Relieving User Effort for the `Auto` Tactic in Coq with Machine Learning

Lasse Blaauwbroek

Czech Technical University in Prague
Radboud University in Nijmegen

September 14, 2020

An exposition of different proof styles in Coq

An exposition of different proof styles in Coq

any ridicule regarding proof styles
should be directed at me and me alone

```
- eapply typing_abs. intros. cbn. eapply typing_app. apply typing_union_eli
  + apply typing_inter_intro.
    * eapply typing_abs. intros. cbn. apply typing_merge1.
      apply typing_union_intro1. eapply typing_var. apply M.add_1.
      auto. reflexivity. constructor. econstructor. intros. cbn. constructor.
    * eapply typing_abs. intros. cbn. apply typing_merge2. eapply typing_ap
      -- apply typing_inter_intro.
         ++ eapply typing_abs. intros. cbn. apply typing_merge1.
         ++ eapply typing_abs. intros. cbn. apply typing_merge2.
            ** apply typing_merge1. eapply typing_var. apply M.add_2.
            ** apply typing_merge2. eapply typing_var. apply M.add_1.
            ** constructor.
      -- eapply typing_inter_elim2. eapply typing_var. apply M.add_2.
      -- constructor.
  + eapply typing_inter_elim1. eapply typing_var. apply M.add_1. auto.
```

```
- eapply typing_abs. intros. cbn. eapply typing_app. apply typing_union_eli
  + apply typing_inter_intro.
    * eapply typing_abs. intros. cbn. apply typing_merge1.
      apply typing_union_intro1. eapply typing_var. apply M.add_1.
      auto. reflexivity. constructor. econstructor. intros. cbn. constructor.
    * eapply typing_abs. intros. cbn. apply typing_merge2. eapply typing_ap
      -- apply typing_inter_intro.
         ++ eapply typing_abs. intros. cbn. apply typing_merge1.
         ++ eapply typing_abs. intros. cbn. apply typing_merge2.
            ** apply typing_merge1. eapply typing_var. apply M.add_2.
            ** apply typing_merge2. eapply typing_var. apply M.add_1.
            ** constructor.
      -- eapply typing_inter_elim2. eapply typing_var. apply M.add_2.
      -- constructor.
  + eapply typing_inter_elim1. eapply typing_var. apply M.add_1. auto.
```

```
- eapply typing_abs. intros. cbn. eapply typing_app. apply typing_union_eli
  + apply typing_inter_intro.
    * eapply typing_abs. intros. cbn. apply typing_merge1.
      apply typing_union_intro1. eapply typing_var. apply M.add_1.
      auto. reflexivity. constructor. econstructor. intros. cbn. constructor.
    * eapply typing_abs. intros. cbn. apply typing_merge2. eapply typing_ap
      -- apply typing_inter_intro.
         ++ eapply typing_abs. intros. cbn. apply typing_merge1.
         ++ eapply typing_abs. intros. cbn. apply typing_merge2.
            ** apply typing_merge1. eapply typing_var. apply M.add_2.
            ** apply typing_merge2. eapply typing_var. apply M.add_1.
            ** constructor.
      -- eapply typing_inter_elim2. eapply typing_var. apply M.add_2.
      -- constructor.
  + eapply typing_inter_elim1. eapply typing_var. apply M.add_1. auto.
```

```
- eapply typing_abs. intros. cbn. eapply typing_app. apply typing_union_eli
  + apply typing_inter_intro.
    * eapply typing_abs. intros. cbn. apply typing_merge1.
      apply typing_union_intro1. eapply typing_var. apply M.add_1.
      auto. reflexivity. constructor. econstructor. intros. cbn. constructor.
    * eapply typing_abs. intros. cbn. apply typing_merge2. eapply typing_ap
      -- apply typing_inter_intro.
         ++ eapply typing_abs. intros. cbn. apply typing_merge1.
         ++ eapply typing_abs. intros. cbn. apply typing_merge2.
            ** apply typing_merge1. eapply typing_var. apply M.add_2.
            ** apply typing_merge2. eapply typing_var. apply M.add_1.
            ** constructor.
      -- eapply typing_inter_elim2. eapply typing_var. apply M.add_2.
      -- constructor.
  + eapply typing_inter_elim1. eapply typing_var. apply M.add_1. auto.
```

```
- eapply typing_abs. intros. cbn. eapply typing_app. apply typing_union_eli
  + apply typing_inter_intro.
    * eapply typing_abs. intros. cbn. apply typing_merge1.
      apply typing_union_intro1. eapply typing_var. apply M.add_1.
      auto. reflexivity. constructor. econstructor. intros. cbn. constructor.
    * eapply typing_abs. intros. cbn. apply typing_merge2. eapply typing_ap
      -- apply typing_inter_intro.
         ++ eapply typing_abs. intros. cbn. apply typing_merge1.
         ++ eapply typing_abs. intros. cbn. apply typing_merge2.
            ** apply typing_merge1. eapply typing_var. apply M.add_2.
            ** apply typing_merge2. eapply typing_var. apply M.add_1.
            ** constructor.
      -- eapply typing_inter_elim2. eapply typing_var. apply M.add_2.
      -- constructor.
  + eapply typing_inter_elim1. eapply typing_var. apply M.add_1. auto.
```

easy to read

easy to step through

```
- eapply typing_abs. intros. cbn. eapply typing_app. apply typing_union_eli
  + apply typing_inter_intro.
    * eapply typing_abs. intros. cbn. apply typing_merge1.
      apply typing_union_intro1. eapply typing_var. apply M.add_1.
      auto. reflexivity. constructor. econstructor. intros. cbn. constructor.
    * eapply typing_abs. intros. cbn. apply typing_merge2. eapply typing_ap
      -- apply typing_inter_intro.
         ++ eapply typing_abs. intros. cbn. apply typing_merge1.
         ++ eapply typing_abs. intros. cbn. apply typing_merge2.
            ** apply typing_merge1. eapply typing_var. apply M.add_2.
            ** apply typing_merge2. eapply typing_var. apply M.add_1.
            ** constructor.
      -- eapply typing_inter_elim2. eapply typing_var. apply M.add_2.
      -- constructor.
  + eapply typing_inter_elim1. eapply typing_var. apply M.add_1. auto.
```

tedious to write

tedious to maintain

```
- eapply typing_abs. intros. cbn. eapply typing_app. apply typing_union_eli
  + apply typing_inter_intro.
    * eapply typing_abs. intros. cbn. apply typing_merge1.
      apply typing_union_intro1. eapply typing_var. apply M.add_1.
      auto. reflexivity. constructor. econstructor. intros. cbn. constructor.
    * eapply typing_abs. intros. cbn. apply typing_merge2. eapply typing_ap
      -- apply typing_inter_intro.
         ++ eapply typing_abs. intros. cbn. apply typing_merge1.
         ++ eapply typing_abs. intros. cbn. apply typing_merge2.
            ** apply typing_merge1. eapply typing_var. apply M.add_2.
            ** apply typing_merge2. eapply typing_var. apply M.add_1.
            ** constructor.
      -- eapply typing_inter_elim2. eapply typing_var. apply M.add_2.
      -- constructor.
  + eapply typing_inter_elim1. eapply typing_var. apply M.add_1. auto.
```

```
split ; induction T ; cbn in * ; auto ; unfold compare, ge, flip in *.
firstorder ; (rewrite tree_forall_occurs in H, H1 ;
             [| apply le_q_compatible | apply ge_eq_compatible]) ;
[rewrite H0, H4 | rewrite H | rewrite H1] ; try destruct leb ; auto.
firstorder ; destruct (compare_spec t0 n) ; auto ; destruct H6 ; rewrite H6 in
[case_eq (leb n t0) | case_eq (leb t0 n)] ; intro ; try contradiction ;
rewrite H8 in H0 ; firstorder.
```

```
split ; induction T ; cbn in * ; auto ; unfold compare, ge, flip in *.
firstorder ; (rewrite tree_forall_occurs in H, H1 ;
             [| apply le_q_compatible | apply ge_eq_compatible]) ;
[rewrite H0, H4 | rewrite H | rewrite H1] ; try destruct leb ; auto.
firstorder ; destruct (compare_spec t0 n) ; auto ; destruct H6 ; rewrite H6 in
[case_eq (leb n t0) | case_eq (leb t0 n)] ; intro ; try contradiction ;
rewrite H8 in H0 ; firstorder.
```

short to write

less maintenance effort

```
split ; induction T ; cbn in * ; auto ; unfold compare, ge, flip in *.
firstorder ; (rewrite tree_forall_occurs in H, H1 ;
             [| apply le_q_compatible | apply ge_eq_compatible]) ;
[rewrite H0, H4 | rewrite H | rewrite H1] ; try destruct leb ; auto.
firstorder ; destruct (compare_spec t0 n) ; auto ; destruct H6 ; rewrite H6 in
[case_eq (leb n t0) | case_eq (leb t0 n)] ; intro ; try contradiction ;
rewrite H8 in H0 ; firstorder.
```

short to write

less maintenance effort

difficult to write

```
split; induction T; cbn in *; auto; unfold compare, ge, flip in *.
firstorder; (rewrite tree_forall_occurs in H, H1;
             [| apply le_q_compatible | apply ge_eq_compatible]);
[rewrite H0, H4 | rewrite H | rewrite H1]; try destruct leb; auto.
firstorder; destruct (compare_spec t0 n); auto; destruct H6; rewrite H6 in
[case_eq (leb n t0) | case_eq (leb t0 n)]; intro; try contradiction;
rewrite H8 in H0; firstorder.
```

short to write

less maintenance effort

difficult to write

```
split ; induction T ; cbn in * ; auto ; unfold compare, ge, flip in *.
firstorder ; (rewrite tree_forall_occurs in H, H1 ;
             [| apply le_q_compatible | apply ge_eq_compatible]) ;
[rewrite H0, H4 | rewrite H | rewrite H1] ; try destruct leb ; auto.
firstorder ; destruct (compare_spec t0 n) ; auto ; destruct H6 ; rewrite H6 in
[case_eq (leb n t0) | case_eq (leb t0 n)] ; intro ; try contradiction ;
rewrite H8 in H0 ; firstorder.
```

short to write

less maintenance effort

difficult to write

difficult to read

```
have [-> | nzU] := eqVneq U 0%VS.
  by right=> [[e []]]; rewrite memv0 => ->.
pose X := vbasis U; pose feq f1 f2 := [tuple of map f1 X ++ map f2 X].
have feqL f i: tnth (feq _ f _) (lshift _ i) = f X`_i.
  set v := f _; rewrite (tnth_nth v) /= nth_cat size_map size_tuple.
  by rewrite ltn_ord (nth_map 0) ?size_tuple.
apply: (iffP (vsolve_eqP _ _ _)) => [[e Ue id_e] | [e [Ue _ id_e]]].
  suffices idUe: in U, forall u, e * u = u /\ u * e = u.
    exists e; split=> //; apply: contraNneq nzU => e0; rewrite -subv0.
    by apply/subvP=> u /idUe[<- _]; rewrite e0 mul0r mem0v.
  move=> u /coord_vbasis->; rewrite mulr_sumr mulr_suml.
  have: (\dim (A * M) - \dim (sumA X) < k.+1)%N by [].
  have: [/\ (sumA X <= A * M)%VS, directv (sumA X) & 0 \notin X].
  split; apply/eq_bigr=> i _; rewrite -(scalerAr, scalerAl); congr (_ *: _).
    by have:= id_e (lshift _ i); rewrite  lfunE.
  by have:= id_e (rshift _ i); rewrite  lfunE.
```

```
have [-> | nzU] := eqVneq U 0%VS.
  by right=> [[e []]]; rewrite memv0 => ->.
pose X := vbasis U; pose feq f1 f2 := [tuple of map f1 X ++ map f2 X].
have feqL f i: tnth (feq _ f _) (lshift _ i) = f X`_i.
  set v := f _; rewrite (tnth_nth v) /= nth_cat size_map size_tuple.
  by rewrite ltn_ord (nth_map 0) ?size_tuple.
apply: (iffP (vsolve_eqP _ _ _)) => [[e Ue id_e] | [e [Ue _ id_e]]].
  suffices idUe: in U, forall u, e * u = u /\ u * e = u.
    exists e; split=> //; apply: contraNneq nzU => e0; rewrite -subv0.
    by apply/subvP=> u /idUe[<- _]; rewrite e0 mul0r mem0v.
  move=> u /coord_vbasis->; rewrite mulr_sumr mulr_suml.
  have: (\dim (A * M) - \dim (sumA X) < k.+1)%N by [].
  have: [/\ (sumA X <= A * M)%VS, directv (sumA X) & 0 \notin X].
  split; apply/eq_bigr=> i _; rewrite -(scalerAr, scalerAl); congr (_ *: _).
    by have:= id_e (lshift _ i); rewrite  lfunE.
  by have:= id_e (rshift _ i); rewrite  lfunE.
```

compact and independently readable in principle

```
have [-> | nzU] := eqVneq U 0%VS.
  by right=> [[e []]]; rewrite memv0 => ->.
pose X := vbasis U; pose feq f1 f2 := [tuple of map f1 X ++ map f2 X].
have feqL f i: tnth (feq _ f _) (lshift _ i) = f X`_i.
  set v := f _; rewrite (tnth_nth v) /= nth_cat size_map size_tuple.
  by rewrite ltn_ord (nth_map 0) ?size_tuple.
apply: (iffP (vsolve_eqP _ _ _)) => [[e Ue id_e] | [e [Ue _ id_e]]].
  suffices idUe: in U, forall u, e * u = u /\ u * e = u.
    exists e; split=> //; apply: contraNneq nzU => e0; rewrite -subv0.
    by apply/subvP=> u /idUe[<- _]; rewrite e0 mul0r mem0v.
  move=> u /coord_vbasis->; rewrite mulr_sumr mulr_suml.
  have: (\dim (A * M) - \dim (sumA X) < k.+1)%N by [].
  have: [/\ (sumA X <= A * M)%VS, directv (sumA X) & 0 \notin X].
  split; apply/eq_bigr=> i _; rewrite -(scalerAr, scalerAl); congr (_ *: _).
    by have:= id_e (lshift _ i); rewrite  lfunE.
  by have:= id_e (rshift _ i); rewrite  lfunE.
```

difficult to read in practice, unsuitable for large proof states tauto. omega. ring.

`tauto.` `omega.` `ring.`

generally pretty great

`tauto.` `omega.` `ring.`

generally pretty great

only operate in a specific domain

firstorder. hammer. search.

`firstorder.` `hammer.` `search.`

you don't have to do anything

`firstorder.` `hammer.` `search.`

you don’t have to do anything

you will be doing nothing for a long time

```
induction x; auto.
```

```
induction x; auto.
```

One or two crucial proof steps,
followed by human-guided automation.

`induction x; auto.`

One or two crucial proof steps,
followed by human-guided automation.

- ▷ Abandons the panacea of a “readable” proof
- ▷ Instead, encodes all knowledge into lemmas

`induction x; auto.`

One or two crucial proof steps,
followed by human-guided automation.

- Abandons the panacea of a “readable” proof
- Instead, encodes all knowledge into lemmas
- Plus heuristics on how and when to use lemmas

`induction x; auto.`

One or two crucial proof steps,
followed by human-guided automation.

- Abandons the panacea of a "readable" proof
- Instead, encodes all knowledge into lemmas
- Plus heuristics on how and when to use lemmas
  (which may also be unreadable)

What is `auto`?

(and how does it differ from any other automation styles)

What is `auto`?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

What is `auto`?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

`Lemma le_S :` $\forall xy,$ `x < y` $\rightarrow$ `x + 1 < x + 1.`

Proof. ... Qed.

What is `auto`?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

`Lemma le_S :` $\forall xy$, `x < y` $\rightarrow$ `x + 1 < x + 1.`

Proof. ... Qed.

Hint Rewrite le_S.

What is auto?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

`Lemma le_S :` $\forall xy,$ `x < y` $\rightarrow$ `x + 1 < x + 1.`

Proof. ... Qed.

Hint Rewrite le_S.

Proof State: $\frac{w}{z} + 1 < k * a + 1$

What is `auto`?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

`Lemma le_S :` $\forall xy,$ `x < y` $\rightarrow$ `x + 1 < x + 1`.

Proof. ... Qed.

Hint Rewrite le_S.

Proof State: $\frac{w}{z} + 1 < k * a + 1$

What is auto?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

`Lemma le_S :` $\forall xy,$ `x < y` $\rightarrow$ `x + 1 < x + 1.`

Proof. ... Qed.

Hint Rewrite le_S.

Proof State: $\frac{w}{z} < k * a$

What is `auto`?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

What is `auto`?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

`Lemma plus_comm :` $\forall xy,$ `x + y` $=$ `y + x.`

Proof. ... Qed.

What is `auto`?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

`Lemma plus_comm :` $\forall xy,$ `x + y` $=$ `y + x.`

Proof. ... Qed.

Hint Rewrite plus_comm.

What is `auto`?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

`Lemma plus_comm :` $\forall xy,$ `x + y` $=$ `y + x.`

Proof. ... Qed.

Hint Rewrite plus_comm.

Proof State: $\frac{w}{p} + z = \mathsf{k}$

What is `auto`?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

`Lemma plus_comm :` $\forall xy,$ `x + y` `= y + x.`

Proof. ... Qed.

Hint Rewrite plus_comm.

Proof State: $\boxed{\frac{w}{p} + z} = k$

What is `auto`?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

`Lemma plus_comm :` $\forall xy,$ `x + y` $=$ $\boxed{\texttt{y + x}}$.

Proof. ... Qed.

Hint Rewrite plus_comm.

Proof State: $\boxed{z + \frac{w}{p}} = \mathsf{k}$

What is `auto`?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

What is `auto`?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

```
Hint Extern => tactic_expr
```

What is `auto`?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

Gating

```
Hint Extern =>
match goal with
| gate_expr => apply lemma
| _ => fail
end
```

What is `auto`?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

Gating

`Lemma le_trans :` $\forall xyz, x < y \rightarrow y < z \rightarrow x < z$

`Proof. ... Qed.`

What is `auto`?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

Gating

`Lemma le_trans :` $\forall xyz, x < y \rightarrow y < z \rightarrow x < z$

`Proof. ... Qed.`

`Hint Apply le_trans.`

What is `auto`?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

Gating

`Lemma le_trans :` $\forall xyz, x < y \rightarrow y < z \rightarrow x < z$

`Proof. ... Qed.`

~~`Hint Apply le_trans.`~~

What is auto?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

Gating

Lemma le_trans : $\forall xyz, x < y \rightarrow y < z \rightarrow x < z$

```
Proof.  ...  Qed.
Hint Extern =>
match goal with
| H1 : ?x < ?y , H2 : ?y < ?z |- ?x < ?z =>
        apply le_trans H1 H1
| _ => fail
end
```

What is auto?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

Gating

```
Lemma le_trans :  ∀xyz, x < y → y < z → x < z
Proof.  ...  Qed.
Hint Extern =>
match goal with
| H1 :  ?x < ?y, H2 :  ?y < ?z  |- ?x < ?z  =>
         apply le_trans H1 H1
| _ => fail
end
```

What is auto?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

Gating

```
Lemma le_trans :  ∀xyz, x < y → y < z → x < z
Proof.  ...  Qed.
Hint Extern =>
match goal with
| H1 :  ?x < ?y , H2 :  ?y < ?z  |- ?x < ?z  =>
        apply le_trans H1 H1
| _ => fail
end
```

What is auto?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

Gating

```
Lemma le_trans :  ∀xyz, x < y → y < z → x < z
Proof.   ...   Qed.
Hint Extern =>
match goal with
| H1 :  ?x < ?y ,  H2 :  ?y < ?z  |- ?x < ?z  =>
         apply le_trans H1 H1
| _ => fail
end
```

What is auto?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

Gating

Lemma le_trans : $\forall xyz, x < y \rightarrow y < z \rightarrow x < z$

```
Proof.  ...  Qed.
Hint Extern =>
match goal with
| H1 : ?x < ?y , H2 : ?y < ?z |- ?x < ?z =>
        apply le_trans H1 H1
| _ => fail
end
```

What is auto?

(and how does it differ from any other automation styles)

Simple, general purpose BFS/DFS proof search based on user-hints.

Gating

```
Lemma le_trans :  ∀xyz, x < y → y < z → x < z
Proof.   ...   Qed.
Hint Extern =>
match goal with
| H1 :  ?x < ?y ,  H2 :  ?y < ?z  |- ?x < ?z  =>
          apply le_trans H1 H1
| _ => fail
end
```

Rule of thumb

Keep `auto` branching factor below 1.5

(Otherwise `auto` would just be a bad version of other automation tactics)

Rule of thumb

Keep `auto` branching factor below 1.5
(Otherwise `auto` would just be a bad version of other automation tactics)

Practical experience

Branching factor < 10 easily achievable with gating

Branching factor < 2 increasingly hard for complex developments

Solution

Small-data online machine learning

Solution

Small-data online machine learning

Record the successful hints executed by previous `auto` runs

$$\Gamma_1 \vdash T_1 \Rightarrow \texttt{Hint Apply lemma} \Rightarrow \Gamma_2 \vdash T_2$$

Solution

Small-data online machine learning

Record the successful hints executed by previous `auto` runs

$\Gamma_1 \vdash T_1 \Rightarrow$ `Hint Apply lemma` $\Rightarrow \Gamma_2 \vdash T_2$

Modify the `auto` tactic

- ▷ On each branch point, execute all possible hints (expected +/- 10)

Solution

Small-data online machine learning

Record the successful hints executed by previous `auto` runs

$$\Gamma_1 \vdash T_1 \Rightarrow \texttt{Hint Apply lemma} \Rightarrow \Gamma_2 \vdash T_2$$

Modify the `auto` tactic

- ▷ On each branch point, execute all possible hints (expected +/- 10)
- ▷ Order them based on machine learning
- ▷ Pray to the ML-gods that branching factor is now « 2

Machine Learning

For each recorded triple

$$\Gamma_1 \vdash T_1 \Rightarrow \texttt{Hint} \ \ \ldots \quad \Rightarrow \Gamma_2 \vdash T_2$$

Machine Learning

For each recorded triple

$$\Gamma_1 \vdash T_1 \Rightarrow \texttt{Hint} \;\; \ldots \quad \Rightarrow \Gamma_2 \vdash T_2$$

calculate

$$\Gamma_\Delta \vdash T_\Delta = \Gamma_1 \vdash T_1 - \Gamma_2 \vdash T_2$$

Machine Learning

For each recorded triple

$$\Gamma_1 \vdash T_1 \Rightarrow \texttt{Hint} \ \ldots \quad \Rightarrow \Gamma_2 \vdash T_2$$

calculate

$$\Gamma_\Delta \vdash T_\Delta = \Gamma_1 \vdash T_1 - \Gamma_2 \vdash T_2$$

and execute a k-nearest neighbor on its features

Machine Learning

For each recorded triple

$$\Gamma_1 \vdash T_1 \Rightarrow \texttt{Hint} \ \ldots \quad \Rightarrow \Gamma_2 \vdash T_2$$

calculate

$$\Gamma_\Delta \vdash T_\Delta = \Gamma_1 \vdash T_1 - \Gamma_2 \vdash T_2$$

and execute a k-nearest neighbor on its features

Currently, subtraction is a pointwise textual diff for hypotheses and goal

Integrated into experimental Tactician version

Integrated into experimental Tactician version

The Tactician

- ▷ ML on tactic scripts
- ▷ Seamless integration into user workflow
- ▷ Suitable for proving-in-the-large
- ▷ Alpha version available
- ▷ Version 1.0 expected soon

`https://coq-tactician.github.io`

TODO

TODO

- ▷ Find a suitable large `auto`-focused development

TODO

- ▷ Find a suitable large `auto`-focused development
- ▷ Perform an evaluation to convince skeptics…
  - Have an interesting development? Contact me!

TODO

- ▷ Find a suitable large `auto`-focused development
- ▷ Perform an evaluation to convince skeptics...
  - Have an interesting development? Contact me!
- ▷ Move to more symbolic-based machine learning
  - Have an interesting small-scale ML technique? Contact me!

TODO

- Find a suitable large auto-focused development
- Perform an evaluation to convince skeptics...
  - Have an interesting development? Contact me!
- Move to more symbolic-based machine learning
  - Have an interesting small-scale ML technique? Contact me!

?