Practical proof search for Coq by type inhabitation

Łukasz Czajka, TU Dortmund University

3 July 2020

Or: proof search with “Super Auto”

1. A brief high-level description of the proof search procedure.
2. Demonstration.

Calculus of Inductive Constructions

- A dependently typed lambda calculus with inductive types.

Calculus of Inductive Constructions

- A dependently typed lambda calculus with inductive types.
- Curry-Howard isomorphism: propositions are types, proofs are lambda-terms.

Calculus of Inductive Constructions

- A dependently typed lambda calculus with inductive types.
- Curry-Howard isomorphism: propositions are types, proofs are lambda-terms.
- Inductive types used for inductive predicate definitions.

```
Inductive Forall (A : Type) (P : A -> Prop)
  : List A -> Prop :=
| fnil : Forall P nil
| fcons : forall (x : A) (l : List A),
            P x -> Forall P l -> Forall P (cons x l)
```

Logical connectives as inductive types

```
Inductive ⊥ : Prop := .
Inductive ∧ (A : Prop) (B : Prop) : Prop :=
  conj : A -> B -> A ∧ B.
Inductive ∨ (A : Prop) (B : Prop) : Prop :=
  inl : A -> A ∨ B | inr : B -> A ∨ B.
Inductive ∃ (A : Type) (P : A -> Prop) : Prop :=
  exi : forall x : A, P x -> ∃ A P.
```

Proof search for Coq

- Straightforward naive idea: systematically search for inhabitants of a type (in some sort of normal form).

Proof search for Coq

- Straightforward naive idea: systematically search for inhabitants of a type (in some sort of normal form).
- But how to make this practically feasible?

Proof search for Coq: basic building blocks

- **Introduction.**
  - Goal: `|- forall H : A, B`.
  - Term: lambda abstraction: `(fun H => _)`.

Proof search for Coq: basic building blocks

- **Introduction.**
  - Goal: `|- forall H : A, B`.
  - Term: lambda abstraction: `(fun H => _)`.
- **Application.**
  - Goal: `H : forall x1 ... xn, A' |- A`.
  - Term: application: `(H _ .. _)`.

Proof search for Coq: basic building blocks

- **Introduction.**
  - Goal: `|- forall H : A, B`.
  - Term: lambda abstraction: `(fun H => _)`.
- **Application.**
  - Goal: `H : forall x1 ... xn, A' |- A`.
  - Term: application: `(H _ .. _)`.
- **Construction.**
  - Goal: for I inductive: `|- I _ .. _`.
  - Term: application with constructor head: `(c _ .. _)`.

Proof search for Coq: basic building blocks

- **Introduction.**
  - Goal: `|- forall H : A, B`.
  - Term: lambda abstraction: `(fun H => _)`.
- **Application.**
  - Goal: `H : forall x1 ... xn, A' |- A`.
  - Term: application: `(H _ .. _)`.
- **Construction.**
  - Goal: for I inductive: `|- I _ .. _`.
  - Term: application with constructor head: `(c _ .. _)`.
- **Elimination.**
  - Goal: for I inductive: `H : forall x1 .. xn, I _ .. _ |- _`.
  - Term:

```
match (H _ .. _) as x in I _ .. _ i1 .. ik
                 return P i1 .. ik x
with .. end
```

Proof search for Coq: basic building blocks

- **Introduction.**
  - Goal: `|- forall H : A, B`.
  - Term: lambda abstraction: `(fun H => _)`.
- **Application.**
  - Goal: `H : forall x1 ... xn, A' |- A`.
  - Term: application: `(H _ .. _)`.
- **Construction.**
  - Goal: for I inductive: `|- I _ .. _`.
  - Term: application with constructor head: `(c _ .. _)`.
- **Elimination.**
  - Goal: for I inductive: `H : forall x1 .. xn, I _ .. _ |- _`.
  - Term:

```
match (H _ .. _) as x in I _ .. _ i1 .. ik
                 return P i1 .. ik x
with .. end
```

This corresponds to searching for normal forms wrt. β, ι and permutation conversions for matches.

Proof search for Coq: restrictions

- **Eager introduction**: perform introduction eagerly without backtracking.

Proof search for Coq: restrictions

- **Eager introduction**: perform introduction eagerly without backtracking.
    - Corresponds to searching for η-long normal forms.

Proof search for Coq: restrictions

- **Eager introduction**: perform introduction eagerly without backtracking.
  - Corresponds to searching for η-long normal forms.
- **Elimination restriction**: perform elimination only immediately after introduction or another elimination.

Proof search for Coq: restrictions

- **Eager introduction**: perform introduction eagerly without backtracking.
  - Corresponds to searching for η-long normal forms.
- **Elimination restriction**: perform elimination only immediately after introduction or another elimination.
  - Preserves completeness in a "first-order" fragment with inductive types.

Proof search for Coq: restrictions

- **Eager introduction**: perform introduction eagerly without backtracking.
  - Corresponds to searching for η-long normal forms.
- **Elimination restriction**: perform elimination only immediately after introduction or another elimination.
  - Preserves completeness in a "first-order" fragment with inductive types.
- **Eager simple elimination**: if a hypothesis H has a non-recursive inductive type then eliminate H eagerly without backtracking and remove* it from the context.

Proof search for Coq: restrictions

- **Eager introduction**: perform introduction eagerly without backtracking.
  - Corresponds to searching for η-long normal forms.
- **Elimination restriction**: perform elimination only immediately after introduction or another elimination.
  - Preserves completeness in a "first-order" fragment with inductive types.
- **Eager simple elimination**: if a hypothesis H has a non-recursive inductive type then eliminate H eagerly without backtracking and remove* it from the context.
  - * "remove" should be understood appropriately.

Proof search for Coq: restrictions

- **Eager introduction**: perform introduction eagerly without backtracking.
  - Corresponds to searching for η-long normal forms.
- **Elimination restriction**: perform elimination only immediately after introduction or another elimination.
  - Preserves completeness in a "first-order" fragment with inductive types.
- **Eager simple elimination**: if a hypothesis H has a non-recursive inductive type then eliminate H eagerly without backtracking and remove* it from the context.
  - * "remove" should be understood appropriately.
  - Preserves completeness in a "first-order" fragment with inductive types.

Proof search for Coq: restrictions

- **Eager introduction**: perform introduction eagerly without backtracking.
  - Corresponds to searching for η-long normal forms.
- **Elimination restriction**: perform elimination only immediately after introduction or another elimination.
  - Preserves completeness in a "first-order" fragment with inductive types.
- **Eager simple elimination**: if a hypothesis H has a non-recursive inductive type then eliminate H eagerly without backtracking and remove* it from the context.
  - * "remove" should be understood appropriately.
  - Preserves completeness in a "first-order" fragment with inductive types.
  - E.g. for conjunction this means: replace $\Gamma, x : \alpha \wedge \beta$ with $\Gamma, x_1 : \alpha, x_2 : \beta$.

Proof search for Coq: restrictions

- **Eager introduction**: perform introduction eagerly without backtracking.
  - Corresponds to searching for η-long normal forms.
- **Elimination restriction**: perform elimination only immediately after introduction or another elimination.
  - Preserves completeness in a "first-order" fragment with inductive types.
- **Eager simple elimination**: if a hypothesis H has a non-recursive inductive type then eliminate H eagerly without backtracking and remove* it from the context.
  - * "remove" should be understood appropriately.
  - Preserves completeness in a "first-order" fragment with inductive types.
  - E.g. for conjunction this means: replace $\Gamma, x : \alpha \wedge \beta$ with $\Gamma, x_1 : \alpha, x_2 : \beta$.
- **Loop checking**: if we encounter the same conjecture again without changing the context in the meantime, then we can fail.

Proof search for Coq: restrictions

- **Eager introduction**: perform introduction eagerly without backtracking.
  - Corresponds to searching for η-long normal forms.
- **Elimination restriction**: perform elimination only immediately after introduction or another elimination.
  - Preserves completeness in a “first-order” fragment with inductive types.
- **Eager simple elimination**: if a hypothesis H has a non-recursive inductive type then eliminate H eagerly without backtracking and remove* it from the context.
  - * “remove” should be understood appropriately.
  - Preserves completeness in a “first-order” fragment with inductive types.
  - E.g. for conjunction this means: replace $\Gamma, x : \alpha \wedge \beta$ with $\Gamma, x_1 : \alpha, x_2 : \beta$.
- **Loop checking**: if we encounter the same conjecture again without changing the context in the meantime, then we can fail.
  - Preserves completeness with definitional proof irrelevance (if we do loop checking only for proofs).

Issues with proof search in dependent type theory

- When, how and how much to perform reduction? (heuristics)

Issues with proof search in dependent type theory

- When, how and how much to perform reduction? (heuristics)
- When to perform elimination? (elimination restrictions; heuristics)

Issues with proof search in dependent type theory

- When, how and how much to perform reduction? (heuristics)
- When to perform elimination? (elimination restrictions; heuristics)
- How to perform elimination? (heuristics)

Proof search for Coq: heuristics

- Hypothesis simplification.
- Limited forward reasoning.
- Rewriting (ordered with LPO or heuristic).
- Leaf solver with congruence closure and linear arithmetic tactics.
- Unfolding of constants.
- Simplifications for sigma-types.
- Elimination of discriminees in case expressions.
- ...

Heuristics are optional: the procedure uses reasonable defaults which may be customised.

Permutation conversions

Not always valid!

$$\texttt{case}(u; \lambda\vec{a} : \vec{\alpha}.\lambda x : I\vec{q}\vec{a}.\forall z : \sigma.\tau; \vec{x_1} : \vec{\tau_1} \Rightarrow t_1 \mid \ldots \mid \vec{x_k} : \vec{\tau_k} \Rightarrow t_k)w \to_{\rho_1}$$
$$\texttt{case}(u; \lambda\vec{a} : \vec{\alpha}.\lambda x : I\vec{q}\vec{a}.\tau[w/z]; \vec{x_1} : \vec{\tau_1} \Rightarrow t_1 w \mid \ldots \mid \vec{x_k} : \vec{\tau_k} \Rightarrow t_k w)$$

$$\texttt{case}(\texttt{case}(u; Q; \vec{x_1} : \vec{\tau_1} \Rightarrow t_1 \mid \ldots \mid \vec{x_k} : \vec{\tau_k} \Rightarrow t_k); R; P) \to_{\rho_2}$$
$$\texttt{case}(u; R'; \vec{x_1} : \vec{\tau_1} \Rightarrow \texttt{case}(t_1; R''; P) \mid \ldots \mid \vec{x_k} : \vec{\tau_k} \Rightarrow \texttt{case}(t_k; R''; P))$$

Permutation conversions

Not always valid!

$$\begin{array}{l}\texttt{case}(u; \lambda\vec{a} : \vec{\alpha}.\lambda x : I\vec{q}\vec{a}.\forall z : \sigma.\tau; \vec{x_1} : \vec{\tau_1} \Rightarrow t_1 \mid \ldots \mid \vec{x_k} : \vec{\tau_k} \Rightarrow t_k)w \to_{\rho_1} \\ \qquad \texttt{case}(u; \lambda\vec{a} : \vec{\alpha}.\lambda x : I\vec{q}\vec{a}.\tau[w/z]; \vec{x_1} : \vec{\tau_1} \Rightarrow t_1 w \mid \ldots \mid \vec{x_k} : \vec{\tau_k} \Rightarrow t_k w)\end{array}$$

$$\begin{array}{l}\texttt{case}(\texttt{case}(u; Q; \vec{x_1} : \vec{\tau_1} \Rightarrow t_1 \mid \ldots \mid \vec{x_k} : \vec{\tau_k} \Rightarrow t_k); R; P) \to_{\rho_2} \\ \qquad \texttt{case}(u; R'; \vec{x_1} : \vec{\tau_1} \Rightarrow \texttt{case}(t_1; R''; P) \mid \ldots \mid \vec{x_k} : \vec{\tau_k} \Rightarrow \texttt{case}(t_k; R''; P))\end{array}$$

Some equality information is “forgotten” when type-checking the branches of case expressions, which may make the right-hand sides of the above rules ill-typed.

Long normal forms

The "basic" version of the procedure (with "restrictions" but without "heuristics") systematically searches for "long normal forms" defined below.

Definition

A term t is a long normal form in Γ (Γ-lnf) if:

- $t = \lambda x : \alpha . t'$, and $\Gamma \vdash t : \forall x : \alpha . \beta$, and t' is a Γ-$(x : \alpha)$-lnf (defined below);
- $t = x\vec{u}$, and $\Gamma \vdash t : \tau$ with τ not a product, and each u_i is a Γ-lnf and not a case expression;
- $t = c\vec{q}\vec{v}$, and $\Gamma \vdash t : I\vec{q}\vec{w}$ with $\vec{q}$ the parameters, and c is a constructor of I, and each v_i is a Γ-lnf and not a case expression;
- $t = \mathsf{case}(x\vec{u}; \lambda\vec{a} : \vec{\alpha}.\lambda x : I\vec{q}\vec{a}.\sigma; \vec{x_1} : \vec{\tau_1} \Rightarrow t_1 \mid \ldots \mid \vec{x_k} : \vec{\tau_k} \Rightarrow t_k)$, and $\Gamma \vdash t : \tau$ with τ not a product, and each u_i is a Γ-lnf and not a case expression, and each t_i is a Γ-$(\vec{x_i} : \vec{\tau_i})$-lnf.

A term t is a Γ-Δ-lnf if:

- $\Delta = \langle\rangle$ and t is a Γ-lnf;
- $\Delta = x : \alpha, \Delta'$, and α is not $I\vec{q}\vec{u}$ for I non-recursive with $I \succ \vec{q}$, and t is a $\Gamma, x : \alpha$-Δ'-lnf;
- $\Delta = x : I\vec{q}\vec{u}, \Delta'$, and I is non-recursive with $I \succ \vec{q}$, and $\vec{q}$ are the parameter values, and $t = \mathsf{case}(x; \lambda\vec{a} : \vec{\alpha}.\lambda x : I\vec{q}\vec{a}.\tau; \vec{x_1} : \vec{\tau_1} \Rightarrow t_1 \mid \ldots \mid \vec{x_k} : \vec{\tau_k} \Rightarrow t_k)$, and $\Gamma, \Delta \vdash t : \tau[\vec{u}/\vec{a}]$, and each t_i is a Γ-Δ', $\vec{x_i} : \vec{\tau_i}$-lnf (then $x \notin \mathrm{FV}(t_i)$).

Completeness for a “first-order” fragment

Theorem

The “basic” version of the procedure (with “restrictions” but without “heuristics”) is complete for a “first-order” fragment of Coq.

Completeness for a “first-order” fragment

Theorem

The “basic” version of the procedure (with “restrictions” but without “heuristics”) is complete for a “first-order” fragment of Coq.

Proof.

Show that if there is a proof then there is a proof in long normal form, and that loop checking may be safely performed for proofs.

Completeness for a “first-order” fragment

Theorem

The “basic” version of the procedure (with “restrictions” but without “heuristics”) is complete for a “first-order” fragment of Coq.

Proof.

Show that if there is a proof then there is a proof in long normal form, and that loop checking may be safely performed for proofs.

1. Show weak normalization of $\beta\iota\rho$-reduction.

Completeness for a “first-order” fragment

Theorem

The “basic” version of the procedure (with “restrictions” but without “heuristics”) is complete for a “first-order” fragment of Coq.

Proof.

Show that if there is a proof then there is a proof in long normal form, and that loop checking may be safely performed for proofs.

1. Show weak normalization of $\beta\iota\rho$-reduction.
2. Show how to transform $\beta\iota\rho$-normal forms into long normal forms.

Completeness for a “first-order” fragment

Theorem

The “basic” version of the procedure (with “restrictions” but without “heuristics”) is complete for a “first-order” fragment of Coq.

Proof.

Show that if there is a proof then there is a proof in long normal form, and that loop checking may be safely performed for proofs.

1. Show weak normalization of $\beta\iota\rho$-reduction.
2. Show how to transform $\beta\iota\rho$-normal forms into long normal forms.
3. Loop checking is safe because proofs don’t occur in types. $\square$

Incompleteness

There are essentially two problems as far as existence of “good” normal forms is concerned:

Incompleteness

There are essentially two problems as far as existence of “good” normal forms is concerned:

- permutation conversions are not valid for dependent elimination;

Incompleteness

There are essentially two problems as far as existence of “good” normal forms is concerned:

- permutation conversions are not valid for dependent elimination;
- proofs occur in types, so changing them may break typability if the corresponding proof transformation is not included in the conversion rule.

Some numbers

Coq libraries collection
standalone (4494 problems, 30s)

tactic	proved	proved %
sauto+i	1840	40.9%
coq+i	1229	27.3%
crush+i	1134	25.2%

CompCert
standalone (5495 problems, 30s)

tactic	proved	proved %
sauto+i	941	17.1%
coq+i	372	6.8%
crush+i	355	6.5%

Coq libraries collection
standalone (4494 problems, 5s)

tactic	proved	proved %
coq	978	21.8%
sauto	888	19.6%
crush	663	14.8%
coq-no-fo	607	13.5%

CompCert
standalone (5495 problems, 5s)

tactic	proved	proved %
sauto	420	7.6%
coq	286	5.2%
coq-no-fo	237	4.3%
crush	210	3.8%

Some numbers

Coq libraries collection
standalone (4494 problems, 30s)

tactic	proved	proved %
sauto+i	1840	40.9%
coq+i	1229	27.3%
crush+i	1134	25.2%

CompCert
standalone (5495 problems, 30s)

tactic	proved	proved %
sauto+i	941	17.1%
coq+i	372	6.8%
crush+i	355	6.5%

Coq libraries collection
standalone (4494 problems, 5s)

tactic	proved	proved %
coq	978	21.8%
sauto	888	19.6%
crush	663	14.8%
coq-no-fo	607	13.5%

CompCert
standalone (5495 problems, 5s)

tactic	proved	proved %
sauto	420	7.6%
coq	286	5.2%
coq-no-fo	237	4.3%
crush	210	3.8%

But what do these numbers really mean?

How well does this work in practice?

Demonstration.

How well does this work in practice?

Demonstration.

A "polished" customisable version of the `sauto` tactic will be available soon in the upcoming v1.3 release of the CoqHammer system (`https://github.com/lukaszcz/coqhammer`).