicates**

1. inversion on inductive predicates
2. properties of algebraic data types
3. generation statement simplification
4. pattern matching elimination

Inductive types: inductive relations

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall (n : nat), add 0 n n
  | addS :
    forall (n m k : nat), add n m k → add (S n) m (S k).
```

initial proof context

```
H: add n m k
```

final proof context

```
H': (exists (n' : nat), n = 0 /\ m = n' /\ k = n') \/
    (exists (n' m' k' : nat),
      add n' m' k' /\ n = S n' /\ m = m' /\ k = S k')
```

Inductive types: algebraic data types

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

non confusion

```
H1: forall (x : A) (l : list A), [] <> x :: l
```

injectivity

```
H2: forall (x y : A) (l l' : list A),
      x :: l = y :: l' → x = y /\ l = l'
```

generation

```
H3: forall (l : list A), exists (x : A) (l' : list A),
      l = x :: l' \/ l = []
```

Inductive types: pattern matching

```
H: forall (A : Type) (def : A) (l : list A) (n : nat),
  nth_default def l n =
    match nth l n with
    | Some x ⇒ x
    | None ⇒ def
    end
```

initial proof context

final proof context

```
H1: forall (A : Type) (def : A) (l : list A) (n : nat),
  nth l n = Some x → nth_default def l n = x

H2: forall (A : Type) (def : A) (l : list A) (n : nat),
  nth l n = None → nth_default def l n = def
```

Handling symbols

objective: **interpret symbols in the proof context**

1. definition unfolding
2. fixpoint elimination
3. theory-based pre-processing

Theory-based pre-processing: Trakt

```
forall (x : Z), x ⩾ 0 → x + x ⩾ x                      lia.

forall (n : nat), n + n ⩾ n                           zify; lia.
```

`zify` [Besson, 2017]

```
forall (x : int), x ⩾ 0 → (2 * x = x + x)%R   zify; lia.
```

`mczify` [Sakaguchi, 2021]

```
forall (x : int) (f : int → int), x ⩾ 0 →
  f (2 * x)%R == f (x + x)%R = true                       ?
```

Theory-based pre-processing: Trakt

```
forall (x : int) (f : int → int), x ≥ 0 →
  f (2 * x)%R == f (x + x)%R = true
```

Theory-based pre-processing: Trakt

```
forall (x : int) (f : int → int), x ⩾ 0 →
  @eq_op int_eqType
    (f (@GRing.mul int_Ring 2 x))
    (f (@GRing.add int_ZmodType x x))
      = true
```

Theory-based pre-processing: Trakt

```
forall (x : int) (f : int → int), x ⩾ 0 →
  @eq_op int_eqType
    (f (@GRing.mul int_Ring 2 x))
    (f (@GRing.add int_ZmodType x x))
      = true
```

```
trakt Z bool.
```

Theory-based pre-processing: Trakt

```
forall (x : int) (f : int → int), x ≥ 0 →
  @eq_op int_eqType
    (f (@GRing.mul int_Ring 2 x))
    (f (@GRing.add int_ZmodType x x))
      = true
```

```
trakt Z bool.
```

```
forall (x : Z) (f : Z → Z), x ≥? 0 = true →
  f (2 * x) =? f (x + x) = true
```

Z.mul Z.eqb Z.add

Theory-based pre-processing: Trakt

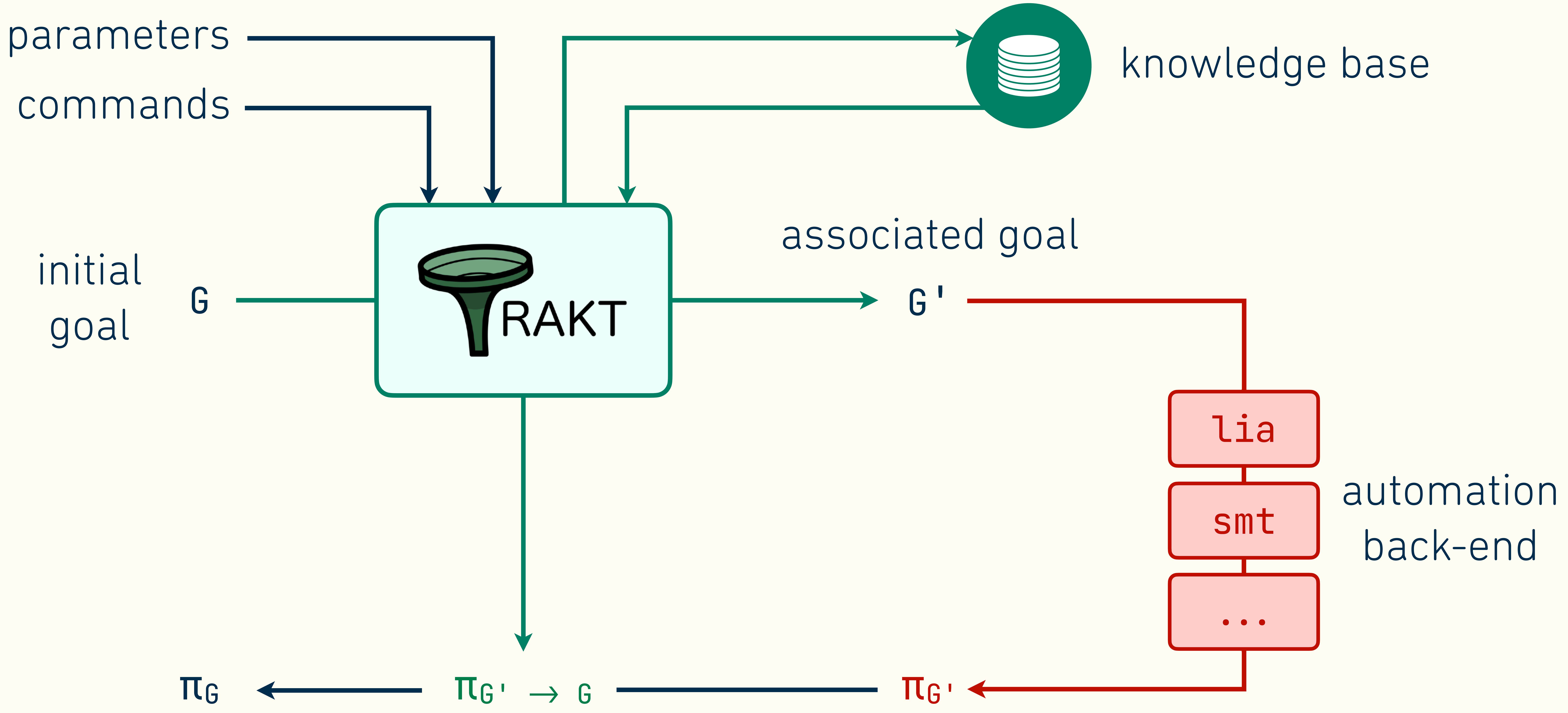
database construction before running the tactic

```
Trakt Add Embedding int Z Z_of_int Z_to_int proof.

Trakt Add Relation (@eq_op int_eqType) Z.eqb proof.

Trakt Add Symbol intZmod.addz Z.add proof.
```

Theory-based pre-processing: Trakt



Going first order

objective: **align the proof context with the scope of an external prover**

1. hypothesis monomorphisation

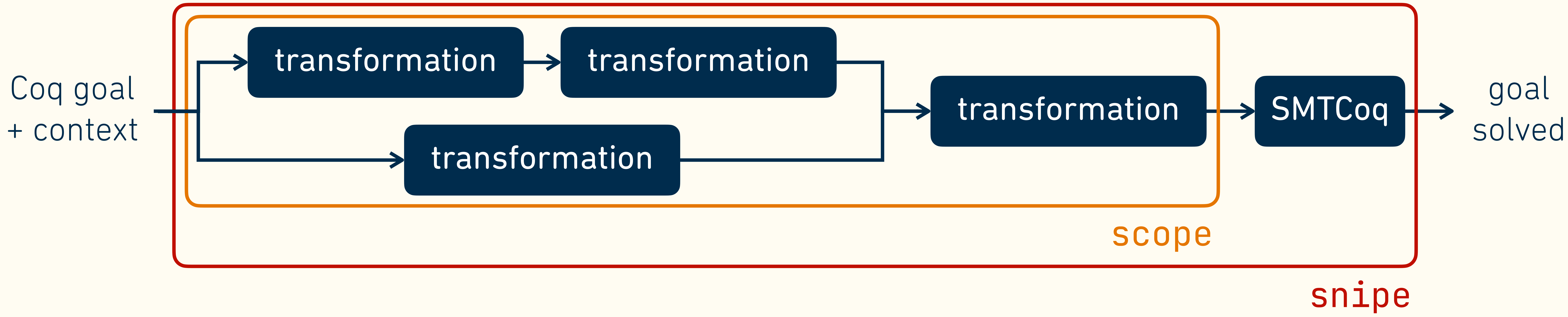
2. pointwise version of higher-order equalities

Orchestration

The snipe orchestration tactic

scope = combination of pre-processing tactics

snipe = pre-processing + SMTCoq



Example

```
Inductive term : Type :=
  | var : nat → term
  | app : term → term → term
  | abs : term → term.
```

```
Fixpoint shift (d c : nat) (t : term) : term :=
  match t with
  | var n ⇒ var (if c ≤? n then n + d else n)
  | app t1 t2 ⇒ app (shift d c t1) (shift d c t2)
  | abs t' ⇒ abs (shift d (S c) t')
  end.
```

```
Lemma shift_zero (n : nat) (t : term) : shift 0 n t = t.
Proof. elim: t n; snipe. Qed.
```

$\uparrow_n^0 t = t$

global references to interpret in the subgoals: `term`, `shift`

Example: interpreting `term`

```
Inductive term : Type :=
  | var : nat → term
  | app : term → term → term
  | abs : term → term.
```

non confusion

```
H1: forall n t1 t2, var n <> app t1 t2
H2: forall n t, var n <> abs t
H3: forall t1 t2 t, app t1 t2 <> abs t
```

injectivity

```
H4: forall n n', var n = var n' → n = n'
H5: forall t1 t1' t2 t2', app t1 t2 = app t1' t2' → t1 = t1' /\ t2 = t2'
H6: forall t t', abs t = abs t' → t = t'
```

generation

```
H7: forall t, exists n t1 t2 t', t = var n \/ t = app t1 t2 \/ t = abs t'
```

Example: interpreting shift

```
Fixpoint shift (d c : nat) (t : term) : term :=
  match t with
  | var n ⇒ var (if c ≤? n then n + d else n)
  | app t1 t2 ⇒ app (shift d c t1) (shift d c t2)
  | abs t' ⇒ abs (shift d (S c) t')
  end.
```

definition unfolding

```
H8:
  shift = fix F d c t :=
    match t with
    | var n ⇒ var (if c ≤? n then n + d else n)
    | app t1 t2 ⇒ app (F d c t1) (F d c t2)
    | abs t' ⇒ abs (F d (S c) t')
    end
```

Example: interpreting shift

```
Fixpoint shift (d c : nat) (t : term) : term :=
  match t with
  | var n ⇒ var (if c ≤? n then n + d else n)
  | app t1 t2 ⇒ app (shift d c t1) (shift d c t2)
  | abs t' ⇒ abs (shift d (S c) t')
  end.
```

definition unfolding

higher-order equality elimination

```
H8': forall d c t,
  shift d c t = (fix F d c t :=
    match t with
    | var n ⇒ var (if c ≤? n then n + d else n)
    | app t1 t2 ⇒ app (F d c t1) (F d c t2)
    | abs t' ⇒ abs (F d (S c) t')
    end) d c t
```

Example: interpreting shift

```
Fixpoint shift (d c : nat) (t : term) : term :=
  match t with
  | var n ⇒ var (if c ≤? n then n + d else n)
  | app t1 t2 ⇒ app (shift d c t1) (shift d c t2)
  | abs t' ⇒ abs (shift d (S c) t')
  end.
```

definition unfolding

higher-order equality elimination

fixpoint elimination

```
H8'': forall d c t,
  shift d c t =
    match t with
    | var n ⇒ var (if c ≤? n then n + d else n)
    | app t1 t2 ⇒ app (shift d c t1) (shift d c t2)
    | abs t' ⇒ abs (shift d (S c) t')
    end
```

Example: interpreting shift

```
Fixpoint shift (d c : nat) (t : term) : term :=
  match t with
  | var n ⇒ var (if c ⩽? n then n + d else n)
  | app t1 t2 ⇒ app (shift d c t1) (shift d c t2)
  | abs t' ⇒ abs (shift d (S c) t')
  end.
```

definition unfolding
higher-order equality elimination
fixpoint elimination
pattern matching elimination

```
H9: forall d c t n,
  t = var n → shift d c t = var (if c ⩽? n then n + d else n)

H10: forall d c t t1 t2,
  t = app t1 t2 → shift d c t = app (shift d c t1) (shift d c t2)

H11: forall d c t t',
  t = abs t' → shift d c t = abs (shift d (S c) t')
```

Example: interpreting shift

```
Fixpoint shift (d c : nat) (t : term) : term :=
  match t with
  | var n ⇒ var (if c ≤? n then n + d else n)
  | app t1 t2 ⇒ app (shift d c t1) (shift d c t2)
  | abs t' ⇒ abs (shift d (S c) t')
  end.
```

definition unfolding

higher-order equality elimination

fixpoint elimination

pattern matching elimination

```
H9a: forall d c t n,
  t = var n → c ≤? n = true → shift d c t = var (n + d)

H9b: forall d c t n,
  t = var n → c ≤? n = false → shift d c t = var n

H10: forall d c t t1 t2,
  t = app t1 t2 → shift d c t = app (shift d c t1) (shift d c t2)

H11: forall d c t t',
  t = abs t' → shift d c t = abs (shift d (S c) t')
```

Example: interpreting shift

```
Fixpoint shift (d c : nat) (t : term) : term :=
  match t with
  | var n ⇒ var (if c ≤? n then n + d else n)
  | app t1 t2 ⇒ app (shift d c t1) (shift d c t2)
  | abs t' ⇒ abs (shift d (S c) t')
  end.
```

definition unfolding

higher-order equality elimination

fixpoint elimination

pattern matching elimination

Trakt

```
H9a: forall (d c : nat) (t : term) (n : nat),
  t = var n → c ≤? n = true →
    shift d c t = var (n + d)
```

```
H9a': forall (d' : Z), d' ≥ 0 → forall (c' : Z), c' ≥ 0 → forall (t : term) (n' : Z), n' ≥ 0 →
  t = var (Z.to_nat n') → c' ≤ n' →
    shift (Z.to_nat d') (Z.to_nat c') t = var (Z.to_nat (n' + d'))
```

Conclusion

Conclusion

suite of compositional standalone transformations

add your own!

feedback is welcome!

compositionality: several meta-languages

Ltac	MetaCoq	Coq-Elpi
[Delahaye, 2000]	[Sozeau et al., 2020]	[Tassi, 2018]

orchestration in the sniper plugin

make your own!

https://github.com/smtcoq/sniper/releases/tag/cpp23
https://github.com/ecranceMERCE/trakt/releases/tag/1.2

Thank you!

Laboratoire Méthodes Formelles
université PARIS-SACLAY
Inria
Nantes Université
LS2N LABORATOIRE DES SCIENCES DU NUMÉRIQUE DE NANTES
MITSUBISHI ELECTRIC

lia
Frédéric Besson
Fast reflexive arithmetic tactics: the linear case and beyond
TYPES 2006

itauto
Frédéric Besson
Itauto: An Extensible Intuitionistic SAT Solver
ITP 2021

hammer
sauto
Łukasz Czajka & Cezary Kaliszyk
Hammer for Coq: Automation for dependent type theory
Journal of automated reasoning, 2018

ring
Benjamin Grégoire & Assia Mahboubi
Proving equalities in a commutative ring done right in Coq
TPHOL 2005

firstorder
Pierre Corbineau
First-order reasoning in the calculus of inductive constructions
Lecture notes in computer science, 2004

crush
Adam Chlipala
Certified programming with dependent types:
a pragmatic introduction to the Coq proof assistant
MIT Press, 2013

λ-calculus formalisation
Kazuhiko Sakaguchi
`https://github.com/pi8027/lambda-calculus`
2020

zify
Frédéric Besson
ppsimpl: a reflexive Coq tactic for canonising goals
CoqPL, 2017

mczify
Kazuhiko Sakaguchi
`https://github.com/math-comp/mczify`
2021

Ltac
David Delahaye
A tactic language for the system Coq
LPAR 2000

MetaCoq
Matthieu Sozeau, et al.
The MetaCoq project
Journal of automated reasoning, 2020

Coq-Elpi
Enrico Tassi
Elpi: an extension language for Coq
(Metaprogramming Coq in the Elpi λProlog dialect)
CoqPL, 2018

SMTCoq
Michaël Armand, Germain Faure, Benjamin Grégoire, et al.
A Modular Integration of SAT/SMT Solvers to Coq through
Proof Witnesses
CPP 2011

Burak Ekici, Alain Mebsout, Cesare Tinelli, et al.
SMTCoq: A plug-in for integrating SMT solvers into Coq
CAV 2017