LIBTACTICS

A COLLECTION OF HANDY GENERAL-PURPOSE TACTICS

```
(* Chapter maintained by Arthur Chargueraud *)
```

This file contains a set of tactics that extends the set of builtin tactics provided with the standard distribution of Coq. It intends to overcome a number of limitations of the standard set of tactics, and thereby to help user to write shorter and more robust scripts.

Hopefully, Coq tactics will be improved as time goes by, and this file should ultimately be useless. In the meanwhile, serious Coq users will probably find it very useful.

The present file contains the implementation and the detailed documentation of those tactics. The SF reader need not read this file; instead, he/she is encouraged to read the chapter named UseTactics.v, which is gentle introduction to the most useful tactics from the LibTactic library.

The main features offered are:

- More convenient syntax for naming hypotheses, with tactics for introduction and inversion that take as input only the name of hypotheses of type `Prop`, rather than the name of all variables.
- Tactics providing true support for manipulating N-ary conjunctions, disjunctions and existentials, hidding the fact that the underlying implementation is based on binary propositions.
- Convenient support for automation: tactics followed with the symbol "~" or "*" will call automation on the generated subgoals. The symbol "~" stands for `auto` and "*" for `intuition eauto`. These bindings can be customized.
- Forward-chaining tactics are provided to instantiate lemmas either with variable or hypotheses or a mix of both.
- A more powerful implementation of `apply` is provided (it is based on `refine` and thus behaves better with respect to conversion).
- An improved inversion tactic which substitutes equalities on variables generated by the standard inversion mecanism. Moreover, it supports the elimination of dependently-typed equalities (requires axiom `K`, which is a weak form of Proof Irrelevance).

- Tactics for saving time when writing proofs, with tactics to asserts hypotheses or sub-goals, and improved tactics for clearing, renaming, and sorting hypotheses.

External credits:

- thanks to Xavier Leroy for providing the idea of tactic `forward`
- thanks to Georges Gonthier for the implementation trick in `rapply`

```
Set Implicit Arguments.

Require Import List.

(* Very important to remove hint trans_eq_bool from LibBool,
   otherwise eauto slows down dramatically:
  Lemma test : forall b, b = false.
  time eauto 7. (* takes over 4 seconds  to fail! *) *)

Remove Hints Bool.trans_eq_bool.
```

Tools for Programming with Ltac

Identity Continuation

```
Ltac idcont tt :=
  idtac.
```

Untyped Arguments for Tactics

Any Coq value can be boxed into the type `Boxer`. This is useful to use Coq computations for implementing tactics.

```
Inductive Boxer : Type :=
  | boxer : ∀ (A:Type), A → Boxer.
```

Optional Arguments for Tactics

`ltac_no_arg` is a constant that can be used to simulate optional arguments in tactic definitions. Use `mytactic ltac_no_arg` on the tactic invokation, and use `match arg with ltac_no_arg ⇒ ..` or `match type of arg with ltac_No_arg ⇒ ..` to test whether an argument was provided.

```
Inductive ltac_No_arg : Set :=
  | ltac_no_arg : ltac_No_arg.
```

Wildcard Arguments for Tactics

`ltac_wild` is a constant that can be used to simulate wildcard arguments in tactic definitions. Notation is __.

```
Inductive ltac_Wild : Set :=
  | ltac_wild : ltac_Wild.

Notation "'__'" := ltac_wild : ltac_scope.
```

`ltac_wilds` is another constant that is typically used to simulate a sequence of `N` wildcards, with `N` chosen appropriately depending on the context. Notation is ____.

```
Inductive ltac_Wilds : Set :=
  | ltac_wilds : ltac_Wilds.

Notation "'___'" := ltac_wilds : ltac_scope.

Open Scope ltac_scope.
```

Position Markers

`ltac_Mark` and `ltac_mark` are dummy definitions used as sentinel by tactics, to mark a certain position in the context or in the goal.

```
Inductive ltac_Mark : Type :=
  | ltac_mark : ltac_Mark.
```

`gen_until_mark` repeats `generalize` on hypotheses from the context, starting from the bottom and stopping as soon as reaching an hypothesis of type `Mark`. If fails if `Mark` does not appear in the context.

```
Ltac gen_until_mark :=
  match goal with H: ?T |- _ ⇒
  match T with
  | ltac_Mark ⇒ clear H
  | _ ⇒ generalize H; clear H; gen_until_mark
  end end.
```

`intro_until_mark` repeats `intro` until reaching an hypothesis of type `Mark`. It throws away the hypothesis `Mark`. It fails if `Mark` does not appear as an hypothesis in the goal.

```
Ltac intro_until_mark :=
  match goal with
  | |- (ltac_Mark → _) ⇒ intros _
  | _ ⇒ intro; intro_until_mark
  end.
```

List of Arguments for Tactics

A datatype of type `list Boxer` is used to manipulate list of Coq values in ltac. Notation is >> v_1 v_2 ... vN for building a list containing the values v_1 through vN.

```
Notation "'>>'" :=
  (@nil Boxer)
  (at level 0)
  : ltac_scope.
Notation "'>>' v1" :=
  ((boxer v1)::nil)
  (at level 0, v1 at level 0)
  : ltac_scope.
Notation "'>>' v1 v2" :=
  ((boxer v1)::(boxer v2)::nil)
  (at level 0, v1 at level 0, v2 at level 0)
  : ltac_scope.
```

```
Notation "'>>' v1 v2 v3" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0)
  : ltac_scope.
Notation "'>>' v1 v2 v3 v4" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
   v4 at level 0)
  : ltac_scope.
Notation "'>>' v1 v2 v3 v4 v5" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
   v4 at level 0, v5 at level 0)
  : ltac_scope.
Notation "'>>' v1 v2 v3 v4 v5 v6" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
   ::(boxer v6)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
   v4 at level 0, v5 at level 0, v6 at level 0)
  : ltac_scope.
Notation "'>>' v1 v2 v3 v4 v5 v6 v7" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
   ::(boxer v6)::(boxer v7)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
   v4 at level 0, v5 at level 0, v6 at level 0, v7 at level 0)
  : ltac_scope.
Notation "'>>' v1 v2 v3 v4 v5 v6 v7 v8" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
   ::(boxer v6)::(boxer v7)::(boxer v8)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
   v4 at level 0, v5 at level 0, v6 at level 0, v7 at level 0,
   v8 at level 0)
  : ltac_scope.
Notation "'>>' v1 v2 v3 v4 v5 v6 v7 v8 v9" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
   ::(boxer v6)::(boxer v7)::(boxer v8)::(boxer v9)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
   v4 at level 0, v5 at level 0, v6 at level 0, v7 at level 0,
   v8 at level 0, v9 at level 0)
  : ltac_scope.
Notation "'>>' v1 v2 v3 v4 v5 v6 v7 v8 v9 v10" :=
  ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
   ::(boxer v6)::(boxer v7)::(boxer v8)::(boxer v9)::(boxer
v10)::nil)
  (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
   v4 at level 0, v5 at level 0, v6 at level 0, v7 at level 0,
   v8 at level 0, v9 at level 0, v10 at level 0)
  : ltac_scope.
Notation "'>>' v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11" :=
```

```
    ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
     ::(boxer v6)::(boxer v7)::(boxer v8)::(boxer v9)::(boxer v10)
     ::(boxer v11)::nil)
    (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
     v4 at level 0, v5 at level 0, v6 at level 0, v7 at level 0,
     v8 at level 0, v9 at level 0, v10 at level 0, v11 at level 0)
    : ltac_scope.
Notation "'>>' v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12" :=
    ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
     ::(boxer v6)::(boxer v7)::(boxer v8)::(boxer v9)::(boxer v10)
     ::(boxer v11)::(boxer v12)::nil)
    (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
     v4 at level 0, v5 at level 0, v6 at level 0, v7 at level 0,
     v8 at level 0, v9 at level 0, v10 at level 0, v11 at level 0,
     v12 at level 0)
    : ltac_scope.
Notation "'>>' v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13" :=
    ((boxer v1)::(boxer v2)::(boxer v3)::(boxer v4)::(boxer v5)
     ::(boxer v6)::(boxer v7)::(boxer v8)::(boxer v9)::(boxer v10)
     ::(boxer v11)::(boxer v12)::(boxer v13)::nil)
    (at level 0, v1 at level 0, v2 at level 0, v3 at level 0,
     v4 at level 0, v5 at level 0, v6 at level 0, v7 at level 0,
     v8 at level 0, v9 at level 0, v10 at level 0, v11 at level 0,
     v12 at level 0, v13 at level 0)
    : ltac_scope.
```

The tactic `list_boxer_of` inputs a term `E` and returns a term of type "list boxer", according to the following rules:

- if `E` is already of type "list Boxer", then it returns `E`;
- otherwise, it returns the list `(boxer E)::nil`.

```
Ltac list_boxer_of E :=
  match type of E with
  | List.list Boxer ⇒ constr:(E)
  | _ ⇒ constr:((boxer E)::nil)
  end.
```

Databases of Lemmas

Use the hint facility to implement a database mapping terms to terms. To declare a new database, use a definition: `Definition mydatabase := True.`

Then, to map `mykey` to `myvalue`, write the hint: `Hint Extern 1 (Register mydatabase mykey) ⇒ Provide myvalue.`

Finally, to query the value associated with a key, run the tactic `ltac_database_get mydatabase mykey`. This will leave at the head of the goal the term `myvalue`. It can then be named and exploited using `intro`.

```
Inductive Ltac_database_token : Prop := ltac_database_token.
```

```
Definition ltac_database (D:Boxer) (T:Boxer) (A:Boxer) :=
Ltac_database_token.

Notation "'Register' D T" := (ltac_database (boxer D) (boxer T) _)
  (at level 69, D at level 0, T at level 0).

Lemma ltac_database_provide : ∀ (A:Boxer) (D:Boxer) (T:Boxer),
  ltac_database D T A.
Proof using. split. Qed.

Ltac Provide T := apply (@ltac_database_provide (boxer T)).

Ltac ltac_database_get D T :=
  let A := fresh "TEMP" in evar (A:Boxer);
  let H := fresh "TEMP" in
  assert (H : ltac_database (boxer D) (boxer T) A);
  [ subst A; auto
  | subst A; match type of H with ltac_database _ _ (boxer ?L) ⇒
               generalize L end; clear H ].

(* Note for a possible alternative implementation of the ltac_database_token:
   Inductive Ltac_database : Type :=
     | ltac_database : forall A, A -> Ltac_database.
   Implicit Arguments ltac_database A.
*)
```

On-the-Fly Removal of Hypotheses

In a list of arguments >> `H1 H2 .. HN` passed to a tactic such as `lets` or `applys` or `forwards` or `specializes`, the term `rm`, an identity function, can be placed in front of the name of an hypothesis to be deleted.

```
Definition rm (A:Type) (X:A) := X.
```

`rm_term E` removes one hypothesis that admits the same type as `E`.

```
Ltac rm_term E :=
  let T := type of E in
  match goal with H: T |- _ ⇒ try clear H end.
```

`rm_inside E` calls `rm_term Ei` for any subterm of the form `rm Ei` found in E

```
Ltac rm_inside E :=
  let go E := rm_inside E in
  match E with
  | rm ?X ⇒ rm_term X
  | ?X1 ?X2 ⇒
     go X1; go X2
  | ?X1 ?X2 ?X3 ⇒
     go X1; go X2; go X3
  | ?X1 ?X2 ?X3 ?X4 ⇒
     go X1; go X2; go X3; go X4
  | ?X1 ?X2 ?X3 ?X4 ?X5 ⇒
     go X1; go X2; go X3; go X4; go X5
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ⇒
     go X1; go X2; go X3; go X4; go X5; go X6
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X7 ⇒
```

```
     go X1; go X2; go X3; go X4; go X5; go X6; go X7
   | ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X7 ?X8 ⇒
     go X1; go X2; go X3; go X4; go X5; go X6; go X7; go X8
   | ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X7 ?X8 ?X9 ⇒
     go X1; go X2; go X3; go X4; go X5; go X6; go X7; go X8; go X9
   | ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X7 ?X8 ?X9 ?X10 ⇒
     go X1; go X2; go X3; go X4; go X5; go X6; go X7; go X8; go X9;
go X10
   | _ ⇒ idtac
  end.
```

For faster performance, one may deactivate `rm_inside` by replacing the body of this definition with `idtac`.

```
Ltac fast_rm_inside E :=
  rm_inside E.
```

Numbers as Arguments

When tactic takes a natural number as argument, it may be parsed either as a natural number or as a relative number. In order for tactics to convert their arguments into natural numbers, we provide a conversion tactic.

```
Require BinPos Coq.ZArith.BinInt.

Definition ltac_nat_from_int (x:BinInt.Z) : nat :=
  match x with
  | BinInt.Z0 ⇒ 0%nat
  | BinInt.Zpos p ⇒ BinPos.nat_of_P p
  | BinInt.Zneg p ⇒ 0%nat
  end.

Ltac nat_from_number N :=
  match type of N with
  | nat ⇒ constr:(N)
  | BinInt.Z ⇒ let N' := constr:(ltac_nat_from_int N) in eval
compute in N'
  end.
```

`ltac_pattern E at K` is the same as `pattern E at K` except that `K` is a Coq natural rather than a Ltac integer. Syntax `ltac_pattern E as K in H` is also available.

```
Tactic Notation "ltac_pattern" constr(E) "at" constr(K) :=
  match nat_from_number K with
  | 1 ⇒ pattern E at 1
  | 2 ⇒ pattern E at 2
  | 3 ⇒ pattern E at 3
  | 4 ⇒ pattern E at 4
  | 5 ⇒ pattern E at 5
  | 6 ⇒ pattern E at 6
  | 7 ⇒ pattern E at 7
  | 8 ⇒ pattern E at 8
  end.

Tactic Notation "ltac_pattern" constr(E) "at" constr(K) "in" hyp(H)
:=
  match nat_from_number K with
```

```
  | 1 ⇒ pattern E at 1 in H
  | 2 ⇒ pattern E at 2 in H
  | 3 ⇒ pattern E at 3 in H
  | 4 ⇒ pattern E at 4 in H
  | 5 ⇒ pattern E at 5 in H
  | 6 ⇒ pattern E at 6 in H
  | 7 ⇒ pattern E at 7 in H
  | 8 ⇒ pattern E at 8 in H
  end.
```

Testing Tactics

`show tac` executes a tactic `tac` that produces a result, and then display its result.

```
Tactic Notation "show" tactic(tac) :=
  let R := tac in pose R.
```

`dup N` produces `N` copies of the current goal. It is useful for building examples on which to illustrate behaviour of tactics. `dup` is short for `dup 2`.

```
Lemma dup_lemma : ∀ P, P → P → P.
Proof using. auto. Qed.

Ltac dup_tactic N :=
  match nat_from_number N with
  | 0 ⇒ idtac
  | S 0 ⇒ idtac
  | S ?N' ⇒ apply dup_lemma; [ | dup_tactic N' ]
  end.

Tactic Notation "dup" constr(N) :=
  dup_tactic N.
Tactic Notation "dup" :=
  dup 2.
```

Check No Evar in Goal

```
Ltac check_noevar M :=
  first [ has_evar M; fail 2 | idtac ].

Ltac check_noevar_hyp H := (* todo: imlement using check_noevar *)
  let T := type of H in check_noevar T.
Ltac check_noevar_goal := (* todo: imlement using check_noevar *)
  match goal with |- ?G ⇒ check_noevar G end.
```

Helper Function for Introducing Evars

`with_evar T (fun M ⇒ tac)` creates a new evar that can be used in the tactic `tac` under the name `M`.

```
Ltac with_evar_base T cont :=
  let x := fresh in evar (x:T); cont x; subst x.

Tactic Notation "with_evar" constr(T) tactic(cont) :=
  with_evar_base T cont.
```

Tagging of Hypotheses

`get_last_hyp tt` is a function that returns the last hypothesis at the bottom of the context. It is useful to obtain the default name associated with the hypothesis, e.g. `intro; let H := get_last_hyp tt in let H' := fresh "P" H in ...`

```
Ltac get_last_hyp tt :=
  match goal with H: _ |- _ ⇒ constr:(H) end.
```

More Tagging of Hypotheses

`ltac_tag_subst` is a specific marker for hypotheses which is used to tag hypotheses that are equalities to be substituted.

```
Definition ltac_tag_subst (A:Type) (x:A) := x.
```

`ltac_to_generalize` is a specific marker for hypotheses to be generalized.

```
Definition ltac_to_generalize (A:Type) (x:A) := x.

Ltac gen_to_generalize :=
  repeat match goal with
    H: ltac_to_generalize _ |- _ ⇒ generalize H; clear H end.

Ltac mark_to_generalize H :=
  let T := type of H in
  change T with (ltac_to_generalize T) in H.
```

Deconstructing Terms

`get_head E` is a tactic that returns the head constant of the term `E`, ie, when applied to a term of the form `P x1 ... xN` it returns `P`. If `E` is not an application, it returns `E`. Warning: the tactic seems to loop in some cases when the goal is a product and one uses the result of this function.

```
Ltac get_head E :=
  match E with
  | ?P _ _ _ _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ _ ⇒ constr:(P)
  | ?P _ _ _ ⇒ constr:(P)
  | ?P _ _ ⇒ constr:(P)
  | ?P _ ⇒ constr:(P)
  | ?P ⇒ constr:(P)
  end.
```

`get_fun_arg E` is a tactic that decomposes an application term `E`, ie, when applied to a term of the form `X1 ... XN` it returns a pair made of `X1 .. X(N-1)` and `XN`.

```
Ltac get_fun_arg E :=
  match E with
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X7 ?X ⇒ constr:((X1 X2 X3 X4 X5 X6,X))
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X6 ?X ⇒ constr:((X1 X2 X3 X4 X5,X))
```

```
  | ?X1 ?X2 ?X3 ?X4 ?X5 ?X ⇒ constr:((X1 X2 X3 X4,X))
  | ?X1 ?X2 ?X3 ?X4 ?X ⇒ constr:((X1 X2 X3,X))
  | ?X1 ?X2 ?X3 ?X ⇒ constr:((X1 X2,X))
  | ?X1 ?X2 ?X ⇒ constr:((X1,X))
  | ?X1 ?X ⇒ constr:((X1,X))
  end.
```

Action at Occurence and Action Not at Occurence

`ltac_action_at K of E do Tac` isolates the `K`-th occurence of `E` in the goal, setting it in the form `P E` for some named pattern `P`, then calls tactic `Tac`, and finally unfolds `P`. Syntax `ltac_action_at K of E in H do Tac` is also available.

```
Tactic Notation "ltac_action_at" constr(K) "of" constr(E) "do"
tactic(Tac) :=
  let p := fresh in ltac_pattern E at K;
  match goal with |- ?P _ ⇒ set (p:=P) end;
  Tac; unfold p; clear p.

Tactic Notation "ltac_action_at" constr(K) "of" constr(E) "in"
hyp(H) "do" tactic(Tac) :=
  let p := fresh in ltac_pattern E at K in H;
  match type of H with ?P _ ⇒ set (p:=P) in H end;
  Tac; unfold p in H; clear p.
```

`protects E do Tac` temporarily assigns a name to the expression `E` so that the execution of tactic `Tac` will not modify `E`. This is useful for instance to restrict the action of `simpl`.

```
Tactic Notation "protects" constr(E) "do" tactic(Tac) :=
  (* let x := fresh "TEMP" in sets_eq x: E; T; subst x. *)
  let x := fresh "TEMP" in let H := fresh "TEMP" in
  set (X := E) in *; assert (H : X = E) by reflexivity;
  clearbody X; Tac; subst x.

Tactic Notation "protects" constr(E) "do" tactic(Tac) "/" :=
  protects E do Tac.
```

An Alias for `eq`

`eq'` is an alias for `eq` to be used for equalities in inductive definitions, so that they don't get mixed with equalities generated by `inversion`.

```
Definition eq' := @eq.

Hint Unfold eq'.

Notation "x '='' y" := (@eq' _ x y)
  (at level 70, y at next level).
```

Common Tactics for Simplifying Goals Like `intuition`

```
Ltac jauto_set_hyps :=
  repeat match goal with H: ?T |- _ ⇒
    match T with
    | _ ∧ _ ⇒ destruct H
    | ∃ a, _ ⇒ destruct H
    | _ ⇒ generalize H; clear H
    end
  end.

Ltac jauto_set_goal :=
  repeat match goal with
  | |- ∃ a, _ ⇒ esplit
  | |- _ ∧ _ ⇒ split
  end.

Ltac jauto_set :=
  intros; jauto_set_hyps;
  intros; jauto_set_goal;
  unfold not in *.
```

Backward and Forward Chaining

Application

```
Ltac old_refine f :=
  refine f. (* ; shelve_unifiable. *)
```

`rapply` is a tactic similar to `eapply` except that it is based on the `refine` tactics, and thus is strictly more powerful (at least in theory :). In short, it is able to perform on-the-fly conversions when required for arguments to match, and it is able to instantiate existentials when required.

```
Tactic Notation "rapply" constr(t) :=
  first (* todo: les @ sont inutiles *)
  [ eexact (@t)
  | refine (@t)
  | refine (@t _)
  | refine (@t _ _)
  | refine (@t _ _ _)
  | refine (@t _ _ _ _)
  | refine (@t _ _ _ _ _)
  | refine (@t _ _ _ _ _ _)
  | refine (@t _ _ _ _ _ _ _)
  | refine (@t _ _ _ _ _ _ _ _)
  | refine (@t _ _ _ _ _ _ _ _ _)
  | refine (@t _ _ _ _ _ _ _ _ _ _)
  | refine (@t _ _ _ _ _ _ _ _ _ _ _)
  | refine (@t _ _ _ _ _ _ _ _ _ _ _ _)
  | refine (@t _ _ _ _ _ _ _ _ _ _ _ _ _)
  | refine (@t _ _ _ _ _ _ _ _ _ _ _ _ _ _)
  | refine (@t _ _ _ _ _ _ _ _ _ _ _ _ _ _ _)
  ].
```

The tactics applys_N T, where N is a natural number, provides a more efficient way of using applys T. It avoids trying out all possible arities, by specifying explicitely the arity of function T.

```
Tactic Notation "rapply_0" constr(t) :=
  refine (@t).
Tactic Notation "rapply_1" constr(t) :=
  refine (@t _).
Tactic Notation "rapply_2" constr(t) :=
  refine (@t _ _).
Tactic Notation "rapply_3" constr(t) :=
  refine (@t _ _ _).
Tactic Notation "rapply_4" constr(t) :=
  refine (@t _ _ _ _).
Tactic Notation "rapply_5" constr(t) :=
  refine (@t _ _ _ _ _).
Tactic Notation "rapply_6" constr(t) :=
  refine (@t _ _ _ _ _ _).
Tactic Notation "rapply_7" constr(t) :=
  refine (@t _ _ _ _ _ _ _).
Tactic Notation "rapply_8" constr(t) :=
  refine (@t _ _ _ _ _ _ _ _).
Tactic Notation "rapply_9" constr(t) :=
  refine (@t _ _ _ _ _ _ _ _ _).
Tactic Notation "rapply_10" constr(t) :=
  refine (@t _ _ _ _ _ _ _ _ _ _).
```

lets_base H E adds an hypothesis H : T to the context, where T is the type of term E. If H is an introduction pattern, it will destruct H according to the pattern.

```
Ltac lets_base I E := generalize E; intros I.
```

applys_to H E transform the type of hypothesis H by replacing it by the result of the application of the term E to H. Intuitively, it is equivalent to lets H: (E H).

```
Tactic Notation "applys_to" hyp(H) constr(E) :=
  let H' := fresh in rename H into H';
  (first [ lets_base H (E H')
         | lets_base H (E _ H')
         | lets_base H (E _ _ H')
         | lets_base H (E _ _ _ H')
         | lets_base H (E _ _ _ _ H')
         | lets_base H (E _ _ _ _ _ H')
         | lets_base H (E _ _ _ _ _ _ H')
         | lets_base H (E _ _ _ _ _ _ _ H')
         | lets_base H (E _ _ _ _ _ _ _ _ H')
         | lets_base H (E _ _ _ _ _ _ _ _ _ H') ]
  ); clear H'.
```

applys_to H1,...,HN E applys E to several hypotheses

```
Tactic Notation "applys_to" hyp(H1) "," hyp(H2) constr(E) :=
  applys_to H1 E; applys_to H2 E.
Tactic Notation "applys_to" hyp(H1) "," hyp(H2) "," hyp(H3)
constr(E) :=
  applys_to H1 E; applys_to H2 E; applys_to H3 E.
```

```
Tactic Notation "applys_to" hyp(H1) "," hyp(H2) "," hyp(H3) ","
hyp(H4) constr(E) :=
  applys_to H1 E; applys_to H2 E; applys_to H3 E; applys_to H4 E.
```

`constructors` calls `constructor` or `econstructor`.

```
Tactic Notation "constructors" :=
  first [ constructor | econstructor ]; unfold eq'.
```

Assertions

`asserts H: T` is another syntax for `assert (H : T)`, which also works with introduction patterns. For instance, one can write: `asserts \[x P\] (∃n, n = 3)`, or `asserts \[H|H\] (n = 0 ∨ n = 1)`.

```
Tactic Notation "asserts" simple_intropattern(I) ":" constr(T) :=
  let H := fresh in assert (H : T);
  [ | generalize H; clear H; intros I ].
```

`asserts H1 .. HN: T` is a shorthand for `asserts \[H1 \[H2 \[.. HN\]\]\]\`: T].

```
Tactic Notation "asserts" simple_intropattern(I1)
 simple_intropattern(I2) ":" constr(T) :=
  asserts [I1 I2]: T.
Tactic Notation "asserts" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3) ":" constr(T) :=
  asserts [I1 [I2 I3]]: T.
Tactic Notation "asserts" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3)
 simple_intropattern(I4) ":" constr(T) :=
  asserts [I1 [I2 [I3 I4]]]: T.
Tactic Notation "asserts" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3)
 simple_intropattern(I4) simple_intropattern(I5) ":" constr(T) :=
  asserts [I1 [I2 [I3 [I4 I5]]]]: T.
Tactic Notation "asserts" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3)
 simple_intropattern(I4) simple_intropattern(I5)
 simple_intropattern(I6) ":" constr(T) :=
  asserts [I1 [I2 [I3 [I4 [I5 I6]]]]]: T.
```

`asserts: T` is `asserts H: T` with `H` being chosen automatically.

```
Tactic Notation "asserts" ":" constr(T) :=
  let H := fresh in asserts H : T.
```

`cuts H: T` is the same as `asserts H: T` except that the two subgoals generated are swapped: the subgoal `T` comes second. Note that contrary to `cut`, it introduces the hypothesis.

```
Tactic Notation "cuts" simple_intropattern(I) ":" constr(T) :=
  cut (T); [ intros I | idtac ].
```

cuts: T is cuts H: T with H being chosen automatically.

```
Tactic Notation "cuts" ":" constr(T) :=
  let H := fresh in cuts H: T.
```

cuts H_1 .. HN: T is a shorthand for cuts \[H_1 \[H_2 \[.. HN\]\]\]\: T].

```
Tactic Notation "cuts" simple_intropattern(I1)
 simple_intropattern(I2) ":" constr(T) :=
  cuts [I1 I2]: T.
Tactic Notation "cuts" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3) ":" constr(T) :=
  cuts [I1 [I2 I3]]: T.
Tactic Notation "cuts" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3)
 simple_intropattern(I4) ":" constr(T) :=
  cuts [I1 [I2 [I3 I4]]]: T.
Tactic Notation "cuts" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3)
 simple_intropattern(I4) simple_intropattern(I5) ":" constr(T) :=
  cuts [I1 [I2 [I3 [I4 I5]]]]: T.
Tactic Notation "cuts" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3)
 simple_intropattern(I4) simple_intropattern(I5)
 simple_intropattern(I6) ":" constr(T) :=
  cuts [I1 [I2 [I3 [I4 [I5 I6]]]]]: T.
```

Instantiation and Forward-Chaining

The instantiation tactics are used to instantiate a lemma E (whose type is a product) on some arguments. The type of E is made of implications and universal quantifications, e.g. ∀x, P x → ∀y z, Q x y z → R z.

The first possibility is to provide arguments in order: first x, then a proof of P x, then y etc... In this mode, called "Args", all the arguments are to be provided. If a wildcard is provided (written ___), then an existential variable will be introduced in place of the argument.

It is very convenient to give some arguments the lemma should be instantiated on, and let the tactic find out automatically where underscores should be insterted. Underscore arguments ___ are interpret as follows: an underscore means that we want to skip the argument that has the same type as the next real argument provided (real means not an underscore). If there is no real argument after underscore, then the underscore is used for the first possible argument.

The general syntax is tactic (>> E_1 .. EN) where tactic is the name of the tactic (possibly with some arguments) and Ei are the arguments. Moreover, some tactics accept the syntax tactic E_1 .. EN as short for tactic (>> E_1 .. EN) for values of N up to 5.

Finally, if the argument `EN` given is a triple-underscore ___, then it is equivalent to providing a list of wildcards, with the appropriate number of wildcards. This means that all the remaining arguments of the lemma will be instantiated. Definitions in the conclusion are not unfolded in this case.

```
(* Underlying implementation *)

Ltac app_assert t P cont :=
  let H := fresh "TEMP" in
  assert (H : P); [ | cont(t H); clear H ].

Ltac app_evar t A cont :=
  let x := fresh "TEMP" in
  evar (x:A);
  let t' := constr:(t x) in
  let t'' := (eval unfold x in t') in
  subst x; cont t''.

Ltac app_arg t P v cont :=
  let H := fresh "TEMP" in
  assert (H : P); [ apply v | cont(t H); try clear H ].

Ltac build_app_alls t final :=
  let rec go t :=
    match type of t with
    | ?P → ?Q ⇒ app_assert t P go
    | ∀ _:?A, _ ⇒ app_evar t A go
    | _ ⇒ final t
    end in
  go t.

Ltac boxerlist_next_type vs :=
  match vs with
  | nil ⇒ constr:(ltac_wild)
  | (boxer ltac_wild)::?vs' ⇒ boxerlist_next_type vs'
  | (boxer ltac_wilds)::_ ⇒ constr:(ltac_wild)
  | (@boxer ?T _)::_ ⇒ constr:(T)
  end.

(* Note: refuse to instantiate a dependent hypothesis with a proposition;
    refuse to instantiate an argument of type Type with one that
    does not have the type Type.
*)

Ltac build_app_hnts t vs final :=
  let rec go t vs :=
    match vs with
    | nil ⇒ first [ final t | fail 1 ]
    | (boxer ltac_wilds)::_ ⇒ first [ build_app_alls t final |
fail 1 ]
    | (boxer ?v)::?vs' ⇒
      let cont t' := go t' vs in
      let cont' t' := go t' vs' in
      let T := type of t in
      let T := eval hnf in T in
      match v with
      | ltac_wild ⇒
         first [ let U := boxerlist_next_type vs' in
           match U with
           | ltac_wild ⇒
```

```
                match T with
                | ?P → ?Q ⇒ first [ app_assert t P cont' | fail 3 ]
                | ∀ _:?A, _ ⇒ first [ app_evar t A cont' | fail 3 ]
                end
              | _ ⇒
                match T with (* should test T for unifiability *)
                | U → ?Q ⇒ first [ app_assert t U cont' | fail 3 ]
                | ∀ _:U, _ ⇒ first [ app_evar t U cont' | fail 3 ]
                | ?P → ?Q ⇒ first [ app_assert t P cont | fail 3 ]
                | ∀ _:?A, _ ⇒ first [ app_evar t A cont | fail 3 ]
                end
              end
            | fail 2 ]
          | _ ⇒
             match T with
             | ?P → ?Q ⇒ first [ app_arg t P v cont'
                                 | app_assert t P cont
                                 | fail 3 ]
             | ∀ _:Type, _ ⇒
                match type of v with
                | Type ⇒ first [ cont' (t v)
                                 | app_evar t Type cont
                                 | fail 3 ]
                | _ ⇒ first [ app_evar t Type cont
                             | fail 3 ]
                end
             | ∀ _:?A, _ ⇒
                let V := type of v in
                match type of V with
                | Prop ⇒ first [ app_evar t A cont
                                 | fail 3 ]
                | _ ⇒ first [ cont' (t v)
                             | app_evar t A cont
                             | fail 3 ]
                end
             end
        end
      end in
    go t vs.
```

newer version : support for typeclasses

```
Ltac app_typeclass t cont :=
  let t' := constr:(t _) in
  cont t'.

Ltac build_app_alls t final ::=
  let rec go t :=
    match type of t with
    | ?P → ?Q ⇒ app_assert t P go
    | ∀ _:?A, _ ⇒
        first [ app_evar t A go
              | app_typeclass t go
              | fail 3 ]
    | _ ⇒ final t
    end in
  go t.
```

```
Ltac build_app_hnts t vs final ::=
  let rec go t vs :=
    match vs with
    | nil ⇒ first [ final t | fail 1 ]
    | (boxer ltac_wilds)::_ ⇒ first [ build_app_alls t final |
fail 1 ]
    | (boxer ?v)::?vs' ⇒
      let cont t' := go t' vs in
      let cont' t' := go t' vs' in
      let T := type of t in
      let T := eval hnf in T in
      match v with
      | ltac_wild ⇒
         first [ let U := boxerlist_next_type vs' in
           match U with
           | ltac_wild ⇒
             match T with
             | ?P → ?Q ⇒ first [ app_assert t P cont' | fail 3 ]
             | ∀ _:?A, _ ⇒ first [ app_typeclass t cont'
                                  | app_evar t A cont'
                                  | fail 3 ]
             end
           | _ ⇒
             match T with (* should test T for unifiability *)
             | U → ?Q ⇒ first [ app_assert t U cont' | fail 3 ]
             | ∀ _:U, _ ⇒ first
                 [ app_typeclass t cont'
                 | app_evar t U cont'
                 | fail 3 ]
             | ?P → ?Q ⇒ first [ app_assert t P cont | fail 3 ]
             | ∀ _:?A, _ ⇒ first
                 [ app_typeclass t cont
                 | app_evar t A cont
                 | fail 3 ]
             end
           end
         | fail 2 ]
      | _ ⇒
         match T with
         | ?P → ?Q ⇒ first [ app_arg t P v cont'
                            | app_assert t P cont
                            | fail 3 ]
          | ∀ _:Type, _ ⇒
             match type of v with
             | Type ⇒ first [ cont' (t v)
                            | app_evar t Type cont
                            | fail 3 ]
             | _ ⇒ first [ app_evar t Type cont
                         | fail 3 ]
             end
         | ∀ _:?A, _ ⇒
            let V := type of v in
            match type of V with
            | Prop ⇒ first [ app_typeclass t cont
                           | app_evar t A cont
                           | fail 3 ]
            | _ ⇒ first [ cont' (t v)
                        | app_typeclass t cont
                        | app_evar t A cont
```

```
                                  | fail 3 ]
              end
           end
        end
      end in
    go t vs.
    (* todo: use local function for first ... *)

(*--old version
Ltac build_app_hnts t vs final :=
  let rec go t vs :=
    match vs with
    | nil => first  final t | fail 1
    | (boxer ltac_wilds)::_ => first  build_app_alls t final | fail
1
    | (boxer ?v)::?vs' =>
      let cont t' := go t' vs in
      let cont' t' := go t' vs' in
      let T := type of t in
      let T := eval hnf in T in
      match v with
      | ltac_wild =>
         first  let U := boxerlist_next_type vs' in match U with |
ltac_wild ⇒ match T with | ?P → ?Q ⇒ first [ app_assert t P cont'
| fail 3 ] | ∀_:?A, _ ⇒ first [ app_evar t A cont' | fail 3 ] end
| _ ⇒ match T with  should test T for unifiability *)
| U → ?Q ⇒ first [ app_assert t U cont' | fail 3 ] | ∀_:U, _ ⇒
first [ app_evar t U cont' | fail 3 ] | ?P → ?Q ⇒ first [
app_assert t P cont | fail 3 ] | ∀_:?A, _ ⇒ first [ app_evar t A
cont | fail 3 ] end end | fail 2
      | _ =>
          match T with
          | ?P -> ?Q => first  app_arg t P v cont' | app_assert t P
cont | fail 3
          | forall _:?A, _ => first  cont' (t v) | app_evar t A
cont | fail 3
          end
      end
    end in
  go t vs.
*)

Ltac build_app args final :=
  first [
    match args with (@boxer ?T ?t)::?vs ⇒
      let t := constr:(t:T) in
      build_app_hnts t vs final;
      fast_rm_inside args
    end
  | fail 1 "Instantiation fails for:" args].

Ltac unfold_head_until_product T :=
  eval hnf in T.

Ltac args_unfold_head_if_not_product args :=
  match args with (@boxer ?T ?t)::?vs ⇒
    let T' := unfold_head_until_product T in
    constr:((@boxer T' t)::vs)
  end.
```

```
Ltac args_unfold_head_if_not_product_but_params args :=
  match args with
  | (boxer ?t)::(boxer ?v)::?vs ⇒
     args_unfold_head_if_not_product args
  | _ ⇒ constr:(args)
  end.
```

`lets H: (>> E0 E1 .. EN)` will instantiate lemma `E0` on the arguments `Ei` (which may be wildcards `__`), and name `H` the resulting term. `H` may be an introduction pattern, or a sequence of introduction patterns `I1 I2 IN`, or empty. Syntax `lets H: E0 E1 .. EN` is also available. If the last argument `EN` is `___` (triple-underscore), then all arguments of `H` will be instantiated.

```
Ltac lets_build I Ei :=
  let args := list_boxer_of Ei in
  let args := args_unfold_head_if_not_product_but_params args in
(*    let Ei''' := args_unfold_head_if_not_product Ei'' in*)
  build_app args ltac:(fun R ⇒ lets_base I R).

Tactic Notation "lets" simple_intropattern(I) ":" constr(E) :=
  lets_build I E.
Tactic Notation "lets" ":" constr(E) :=
  let H := fresh in lets H: E.
Tactic Notation "lets" ":" constr(E0)
 constr(A1) :=
  lets: (>> E0 A1).
Tactic Notation "lets" ":" constr(E0)
 constr(A1) constr(A2) :=
  lets: (>> E0 A1 A2).
Tactic Notation "lets" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) :=
  lets: (>> E0 A1 A2 A3).
Tactic Notation "lets" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) :=
  lets: (>> E0 A1 A2 A3 A4).
Tactic Notation "lets" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  lets: (>> E0 A1 A2 A3 A4 A5).

(* --todo: deprecated, do not use *)
Tactic Notation "lets" simple_intropattern(I1)
simple_intropattern(I2)
 ":" constr(E) :=
  lets [I1 I2]: E.
Tactic Notation "lets" simple_intropattern(I1)
simple_intropattern(I2)
 simple_intropattern(I3) ":" constr(E) :=
  lets [I1 [I2 I3]]: E.
Tactic Notation "lets" simple_intropattern(I1)
simple_intropattern(I2)
 simple_intropattern(I3) simple_intropattern(I4) ":" constr(E) :=
```

```
    lets [I1 [I2 [I3 I4]]]: E.
  Tactic Notation "lets" simple_intropattern(I1)
  simple_intropattern(I2)
   simple_intropattern(I3) simple_intropattern(I4)
  simple_intropattern(I5)
   ":" constr(E) :=
    lets [I1 [I2 [I3 [I4 I5]]]]: E.

  Tactic Notation "lets" simple_intropattern(I) ":" constr(E0)
   constr(A1) :=
    lets I: (>> E0 A1).
  Tactic Notation "lets" simple_intropattern(I) ":" constr(E0)
   constr(A1) constr(A2) :=
    lets I: (>> E0 A1 A2).
  Tactic Notation "lets" simple_intropattern(I) ":" constr(E0)
   constr(A1) constr(A2) constr(A3) :=
    lets I: (>> E0 A1 A2 A3).
  Tactic Notation "lets" simple_intropattern(I) ":" constr(E0)
   constr(A1) constr(A2) constr(A3) constr(A4) :=
    lets I: (>> E0 A1 A2 A3 A4).
  Tactic Notation "lets" simple_intropattern(I) ":" constr(E0)
   constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
    lets I: (>> E0 A1 A2 A3 A4 A5).

  Tactic Notation "lets" simple_intropattern(I1)
  simple_intropattern(I2) ":" constr(E0)
   constr(A1) :=
    lets [I1 I2]: E0 A1.
  Tactic Notation "lets" simple_intropattern(I1)
  simple_intropattern(I2) ":" constr(E0)
   constr(A1) constr(A2) :=
    lets [I1 I2]: E0 A1 A2.
  Tactic Notation "lets" simple_intropattern(I1)
  simple_intropattern(I2) ":" constr(E0)
   constr(A1) constr(A2) constr(A3) :=
    lets [I1 I2]: E0 A1 A2 A3.
  Tactic Notation "lets" simple_intropattern(I1)
  simple_intropattern(I2) ":" constr(E0)
   constr(A1) constr(A2) constr(A3) constr(A4) :=
    lets [I1 I2]: E0 A1 A2 A3 A4.
  Tactic Notation "lets" simple_intropattern(I1)
  simple_intropattern(I2) ":" constr(E0)
   constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
    lets [I1 I2]: E0 A1 A2 A3 A4 A5.
```

`forwards H: (>> E0 E1 .. EN)` is short for `forwards H: (>> E0 E1 .. EN ___)`. The arguments `Ei` can be wildcards `__` (except `E0`). `H` may be an introduction pattern, or a sequence of introduction pattern, or empty. Syntax `forwards H: E0 E1 .. EN` is also available.

```
Ltac forwards_build_app_arg Ei :=
  let args := list_boxer_of Ei in
  let args := (eval simpl in (args ++ ((boxer ___)::nil))) in
  let args := args_unfold_head_if_not_product args in
  args.

Ltac forwards_then Ei cont :=
  let args := forwards_build_app_arg Ei in
  let args := args_unfold_head_if_not_product_but_params args in
  build_app args cont.

Tactic Notation "forwards" simple_intropattern(I) ":" constr(Ei) :=
  let args := forwards_build_app_arg Ei in
  lets I: args.

Tactic Notation "forwards" ":" constr(E) :=
  let H := fresh in forwards H: E.
Tactic Notation "forwards" ":" constr(E0)
 constr(A1) :=
  forwards: (>> E0 A1).
Tactic Notation "forwards" ":" constr(E0)
 constr(A1) constr(A2) :=
  forwards: (>> E0 A1 A2).
Tactic Notation "forwards" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) :=
  forwards: (>> E0 A1 A2 A3).
Tactic Notation "forwards" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) :=
  forwards: (>> E0 A1 A2 A3 A4).
Tactic Notation "forwards" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  forwards: (>> E0 A1 A2 A3 A4 A5).

(* todo: deprecated, do not use *)
Tactic Notation "forwards" simple_intropattern(I1)
simple_intropattern(I2)
 ":" constr(E) :=
  forwards [I1 I2]: E.
Tactic Notation "forwards" simple_intropattern(I1)
simple_intropattern(I2)
 simple_intropattern(I3) ":" constr(E) :=
  forwards [I1 [I2 I3]]: E.
Tactic Notation "forwards" simple_intropattern(I1)
simple_intropattern(I2)
 simple_intropattern(I3) simple_intropattern(I4) ":" constr(E) :=
  forwards [I1 [I2 [I3 I4]]]: E.
Tactic Notation "forwards" simple_intropattern(I1)
simple_intropattern(I2)
 simple_intropattern(I3) simple_intropattern(I4)
```

```
  simple_intropattern(I5)
   ":" constr(E) :=
    forwards [I1 [I2 [I3 [I4 I5]]]]: E.

  Tactic Notation "forwards" simple_intropattern(I) ":" constr(E0)
   constr(A1) :=
    forwards I: (>> E0 A1).
  Tactic Notation "forwards" simple_intropattern(I) ":" constr(E0)
   constr(A1) constr(A2) :=
    forwards I: (>> E0 A1 A2).
  Tactic Notation "forwards" simple_intropattern(I) ":" constr(E0)
   constr(A1) constr(A2) constr(A3) :=
    forwards I: (>> E0 A1 A2 A3).
  Tactic Notation "forwards" simple_intropattern(I) ":" constr(E0)
   constr(A1) constr(A2) constr(A3) constr(A4) :=
    forwards I: (>> E0 A1 A2 A3 A4).
  Tactic Notation "forwards" simple_intropattern(I) ":" constr(E0)
   constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
    forwards I: (>> E0 A1 A2 A3 A4 A5).

    (* for use by tactics -- todo: factorize better *)
  Tactic Notation "forwards_nounfold" simple_intropattern(I) ":"
  constr(Ei) :=
    let args := list_boxer_of Ei in
    let args := (eval simpl in (args ++ ((boxer ___)::nil))) in
    build_app args ltac:(fun R ⇒ lets_base I R).

  Ltac forwards_nounfold_then Ei cont :=
    let args := list_boxer_of Ei in
    let args := (eval simpl in (args ++ ((boxer ___)::nil))) in
    build_app args cont.
```

applys (>> E_0 E_1 .. EN) instantiates lemma E_0 on the arguments Ei (which may be wildcards ___), and apply the resulting term to the current goal, using the tactic applys defined earlier on. applys E_0 E_1 E_2 .. EN is also available.

```
  Ltac applys_build Ei :=
    let args := list_boxer_of Ei in
    let args := args_unfold_head_if_not_product_but_params args in
    build_app args ltac:(fun R ⇒
     first [ apply R | eapply R | rapply R ]).

  Ltac applys_base E :=
    match type of E with
    | list Boxer ⇒ applys_build E
    | _ ⇒ first [ rapply E | applys_build E ]
    end; fast_rm_inside E.

  Tactic Notation "applys" constr(E) :=
    applys_base E.
  Tactic Notation "applys" constr(E0) constr(A1) :=
    applys (>> E0 A1).
  Tactic Notation "applys" constr(E0) constr(A1) constr(A2) :=
    applys (>> E0 A1 A2).
```

```
Tactic Notation "applys" constr(E0) constr(A1) constr(A2) constr(A3)
:=
  applys (>> E0 A1 A2 A3).
Tactic Notation "applys" constr(E0) constr(A1) constr(A2) constr(A3)
constr(A4) :=
  applys (>> E0 A1 A2 A3 A4).
Tactic Notation "applys" constr(E0) constr(A1) constr(A2) constr(A3)
constr(A4) constr(A5) :=
  applys (>> E0 A1 A2 A3 A4 A5).
```

`fapplys (>> E0 E1 .. EN)` instantiates lemma `E0` on the arguments `Ei` and on the argument `___` meaning that all evars should be explicitly instantiated, and apply the resulting term to the current goal. `fapplys E0 E1 E2 .. EN` is also available.

```
Ltac fapplys_build Ei :=
  let args := list_boxer_of Ei in
  let args := (eval simpl in (args ++ ((boxer ___)::nil))) in
  let args := args_unfold_head_if_not_product_but_params args in
  build_app args ltac:(fun R ⇒ apply R).

Tactic Notation "fapplys" constr(E0) :=
(* todo: use the tactic for that*)
  match type of E0 with
  | list Boxer ⇒ fapplys_build E0
  | _ ⇒ fapplys_build (>> E0)
  end.
Tactic Notation "fapplys" constr(E0) constr(A1) :=
  fapplys (>> E0 A1).
Tactic Notation "fapplys" constr(E0) constr(A1) constr(A2) :=
  fapplys (>> E0 A1 A2).
Tactic Notation "fapplys" constr(E0) constr(A1) constr(A2)
constr(A3) :=
  fapplys (>> E0 A1 A2 A3).
Tactic Notation "fapplys" constr(E0) constr(A1) constr(A2)
constr(A3) constr(A4) :=
  fapplys (>> E0 A1 A2 A3 A4).
Tactic Notation "fapplys" constr(E0) constr(A1) constr(A2)
constr(A3) constr(A4) constr(A5) :=
  fapplys (>> E0 A1 A2 A3 A4 A5).
```

`specializes H (>> E1 E2 .. EN)` will instantiate hypothesis `H` on the arguments `Ei` (which may be wildcards `__`). If the last argument `EN` is `___` (triple-underscore), then all arguments of `H` get instantiated.

```
Ltac specializes_build H Ei :=
  let H' := fresh "TEMP" in rename H into H';
  let args := list_boxer_of Ei in
  let args := constr:((boxer H')::args) in
  let args := args_unfold_head_if_not_product args in
  build_app args ltac:(fun R ⇒ lets H: R);
  clear H'.
```

```
Ltac specializes_base H Ei :=
  specializes_build H Ei; fast_rm_inside Ei.

Tactic Notation "specializes" hyp(H) :=
  specializes_base H (___).
Tactic Notation "specializes" hyp(H) constr(A) :=
  specializes_base H A.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) :=
  specializes H (>> A1 A2).
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2)
constr(A3) :=
  specializes H (>> A1 A2 A3).
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2)
constr(A3) constr(A4) :=
  specializes H (>> A1 A2 A3 A4).
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2)
constr(A3) constr(A4) constr(A5) :=
  specializes H (>> A1 A2 A3 A4 A5).
```

`specializes_vars H` is equivalent to `specializes H __ .. __` with as many double underscore as the number of dependent arguments visible from the type of `H`. Note that no unfolding is currently being performed (this behavior might change in the future). The current implementation is restricted to the case where `H` is an existing hypothesis — TODO: generalize.

```
Ltac specializes_var_base H :=
  match type of H with
  | ?P → ?Q ⇒ fail 1
  | ∀ _:_, _ ⇒ specializes H __
  end.

Ltac specializes_vars_base H :=
  repeat (specializes_var_base H).

Tactic Notation "specializes_var" hyp(H) :=
  specializes_var_base H.

Tactic Notation "specializes_vars" hyp(H) :=
  specializes_vars_base H.
```

Experimental Tactics for Application

`fapply` is a version of `apply` based on `forwards`.

```
Tactic Notation "fapply" constr(E) :=
  let H := fresh in forwards H: E;
  first [ apply H | eapply H | rapply H | hnf; apply H
        | hnf; eapply H | applys H ].
   (* todo: is applys redundant with rapply ? *)
```

`sapply` stands for "super apply". It tries `apply`, `eapply`, `applys` and `fapply`, and also tries to head-normalize the goal first.

```
Tactic Notation "sapply" constr(H) :=
  first [ apply H | eapply H | rapply H | applys H
```

```
          | hnf; apply H | hnf; eapply H | hnf; applys H
          | fapply H ].
```

Adding Assumptions

`lets_simpl H: E` is the same as `lets H: E` excepts that it calls `simpl` on the hypothesis H. `lets_simpl: E` is also provided.

```
Tactic Notation "lets_simpl" ident(H) ":" constr(E) :=
  lets H: E; try simpl in H.

Tactic Notation "lets_simpl" ":" constr(T) :=
  let H := fresh in lets_simpl H: T.
```

`lets_hnf H: E` is the same as `lets H: E` excepts that it calls `hnf` to set the definition in head normal form. `lets_hnf: E` is also provided.

```
Tactic Notation "lets_hnf" ident(H) ":" constr(E) :=
  lets H: E; hnf in H.

Tactic Notation "lets_hnf" ":" constr(T) :=
  let H := fresh in lets_hnf H: T.
```

`puts X: E` is a synonymous for `pose (X := E)`. Alternative syntax is `puts: E`.

```
Tactic Notation "puts" ident(X) ":" constr(E) :=
  pose (X := E).
Tactic Notation "puts" ":" constr(E) :=
  let X := fresh "X" in pose (X := E).
```

Application of Tautologies

`logic E`, where `E` is a fact, is equivalent to `assert H:E; [tauto | eapply H; clear H]. It is useful for instance to prove a conjunction [A ∧ B] by showing first [A] and then [A → B], through the command [logic (foral A B, A → (A → B) → A ∧ B)]`

```
Ltac logic_base E cont :=
  assert (H:E); [ cont tt | eapply H; clear H ].

Tactic Notation "logic" constr(E) :=
  logic_base E ltac:(fun _ ⇒ tauto).
```

Application Modulo Equalities

The tactic `equates` replaces a goal of the form `P x y z` with a goal of the form `P x ?a z` and a subgoal `?a = y`. The introduction of the evar `?a` makes it possible to apply lemmas that would not apply to the original goal, for example a lemma of the form `∀n m, P n n m`, because `x` and `y` might be equal but not convertible.

Usage is `equates` i_1 `... ik`, where the indices are the positions of the arguments to be replaced by evars, counting from the right-hand side. If `0` is given as argument, then the entire goal is replaced by an evar.

```
Section equatesLemma.
Variables (A₀ A₁ : Type).
```

```
Variables (A2 : ∀ (x1 : A1), Type).
Variables (A3 : ∀ (x1 : A1) (x2 : A2 x1), Type).
Variables (A4 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2), Type).
Variables (A5 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3),
Type).
Variables (A6 : ∀ (x1 : A1) (x2 : A2 x1) (x3 : A3 x2) (x4 : A4 x3)
(x5 : A5 x4), Type).

Lemma equates_0 : ∀ (P Q:Prop),
  P → P = Q → Q.
Proof. intros. subst. auto. Qed.

Lemma equates_1 :
  ∀ (P:A0→Prop) x1 y1,
  P y1 → x1 = y1 → P x1.
Proof. intros. subst. auto. Qed.

Lemma equates_2 :
  ∀ y1 (P:A0→∀(x1:A1),Prop) x1 x2,
  P y1 x2 → x1 = y1 → P x1 x2.
Proof. intros. subst. auto. Qed.

Lemma equates_3 :
  ∀ y1 (P:A0→∀(x1:A1)(x2:A2 x1),Prop) x1 x2 x3,
  P y1 x2 x3 → x1 = y1 → P x1 x2 x3.
Proof. intros. subst. auto. Qed.

Lemma equates_4 :
  ∀ y1 (P:A0→∀(x1:A1)(x2:A2 x1)(x3:A3 x2),Prop) x1 x2 x3 x4,
  P y1 x2 x3 x4 → x1 = y1 → P x1 x2 x3 x4.
Proof. intros. subst. auto. Qed.

Lemma equates_5 :
  ∀ y1 (P:A0→∀(x1:A1)(x2:A2 x1)(x3:A3 x2)(x4:A4 x3),Prop) x1 x2 x3 x4
x5,
  P y1 x2 x3 x4 x5 → x1 = y1 → P x1 x2 x3 x4 x5.
Proof. intros. subst. auto. Qed.

Lemma equates_6 :
  ∀ y1 (P:A0→∀(x1:A1)(x2:A2 x1)(x3:A3 x2)(x4:A4 x3)(x5:A5 x4),Prop)
  x1 x2 x3 x4 x5 x6,
  P y1 x2 x3 x4 x5 x6 → x1 = y1 → P x1 x2 x3 x4 x5 x6.
Proof. intros. subst. auto. Qed.

End equatesLemma.

Ltac equates_lemma n :=
  match nat_from_number n with
  | 0 ⇒ constr:(equates_0)
  | 1 ⇒ constr:(equates_1)
  | 2 ⇒ constr:(equates_2)
  | 3 ⇒ constr:(equates_3)
  | 4 ⇒ constr:(equates_4)
  | 5 ⇒ constr:(equates_5)
```

```
    | 6 ⇒ constr:(equates_6)
    end.

Ltac equates_one n :=
  let L := equates_lemma n in
  eapply L.

Ltac equates_several E cont :=
  let all_pos := match type of E with
    | List.list Boxer ⇒ constr:(E)
    | _ ⇒ constr:((boxer E)::nil)
    end in
  let rec go pos :=
     match pos with
     | nil ⇒ cont tt
     | (boxer ?n)::?pos' ⇒ equates_one n; [ instantiate; go pos' |
]
     end in
  go all_pos.

Tactic Notation "equates" constr(E) :=
  equates_several E ltac:(fun _ ⇒ idtac).
Tactic Notation "equates" constr(n1) constr(n2) :=
  equates (>> n1 n2).
Tactic Notation "equates" constr(n1) constr(n2) constr(n3) :=
  equates (>> n1 n2 n3).
Tactic Notation "equates" constr(n1) constr(n2) constr(n3)
constr(n4) :=
  equates (>> n1 n2 n3 n4).
```

`applys_eq H i1 .. iK` is the same as `equates i1 .. iK` followed by `apply H` on the first subgoal.

```
Tactic Notation "applys_eq" constr(H) constr(E) :=
  equates_several E ltac:(fun _ ⇒ sapply H).
Tactic Notation "applys_eq" constr(H) constr(n1) constr(n2) :=
  applys_eq H (>> n1 n2).
Tactic Notation "applys_eq" constr(H) constr(n1) constr(n2)
constr(n3) :=
  applys_eq H (>> n1 n2 n3).
Tactic Notation "applys_eq" constr(H) constr(n1) constr(n2)
constr(n3) constr(n4) :=
  applys_eq H (>> n1 n2 n3 n4).
```

Absurd Goals

`false_goal` replaces any goal by the goal `False`. Contrary to the tactic `false` (below), it does not try to do anything else

```
Tactic Notation "false_goal" :=
  elimtype False.
```

`false_post` is the underlying tactic used to prove goals of the form `False`. In the default implementation, it proves the goal if the context contains `False` or an hypothesis of the form `C x1 .. xN = D y1 .. yM`, or if the `congruence` tactic finds a proof of `x ≠ x` for some `x`.

```
Ltac false_post :=
  solve [ assumption | discriminate | congruence ].
```

`false` replaces any goal by the goal `False`, and calls `false_post`

```
Tactic Notation "false" :=
  false_goal; try false_post.
```

`tryfalse` tries to solve a goal by contradiction, and leaves the goal unchanged if it cannot solve it. It is equivalent to `try solve \[ false \]`.

```
Tactic Notation "tryfalse" :=
  try solve [ false ].
```

`false E` tries to exploit lemma `E` to prove the goal false. `false E1 .. EN` is equivalent to `false (>> E1 .. EN)`, which tries to apply `applys (>> E1 .. EN)` and if it does not work then tries `forwards H: (>> E1 .. EN)` followed with `false`

```
Ltac false_then E cont :=
  false_goal; first
  [ applys E; instantiate
  | forwards_then E ltac:(fun M ⇒
     pose M; jauto_set_hyps; intros; false) ];
  cont tt.
  (* TODO: is cont needed? *)

Tactic Notation "false" constr(E) :=
  false_then E ltac:(fun _ ⇒ idtac).
Tactic Notation "false" constr(E) constr(E1) :=
  false (>> E E1).
Tactic Notation "false" constr(E) constr(E1) constr(E2) :=
  false (>> E E1 E2).
Tactic Notation "false" constr(E) constr(E1) constr(E2) constr(E3)
:=
  false (>> E E1 E2 E3).
Tactic Notation "false" constr(E) constr(E1) constr(E2) constr(E3)
constr(E4) :=
  false (>> E E1 E2 E3 E4).
```

`false_invert H` proves a goal if it absurd after calling `inversion H` and `false`

```
Ltac false_invert_for H :=
  let M := fresh in pose (M := H); inversion H; false.

Tactic Notation "false_invert" constr(H) :=
  try solve [ false_invert_for H | false ].
```

`false_invert` proves any goal provided there is at least one hypothesis `H` in the context (or as a universally quantified hypothesis visible at the head of the goal) that can be proved absurd by calling `inversion H`.

```
Ltac false_invert_iter :=
  match goal with H:_ |- _ ⇒
    solve [ inversion H; false
          | clear H; false_invert_iter
          | fail 2 ] end.

Tactic Notation "false_invert" :=
  intros; solve [ false_invert_iter | false ].
```

`tryfalse_invert H` and `tryfalse_invert` are like the above but leave the goal unchanged if they don't solve it.

```
Tactic Notation "tryfalse_invert" constr(H) :=
  try (false_invert H).

Tactic Notation "tryfalse_invert" :=
  try false_invert.
```

`false_neq_self_hyp` proves any goal if the context contains an hypothesis of the form `E ≠ E`. It is a restricted and optimized version of `false`. It is intended to be used by other tactics only.

```
Ltac false_neq_self_hyp :=
  match goal with H: ?x ≠ ?x |- _ ⇒
    false_goal; apply H; reflexivity end.
```

Introduction and Generalization

Introduction

`introv` is used to name only non-dependent hypothesis.

- If `introv` is called on a goal of the form `∀x, H`, it should introduce all the variables quantified with a ∀at the head of the goal, but it does not introduce hypotheses that preceed an arrow constructor, like in `P → Q`.
- If `introv` is called on a goal that is not of the form `∀x, H` nor `P → Q`, the tactic unfolds definitions until the goal takes the form `∀x, H` or `P → Q`. If unfolding definitions does not produces a goal of this form, then the tactic `introv` does nothing at all.

```
(* introv_rec introduces all visible variables.
   It does not try to unfold any definition. *)

Ltac introv_rec :=
  match goal with
  | |- ?P → ?Q ⇒ idtac
  | |- ∀ _, _ ⇒ intro; introv_rec
  | |- _ ⇒ idtac
  end.

(* introv_noarg forces the goal to be a ∀or an →,
   and then calls introv_rec to introduces variables
```

```
   (possibly none, in which case introv is the same as hnf).
   If the goal is not a product, then it does not do anything. *)

Ltac introv_noarg :=
  match goal with
  | |- ?P → ?Q ⇒ idtac
  | |- ∀ _, _ ⇒ introv_rec
  | |- ?G ⇒ hnf;
     match goal with
     | |- ?P → ?Q ⇒ idtac
     | |- ∀ _, _ ⇒ introv_rec
     end
  | |- _ ⇒ idtac
  end.

  (* simpler yet perhaps less efficient imlementation *)
  Ltac introv_noarg_not_optimized :=
    intro; match goal with H:_|-_ ⇒ revert H end; introv_rec.

(* introv_arg H introduces one non-dependent hypothesis
   under the name H, after introducing the variables
   quantified with a ∀that preceeds this hypothesis.
   This tactic fails if there does not exist a hypothesis
   to be introduced. *)
  (* todo: __ in introv means "intros" *)

Ltac introv_arg H :=
  hnf; match goal with
  | |- ?P → ?Q ⇒ intros H
  | |- ∀ _, _ ⇒ intro; introv_arg H
  end.

(* introv I1 .. IN iterates introv Ik *)

Tactic Notation "introv" :=
  introv_noarg.
Tactic Notation "introv" simple_intropattern(I1) :=
  introv_arg I1.
Tactic Notation "introv" simple_intropattern(I1)
simple_intropattern(I2) :=
  introv I1; introv I2.
Tactic Notation "introv" simple_intropattern(I1)
simple_intropattern(I2)
 simple_intropattern(I3) :=
  introv I1; introv I2 I3.
Tactic Notation "introv" simple_intropattern(I1)
simple_intropattern(I2)
 simple_intropattern(I3) simple_intropattern(I4) :=
  introv I1; introv I2 I3 I4.
Tactic Notation "introv" simple_intropattern(I1)
simple_intropattern(I2)
 simple_intropattern(I3) simple_intropattern(I4)
simple_intropattern(I5) :=
  introv I1; introv I2 I3 I4 I5.
Tactic Notation "introv" simple_intropattern(I1)
```

```
  simple_intropattern(I2)
   simple_intropattern(I3) simple_intropattern(I4)
  simple_intropattern(I5)
   simple_intropattern(I6) :=
    introv I1; introv I2 I3 I4 I5 I6.
Tactic Notation "introv" simple_intropattern(I1)
  simple_intropattern(I2)
   simple_intropattern(I3) simple_intropattern(I4)
  simple_intropattern(I5)
   simple_intropattern(I6) simple_intropattern(I7) :=
    introv I1; introv I2 I3 I4 I5 I6 I7.
Tactic Notation "introv" simple_intropattern(I1)
  simple_intropattern(I2)
   simple_intropattern(I3) simple_intropattern(I4)
  simple_intropattern(I5)
   simple_intropattern(I6) simple_intropattern(I7)
  simple_intropattern(I8) :=
    introv I1; introv I2 I3 I4 I5 I6 I7 I8.
Tactic Notation "introv" simple_intropattern(I1)
  simple_intropattern(I2)
   simple_intropattern(I3) simple_intropattern(I4)
  simple_intropattern(I5)
   simple_intropattern(I6) simple_intropattern(I7)
  simple_intropattern(I8)
   simple_intropattern(I9) :=
    introv I1; introv I2 I3 I4 I5 I6 I7 I8 I9.
Tactic Notation "introv" simple_intropattern(I1)
  simple_intropattern(I2)
   simple_intropattern(I3) simple_intropattern(I4)
  simple_intropattern(I5)
   simple_intropattern(I6) simple_intropattern(I7)
  simple_intropattern(I8)
   simple_intropattern(I9) simple_intropattern(I10) :=
    introv I1; introv I2 I3 I4 I5 I6 I7 I8 I9 I10.
```

`intros_all` repeats `intro` as long as possible. Contrary to `intros`, it unfolds any definition on the way. Remark that it also unfolds the definition of negation, so applying `introz` to a goal of the form `∀x, P x → ¬Q` will introduce `x` and `P x` and `Q`, and will leave `False` in the goal.

```
Tactic Notation "intros_all" :=
  repeat intro.
```

`intros_hnf` introduces an hypothesis and sets in head normal form

```
Tactic Notation "intro_hnf" :=
  intro; match goal with H: _ |- _ ⇒ hnf in H end.
```

Generalization

`gen X1 .. XN` is a shorthand for calling `generalize dependent` successively on variables `XN...X1`. Note that the variables are generalized in reverse order, following the convention of the `generalize` tactic: it means that `X1` will be the first quantified variable in the resulting goal.

```
Tactic Notation "gen" ident(X1) :=
  generalize dependent X1.
Tactic Notation "gen" ident(X1) ident(X2) :=
  gen X2; gen X1.
Tactic Notation "gen" ident(X1) ident(X2) ident(X3) :=
  gen X3; gen X2; gen X1.
Tactic Notation "gen" ident(X1) ident(X2) ident(X3) ident(X4) :=
  gen X4; gen X3; gen X2; gen X1.
Tactic Notation "gen" ident(X1) ident(X2) ident(X3) ident(X4)
ident(X5) :=
  gen X5; gen X4; gen X3; gen X2; gen X1.
Tactic Notation "gen" ident(X1) ident(X2) ident(X3) ident(X4)
ident(X5)
 ident(X6) :=
  gen X6; gen X5; gen X4; gen X3; gen X2; gen X1.
Tactic Notation "gen" ident(X1) ident(X2) ident(X3) ident(X4)
ident(X5)
 ident(X6) ident(X7) :=
  gen X7; gen X6; gen X5; gen X4; gen X3; gen X2; gen X1.
Tactic Notation "gen" ident(X1) ident(X2) ident(X3) ident(X4)
ident(X5)
 ident(X6) ident(X7) ident(X8) :=
  gen X8; gen X7; gen X6; gen X5; gen X4; gen X3; gen X2; gen X1.
Tactic Notation "gen" ident(X1) ident(X2) ident(X3) ident(X4)
ident(X5)
 ident(X6) ident(X7) ident(X8) ident(X9) :=
  gen X9; gen X8; gen X7; gen X6; gen X5; gen X4; gen X3; gen X2;
gen X1.
Tactic Notation "gen" ident(X1) ident(X2) ident(X3) ident(X4)
ident(X5)
 ident(X6) ident(X7) ident(X8) ident(X9) ident(X10) :=
  gen X10; gen X9; gen X8; gen X7; gen X6; gen X5; gen X4; gen X3;
gen X2; gen X1.
```

`generalizes X` is a shorthand for calling `generalize X; clear X`. It is weaker than tactic `gen X` since it does not support dependencies. It is mainly intended for writing tactics.

```
Tactic Notation "generalizes" hyp(X) :=
  generalize X; clear X.
Tactic Notation "generalizes" hyp(X1) hyp(X2) :=
  generalizes X1; generalizes X2.
Tactic Notation "generalizes" hyp(X1) hyp(X2) hyp(X3) :=
```

```
    generalizes X1 X2; generalizes X3.
  Tactic Notation "generalizes" hyp(X1) hyp(X2) hyp(X3) hyp(X4) :=
    generalizes X1 X2 X3; generalizes X4.
```

Naming

`sets X: E` is the same as `set (X := E) in *`, that is, it replaces all occurences of `E` by a fresh meta-variable `X` whose definition is `E`.

```
Tactic Notation "sets" ident(X) ":" constr(E) :=
  set (X := E) in *.
```

`def_to_eq E X H` applies when `X := E` is a local definition. It adds an assumption `H: X = E` and then clears the definition of `X`. `def_to_eq_sym` is similar except that it generates the equality `H: E = X`.

```
Ltac def_to_eq X HX E :=
  assert (HX : X = E) by reflexivity; clearbody X.
Ltac def_to_eq_sym X HX E :=
  assert (HX : E = X) by reflexivity; clearbody X.
```

`set_eq X H: E` generates the equality `H: X = E`, for a fresh name `X`, and replaces `E` by `X` in the current goal. Syntaxes `set_eq X: E` and `set_eq: E` are also available. Similarly, `set_eq <- X H: E` generates the equality `H: E = X`.

`sets_eq X HX: E` does the same but replaces `E` by `X` everywhere in the goal. `sets_eq X HX: E in H` replaces in `H`. `set_eq X HX: E in |-` performs no substitution at all.

```
Tactic Notation "set_eq" ident(X) ident(HX) ":" constr(E) :=
  set (X := E); def_to_eq X HX E.
Tactic Notation "set_eq" ident(X) ":" constr(E) :=
  let HX := fresh "EQ" X in set_eq X HX: E.
Tactic Notation "set_eq" ":" constr(E) :=
  let X := fresh "X" in set_eq X: E.

Tactic Notation "set_eq" "<-" ident(X) ident(HX) ":" constr(E) :=
  set (X := E); def_to_eq_sym X HX E.
Tactic Notation "set_eq" "<-" ident(X) ":" constr(E) :=
  let HX := fresh "EQ" X in set_eq <- X HX: E.
Tactic Notation "set_eq" "<-" ":" constr(E) :=
  let X := fresh "X" in set_eq <- X: E.

Tactic Notation "sets_eq" ident(X) ident(HX) ":" constr(E) :=
  set (X := E) in *; def_to_eq X HX E.
Tactic Notation "sets_eq" ident(X) ":" constr(E) :=
  let HX := fresh "EQ" X in sets_eq X HX: E.
Tactic Notation "sets_eq" ":" constr(E) :=
  let X := fresh "X" in sets_eq X: E.

Tactic Notation "sets_eq" "<-" ident(X) ident(HX) ":" constr(E) :=
  set (X := E) in *; def_to_eq_sym X HX E.
Tactic Notation "sets_eq" "<-" ident(X) ":" constr(E) :=
  let HX := fresh "EQ" X in sets_eq <- X HX: E.
Tactic Notation "sets_eq" "<-" ":" constr(E) :=
  let X := fresh "X" in sets_eq <- X: E.
```

```
Tactic Notation "set_eq" ident(X) ident(HX) ":" constr(E) "in"
hyp(H) :=
  set (X := E) in H; def_to_eq X HX E.
Tactic Notation "set_eq" ident(X) ":" constr(E) "in" hyp(H) :=
  let HX := fresh "EQ" X in set_eq X HX: E in H.
Tactic Notation "set_eq" ":" constr(E) "in" hyp(H) :=
  let X := fresh "X" in set_eq X: E in H.

Tactic Notation "set_eq" "<-" ident(X) ident(HX) ":" constr(E) "in"
hyp(H) :=
  set (X := E) in H; def_to_eq_sym X HX E.
Tactic Notation "set_eq" "<-" ident(X) ":" constr(E) "in" hyp(H) :=
  let HX := fresh "EQ" X in set_eq <- X HX: E in H.
Tactic Notation "set_eq" "<-" ":" constr(E) "in" hyp(H) :=
  let X := fresh "X" in set_eq <- X: E in H.

Tactic Notation "set_eq" ident(X) ident(HX) ":" constr(E) "in" "|-"
:=
  set (X := E) in |-; def_to_eq X HX E.
Tactic Notation "set_eq" ident(X) ":" constr(E) "in" "|-" :=
  let HX := fresh "EQ" X in set_eq X HX: E in |-.
Tactic Notation "set_eq" ":" constr(E) "in" "|-" :=
  let X := fresh "X" in set_eq X: E in |-.

Tactic Notation "set_eq" "<-" ident(X) ident(HX) ":" constr(E) "in"
"|-" :=
  set (X := E) in |-; def_to_eq_sym X HX E.
Tactic Notation "set_eq" "<-" ident(X) ":" constr(E) "in" "|-" :=
  let HX := fresh "EQ" X in set_eq <- X HX: E in |-.
Tactic Notation "set_eq" "<-" ":" constr(E) "in" "|-" :=
  let X := fresh "X" in set_eq <- X: E in |-.
```

`gen_eq X: E` is a tactic whose purpose is to introduce equalities so as to work around the limitation of the `induction` tactic which typically loses information. `gen_eq E as X` replaces all occurences of term `E` with a fresh variable `X` and the equality `X = E` as extra hypothesis to the current conclusion. In other words a conclusion `C` will be turned into `(X = E) → C`. `gen_eq: E` and `gen_eq: E as X` are also accepted.

```
Tactic Notation "gen_eq" ident(X) ":" constr(E) :=
  let EQ := fresh in sets_eq X EQ: E; revert EQ.
Tactic Notation "gen_eq" ":" constr(E) :=
  let X := fresh "X" in gen_eq X: E.
Tactic Notation "gen_eq" ":" constr(E) "as" ident(X) :=
  gen_eq X: E.
Tactic Notation "gen_eq" ident(X1) ":" constr(E1) ","
  ident(X2) ":" constr(E2) :=
  gen_eq X2: E2; gen_eq X1: E1.
Tactic Notation "gen_eq" ident(X1) ":" constr(E1) ","
  ident(X2) ":" constr(E2) "," ident(X3) ":" constr(E3) :=
  gen_eq X3: E3; gen_eq X2: E2; gen_eq X1: E1.
```

`sets_let X` finds the first let-expression in the goal and names its body `X`. `sets_eq_let X` is similar, except that it generates an explicit equality. Tactics `sets_let X in H` and `sets_eq_let X in H` allow specifying a particular hypothesis (by default, the first one that contains a `let` is considered).

Known limitation: it does not seem possible to support naming of multiple let-in constructs inside a term, from ltac.

```
Ltac sets_let_base tac :=
  match goal with
  | |- context[let _ := ?E in _] ⇒ tac E; cbv zeta
  | H: context[let _ := ?E in _] |- _ ⇒ tac E; cbv zeta in H
  end.

Ltac sets_let_in_base H tac :=
  match type of H with context[let _ := ?E in _] ⇒
    tac E; cbv zeta in H end.

Tactic Notation "sets_let" ident(X) :=
  sets_let_base ltac:(fun E ⇒ sets X: E).
Tactic Notation "sets_let" ident(X) "in" hyp(H) :=
  sets_let_in_base H ltac:(fun E ⇒ sets X: E).
Tactic Notation "sets_eq_let" ident(X) :=
  sets_let_base ltac:(fun E ⇒ sets_eq X: E).
Tactic Notation "sets_eq_let" ident(X) "in" hyp(H) :=
  sets_let_in_base H ltac:(fun E ⇒ sets_eq X: E).
```

Rewriting

`rewrites E` is similar to `rewrite` except that it supports the `rm` directives to clear hypotheses on the fly, and that it supports a list of arguments in the form `rewrites (>> E1 E2 E3)` to indicate that `forwards` should be invoked first before `rewrites` is called.

```
Ltac rewrites_base E cont :=
  match type of E with
  | List.list Boxer ⇒ forwards_then E cont
  | _ ⇒ cont E; fast_rm_inside E
  end.

Tactic Notation "rewrites" constr(E) :=
  rewrites_base E ltac:(fun M ⇒ rewrite M ).
Tactic Notation "rewrites" constr(E) "in" hyp(H) :=
  rewrites_base E ltac:(fun M ⇒ rewrite M in H).
Tactic Notation "rewrites" constr(E) "in" "*" :=
  rewrites_base E ltac:(fun M ⇒ rewrite M in *).
Tactic Notation "rewrites" "<-" constr(E) :=
  rewrites_base E ltac:(fun M ⇒ rewrite <- M ).
Tactic Notation "rewrites" "<-" constr(E) "in" hyp(H) :=
  rewrites_base E ltac:(fun M ⇒ rewrite <- M in H).
Tactic Notation "rewrites" "<-" constr(E) "in" "*" :=
  rewrites_base E ltac:(fun M ⇒ rewrite <- M in *).

(* TODO: extend tactics below to use rewrites *)
```

`rewrite_all E` iterates version of `rewrite E` as long as possible. Warning: this tactic can easily get into an infinite loop. Syntax for rewriting from right to left and/or into an hypothese is similar to the one of `rewrite`.

```
Tactic Notation "rewrite_all" constr(E) :=
  repeat rewrite E.
Tactic Notation "rewrite_all" "<-" constr(E) :=
```

```
    repeat rewrite <- E.
  Tactic Notation "rewrite_all" constr(E) "in" ident(H) :=
    repeat rewrite E in H.
  Tactic Notation "rewrite_all" "<-" constr(E) "in" ident(H) :=
    repeat rewrite <- E in H.
  Tactic Notation "rewrite_all" constr(E) "in" "*" :=
    repeat rewrite E in *.
  Tactic Notation "rewrite_all" "<-" constr(E) "in" "*" :=
    repeat rewrite <- E in *.
```

`asserts_rewrite E` asserts that an equality `E` holds (generating a corresponding subgoal) and rewrite it straight away in the current goal. It avoids giving a name to the equality and later clearing it. Syntax for rewriting from right to left and/or into an hypothese is similar to the one of `rewrite`. Note: the tactic `replaces` plays a similar role.

```
Ltac asserts_rewrite_tactic E action :=
  let EQ := fresh in (assert (EQ : E);
  [ idtac | action EQ; clear EQ ]).

Tactic Notation "asserts_rewrite" constr(E) :=
  asserts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite EQ).
Tactic Notation "asserts_rewrite" "<-" constr(E) :=
  asserts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite <- EQ).
Tactic Notation "asserts_rewrite" constr(E) "in" hyp(H) :=
  asserts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite EQ in H).
Tactic Notation "asserts_rewrite" "<-" constr(E) "in" hyp(H) :=
  asserts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite <- EQ in H).
Tactic Notation "asserts_rewrite" constr(E) "in" "*" :=
  asserts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite EQ in *).
Tactic Notation "asserts_rewrite" "<-" constr(E) "in" "*" :=
  asserts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite <- EQ in *).
```

`cuts_rewrite E` is the same as `asserts_rewrite E` except that subgoals are permuted.

```
Ltac cuts_rewrite_tactic E action :=
  let EQ := fresh in (cuts EQ: E;
  [ action EQ; clear EQ | idtac ]).

Tactic Notation "cuts_rewrite" constr(E) :=
  cuts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite EQ).
Tactic Notation "cuts_rewrite" "<-" constr(E) :=
  cuts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite <- EQ).
Tactic Notation "cuts_rewrite" constr(E) "in" hyp(H) :=
  cuts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite EQ in H).
Tactic Notation "cuts_rewrite" "<-" constr(E) "in" hyp(H) :=
  cuts_rewrite_tactic E ltac:(fun EQ ⇒ rewrite <- EQ in H).
```

`rewrite_except H EQ` rewrites equality `EQ` everywhere but in hypothesis `H`. Mainly useful for other tactics.

```
Ltac rewrite_except H EQ :=
  let K := fresh in let T := type of H in
  set (K := T) in H;
  rewrite EQ in *; unfold K in H; clear K.
```

`rewrites E at K` applies when `E` is of the form $T_1 = T_2$ rewrites the equality `E` at the `K`-th occurence of T_1 in the current goal. Syntaxes `rewrites <- E at K` and `rewrites E at K in H` are also available.

```
Tactic Notation "rewrites" constr(E) "at" constr(K) :=
  match type of E with ?T1 = ?T2 ⇒
    ltac_action_at K of T1 do (rewrites E) end.
Tactic Notation "rewrites" "<-" constr(E) "at" constr(K) :=
  match type of E with ?T1 = ?T2 ⇒
    ltac_action_at K of T2 do (rewrites <- E) end.
Tactic Notation "rewrites" constr(E) "at" constr(K) "in" hyp(H) :=
  match type of E with ?T1 = ?T2 ⇒
    ltac_action_at K of T1 in H do (rewrites E in H) end.
Tactic Notation "rewrites" "<-" constr(E) "at" constr(K) "in"
hyp(H) :=
  match type of E with ?T1 = ?T2 ⇒
    ltac_action_at K of T2 in H do (rewrites <- E in H) end.
```

Replace

`replaces E with F` is the same as `replace E with F` except that the equality `E = F` is generated as first subgoal. Syntax `replaces E with F in H` is also available. Note that contrary to `replace`, `replaces` does not try to solve the equality by `assumption`. Note: `replaces E with F` is similar to `asserts_rewrite (E = F)`.

```
Tactic Notation "replaces" constr(E) "with" constr(F) :=
  let T := fresh in assert (T: E = F); [ | replace E with F; clear
T ].

Tactic Notation "replaces" constr(E) "with" constr(F) "in" hyp(H)
:=
  let T := fresh in assert (T: E = F); [ | replace E with F in H;
clear T ].
```

`replaces E at K with F` replaces the `K`-th occurence of `E` with `F` in the current goal. Syntax `replaces E at K with F in H` is also available.

```
Tactic Notation "replaces" constr(E) "at" constr(K) "with"
constr(F) :=
  let T := fresh in assert (T: E = F); [ | rewrites T at K; clear T
].

Tactic Notation "replaces" constr(E) "at" constr(K) "with"
constr(F) "in" hyp(H) :=
  let T := fresh in assert (T: E = F); [ | rewrites T at K in H;
clear T ].
```

Change

`changes` is like `change` except that it does not silently fail to perform its task. (Note that, `changes` is implemented using `rewrite`, meaning that it might perform additional beta-reductions compared with the original `change` tactic.

```
(* TODO: support "changes (E1 = E2)" *)

Tactic Notation "changes" constr(E1) "with" constr(E2) "in" hyp(H)
:=
  asserts_rewrite (E1 = E2) in H; [ reflexivity | ].

Tactic Notation "changes" constr(E1) "with" constr(E2) :=
  asserts_rewrite (E1 = E2); [ reflexivity | ].

Tactic Notation "changes" constr(E1) "with" constr(E2) "in" "*" :=
  asserts_rewrite (E1 = E2) in *; [ reflexivity | ].
```

Renaming

`renames X1 to Y1, ..., XN to YN` is a shorthand for a sequence of renaming operations `rename Xi into Yi`.

```
Tactic Notation "renames" ident(X1) "to" ident(Y1) :=
  rename X1 into Y1.
Tactic Notation "renames" ident(X1) "to" ident(Y1) ","
 ident(X2) "to" ident(Y2) :=
  renames X1 to Y1; renames X2 to Y2.
Tactic Notation "renames" ident(X1) "to" ident(Y1) ","
 ident(X2) "to" ident(Y2) "," ident(X3) "to" ident(Y3) :=
  renames X1 to Y1; renames X2 to Y2, X3 to Y3.
Tactic Notation "renames" ident(X1) "to" ident(Y1) ","
 ident(X2) "to" ident(Y2) "," ident(X3) "to" ident(Y3) ","
 ident(X4) "to" ident(Y4) :=
  renames X1 to Y1; renames X2 to Y2, X3 to Y3, X4 to Y4.
Tactic Notation "renames" ident(X1) "to" ident(Y1) ","
 ident(X2) "to" ident(Y2) "," ident(X3) "to" ident(Y3) ","
 ident(X4) "to" ident(Y4) "," ident(X5) "to" ident(Y5) :=
  renames X1 to Y1; renames X2 to Y2, X3 to Y3, X4 to Y4, X5 to Y5.
Tactic Notation "renames" ident(X1) "to" ident(Y1) ","
 ident(X2) "to" ident(Y2) "," ident(X3) "to" ident(Y3) ","
 ident(X4) "to" ident(Y4) "," ident(X5) "to" ident(Y5) ","
 ident(X6) "to" ident(Y6) :=
  renames X1 to Y1; renames X2 to Y2, X3 to Y3, X4 to Y4, X5 to Y5,
X6 to Y6.
```

Unfolding

`unfolds` unfolds the head definition in the goal, i.e., if the goal has form `P x1 ... xN` then it calls `unfold P`. If the goal is an equality, it tries to unfold the head constant on the left-hand side, and otherwise tries on the right-hand side. If the goal is a product, it calls `intros` first. warning: this tactic is overriden in LibReflect.

```
Ltac apply_to_head_of E cont :=
  let go E :=
    let P := get_head E in cont P in
  match E with
```

```
    | ∀ _,_ ⇒ intros; apply_to_head_of E cont
    | ?A = ?B ⇒ first [ go A | go B ]
    | ?A ⇒ go A
    end.

Ltac unfolds_base :=
  match goal with |- ?G ⇒
   apply_to_head_of G ltac:(fun P ⇒ unfold P) end.

Tactic Notation "unfolds" :=
  unfolds_base.
```

`unfolds in H` unfolds the head definition of hypothesis `H`, i.e., if `H` has type `P x1 ... xN` then it calls `unfold P in H`.

```
Ltac unfolds_in_base H :=
  match type of H with ?G ⇒
   apply_to_head_of G ltac:(fun P ⇒ unfold P in H) end.

Tactic Notation "unfolds" "in" hyp(H) :=
  unfolds_in_base H.
```

`unfolds in H1,H2,..,HN` allows unfolding the head constant in several hypotheses at once.

```
Tactic Notation "unfolds" "in" hyp(H1) hyp(H2) :=
  unfolds in H1; unfolds in H2.
Tactic Notation "unfolds" "in" hyp(H1) hyp(H2) hyp(H3) :=
  unfolds in H1; unfolds in H2 H3.
Tactic Notation "unfolds" "in" hyp(H1) hyp(H2) hyp(H3) hyp(H4) :=
  unfolds in H1; unfolds in H2 H3 H4.
```

`unfolds P1,..,PN` is a shortcut for `unfold P1,..,PN in *`.

```
Tactic Notation "unfolds" constr(F1) :=
  unfold F1 in *.
Tactic Notation "unfolds" constr(F1) "," constr(F2) :=
  unfold F1,F2 in *.
Tactic Notation "unfolds" constr(F1) "," constr(F2)
 "," constr(F3) :=
  unfold F1,F2,F3 in *.
Tactic Notation "unfolds" constr(F1) "," constr(F2)
 "," constr(F3) "," constr(F4) :=
  unfold F1,F2,F3,F4 in *.
Tactic Notation "unfolds" constr(F1) "," constr(F2)
 "," constr(F3) "," constr(F4) "," constr(F5) :=
  unfold F1,F2,F3,F4,F5 in *.
Tactic Notation "unfolds" constr(F1) "," constr(F2)
 "," constr(F3) "," constr(F4) "," constr(F5) "," constr(F6) :=
  unfold F1,F2,F3,F4,F5,F6 in *.
Tactic Notation "unfolds" constr(F1) "," constr(F2)
 "," constr(F3) "," constr(F4) "," constr(F5)
```

```
  "," constr(F6) "," constr(F7) :=
   unfold F1,F2,F3,F4,F5,F6,F7 in *.
Tactic Notation "unfolds" constr(F1) "," constr(F2)
  "," constr(F3) "," constr(F4) "," constr(F5)
  "," constr(F6) "," constr(F7) "," constr(F8) :=
   unfold F1,F2,F3,F4,F5,F6,F7,F8 in *.
```

`folds P1,..,PN` is a shortcut for `fold P1 in *; ..; fold PN in *`.

```
Tactic Notation "folds" constr(H) :=
  fold H in *.
Tactic Notation "folds" constr(H1) "," constr(H2) :=
  folds H1; folds H2.
Tactic Notation "folds" constr(H1) "," constr(H2) "," constr(H3) :=
  folds H1; folds H2; folds H3.
Tactic Notation "folds" constr(H1) "," constr(H2) "," constr(H3)
 "," constr(H4) :=
  folds H1; folds H2; folds H3; folds H4.
Tactic Notation "folds" constr(H1) "," constr(H2) "," constr(H3)
 "," constr(H4) "," constr(H5) :=
  folds H1; folds H2; folds H3; folds H4; folds H5.
```

Simplification

`simpls` is a shortcut for `simpl in *`.

```
Tactic Notation "simpls" :=
  simpl in *.
```

`simpls P1,..,PN` is a shortcut for `simpl P1 in *; ..; simpl PN in *`.

```
Tactic Notation "simpls" constr(F1) :=
  simpl F1 in *.
Tactic Notation "simpls" constr(F1) "," constr(F2) :=
  simpls F1; simpls F2.
Tactic Notation "simpls" constr(F1) "," constr(F2)
 "," constr(F3) :=
  simpls F1; simpls F2; simpls F3.
Tactic Notation "simpls" constr(F1) "," constr(F2)
 "," constr(F3) "," constr(F4) :=
  simpls F1; simpls F2; simpls F3; simpls F4.
```

`unsimpl E` replaces all occurence of `X` by `E`, where `X` is the result which the tactic `simpl` would give when applied to `E`. It is useful to undo what `simpl` has simplified too far.

```
Tactic Notation "unsimpl" constr(E) :=
  let F := (eval simpl in E) in change F with E.
```

`unsimpl E in H` is similar to `unsimpl E` but it applies inside a particular hypothesis `H`.

```
Tactic Notation "unsimpl" constr(E) "in" hyp(H) :=
  let F := (eval simpl in E) in change F with E in H.
```

`unsimpl E in *` applies `unsimpl E` everywhere possible. `unsimpls E` is a synonymous.

```
Tactic Notation "unsimpl" constr(E) "in" "*" :=
  let F := (eval simpl in E) in change F with E in *.
Tactic Notation "unsimpls" constr(E) :=
  unsimpl E in *.
```

`nosimpl t` protects the Coq term`t` against some forms of simplification. See Gonthier's work for details on this trick.

```
Notation "'nosimpl' t" := (match tt with tt ⇒ t end)
  (at level 10).
```

Reduction

```
Tactic Notation "hnfs" := hnf in *.
```

Substitution

`substs` does the same as `subst`, except that it does not fail when there are circular equalities in the context.

```
Tactic Notation "substs" :=
  repeat (match goal with H: ?x = ?y |- _ ⇒
            first [ subst x | subst y ] end).
```

Implementation of `substs below`, which allows to call `subst` on all the hypotheses that lie beyond a given position in the proof context.

```
Ltac substs_below limit :=
  match goal with H: ?T |- _ ⇒
  match T with
  | limit ⇒ idtac
  | ?x = ?y ⇒
    first [ subst x; substs_below limit
          | subst y; substs_below limit
          | generalizes H; substs_below limit; intro ]
  end end.
```

`substs below body E` applies `subst` on all equalities that appear in the context below the first hypothesis whose body is `E`. If there is no such hypothesis in the context, it is equivalent to `subst`. For instance, if `H` is an hypothesis, then `substs below H` will substitute equalities below hypothesis `H`.

```
Tactic Notation "substs" "below" "body" constr(M) :=
  substs_below M.
```

`substs below H` applies `subst` on all equalities that appear in the context below the hypothesis named `H`. Note that the current implementation is technically incorrect since it will confuse different hypotheses with the same body.

```
Tactic Notation "substs" "below" hyp(H) :=
  match type of H with ?M ⇒ substs below body M end.
```

`subst_hyp H` substitutes the equality contained in the first hypothesis from the context.

```
Ltac intro_subst_hyp := fail. (* definition further on *)
```

subst_hyp H substitutes the equality contained in H.

```
Ltac subst_hyp_base H :=
  match type of H with
  | (_,_,_,_,_) = (_,_,_,_,_) ⇒ injection H; clear H; do 4
intro_subst_hyp
  | (_,_,_,_) = (_,_,_,_) ⇒ injection H; clear H; do 4
intro_subst_hyp
  | (_,_,_) = (_,_,_) ⇒ injection H; clear H; do 3 intro_subst_hyp
  | (_,_) = (_,_) ⇒ injection H; clear H; do 2 intro_subst_hyp
  | ?x = ?x ⇒ clear H
  | ?x = ?y ⇒ first [ subst x | subst y ]
  end.

Tactic Notation "subst_hyp" hyp(H) := subst_hyp_base H.

Ltac intro_subst_hyp ::=
  let H := fresh "TEMP" in intros H; subst_hyp H.
```

intro_subst is a shorthand for intro H; subst_hyp H: it introduces and substitutes the equality at the head of the current goal.

```
Tactic Notation "intro_subst" :=
  let H := fresh "TEMP" in intros H; subst_hyp H.
```

subst_local substitutes all local definition from the context

```
Ltac subst_local :=
  repeat match goal with H:=_ |- _ ⇒ subst H end.
```

subst_eq E takes an equality x = t and replace x with t everywhere in the goal

```
Ltac subst_eq_base E :=
  let H := fresh "TEMP" in lets H: E; subst_hyp H.

Tactic Notation "subst_eq" constr(E) :=
  subst_eq_base E.
```

Tactics to Work with Proof Irrelevance

```
Require Import ProofIrrelevance.
```

pi_rewrite E replaces E of type Prop with a fresh unification variable, and is thus a practical way to exploit proof irrelevance, without writing explicitly rewrite (proof_irrelevance E E'). Particularly useful when E' is a big expression.

```
Ltac pi_rewrite_base E rewrite_tac :=
  let E' := fresh in let T := type of E in evar (E':T);
  rewrite_tac (@proof_irrelevance _ E E'); subst E'.

Tactic Notation "pi_rewrite" constr(E) :=
  pi_rewrite_base E ltac:(fun X ⇒ rewrite X).
Tactic Notation "pi_rewrite" constr(E) "in" hyp(H) :=
  pi_rewrite_base E ltac:(fun X ⇒ rewrite X in H).
```

Proving Equalities

Note: current implementation only supports up to arity 5

`fequal` is a variation on `f_equal` which has a better behaviour on equalities between n-ary tuples.

```
Ltac fequal_base :=
  let go := f_equal; [ fequal_base | ] in
  match goal with
  | |- (_,_,_) = (_,_,_) ⇒ go
  | |- (_,_,_,_) = (_,_,_,_) ⇒ go
  | |- (_,_,_,_,_) = (_,_,_,_,_) ⇒ go
  | |- (_,_,_,_,_,_) = (_,_,_,_,_,_) ⇒ go
  | |- _ ⇒ f_equal
  end.

Tactic Notation "fequal" :=
  fequal_base.
```

`fequals` is the same as `fequal` except that it tries and solve all trivial subgoals, using `reflexivity` and `congruence` (as well as the proof-irrelevance principle). `fequals` applies to goals of the form `f` x_1 `..` `xN` = `f` y_1 `..` `yN` and produces some subgoals of the form `xi` = `yi`).

```
Ltac fequal_post :=
  first [ reflexivity | congruence | apply proof_irrelevance |
idtac ].

Tactic Notation "fequals" :=
  fequal; fequal_post.
```

`fequals_rec` calls `fequals` recursively. It is equivalent to `repeat (progress fequals)`.

```
Tactic Notation "fequals_rec" :=
  repeat (progress fequals).
```

Inversion

Basic Inversion

`invert keep H` is same to `inversion H` except that it puts all the facts obtained in the goal. The keyword `keep` means that the hypothesis `H` should not be removed.

```
Tactic Notation "invert" "keep" hyp(H) :=
  pose ltac_mark; inversion H; gen_until_mark.
```

`invert keep H as` X_1 `..` `XN` is the same as `inversion H as ...` except that only hypotheses which are not variable need to be named explicitely, in a similar fashion as `introv` is used to name only hypotheses.

```
Tactic Notation "invert" "keep" hyp(H) "as" simple_intropattern(I1)
:=
```

```
    invert keep H; introv I₁.
  Tactic Notation "invert" "keep" hyp(H) "as" simple_intropattern(I₁)
   simple_intropattern(I₂) :=
    invert keep H; introv I₁ I₂.
  Tactic Notation "invert" "keep" hyp(H) "as" simple_intropattern(I₁)
   simple_intropattern(I₂) simple_intropattern(I₃) :=
    invert keep H; introv I₁ I₂ I₃.
```

`invert H` is same to `inversion H` except that it puts all the facts obtained in the goal and clears hypothesis `H`. In other words, it is equivalent to `invert keep H; clear H`.

```
Tactic Notation "invert" hyp(H) :=
  invert keep H; clear H.
```

`invert H as X₁ .. XN` is the same as `invert keep H as X₁ .. XN` but it also clears hypothesis `H`.

```
Tactic Notation "invert_tactic" hyp(H) tactic(tac) :=
  let H' := fresh in rename H into H'; tac H'; clear H'.
Tactic Notation "invert" hyp(H) "as" simple_intropattern(I₁) :=
  invert_tactic H (fun H ⇒ invert keep H as I₁).
Tactic Notation "invert" hyp(H) "as" simple_intropattern(I₁)
 simple_intropattern(I₂) :=
  invert_tactic H (fun H ⇒ invert keep H as I₁ I₂).
Tactic Notation "invert" hyp(H) "as" simple_intropattern(I₁)
 simple_intropattern(I₂) simple_intropattern(I₃) :=
  invert_tactic H (fun H ⇒ invert keep H as I₁ I₂ I₃).
```

Inversion with Substitution

Our inversion tactics is able to get rid of dependent equalities generated by `inversion`, using proof irrelevance.

```
(* --
we do not import Eqdep because it imports nasty hints automatically
    Require Import Eqdep. *)

Axiom inj_pair2 :
(* is in fact derivable from the axioms in LibAxiom.v *)
  ∀ (U : Type) (P : U → Type) (p : U) (x y : P p),
      existT P p x = existT P p y → x = y.
(* Proof using. apply Eqdep.EqdepTheory.inj_pair2. Qed.*)

Ltac inverts_tactic H i₁ i₂ i₃ i₄ i₅ i₆ :=
  let rec go i₁ i₂ i₃ i₄ i₅ i₆ :=
    match goal with
    | |- (ltac_Mark → _) ⇒ intros _
    | |- (?x = ?y → _) ⇒ let H := fresh in intro H;
                           first [ subst x | subst y ];
                           go i₁ i₂ i₃ i₄ i₅ i₆
    | |- (existT ?P ?p ?x = existT ?P ?p ?y → _) ⇒
         let H := fresh in intro H;
         generalize (@inj_pair2 _ P p x y H);
         clear H; go i₁ i₂ i₃ i₄ i₅ i₆
```

```
      | |- (?P → ?Q) ⇒ i1; go i2 i3 i4 i5 i6 ltac:(intro)
      | |- (∀ _, _) ⇒ intro; go i1 i2 i3 i4 i5 i6
      end in
    generalize ltac_mark; invert keep H; go i1 i2 i3 i4 i5 i6;
    unfold eq' in *.
```

inverts keep H is same to invert keep H except that it applies subst to all the equalities generated by the inversion.

```
Tactic Notation "inverts" "keep" hyp(H) :=
  inverts_tactic H ltac:(intro) ltac:(intro) ltac:(intro)
                   ltac:(intro) ltac:(intro) ltac:(intro).
```

inverts keep H as X_1 .. XN is the same as invert keep H as X_1 .. XN except that it applies subst to all the equalities generated by the inversion

```
Tactic Notation "inverts" "keep" hyp(H) "as"
simple_intropattern(I1) :=
  inverts_tactic H ltac:(intros I1)
   ltac:(intro) ltac:(intro) ltac:(intro) ltac:(intro) ltac:
(intro).
Tactic Notation "inverts" "keep" hyp(H) "as"
simple_intropattern(I1)
 simple_intropattern(I2) :=
  inverts_tactic H ltac:(intros I1) ltac:(intros I2)
   ltac:(intro) ltac:(intro) ltac:(intro) ltac:(intro).
Tactic Notation "inverts" "keep" hyp(H) "as"
simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3) :=
  inverts_tactic H ltac:(intros I1) ltac:(intros I2) ltac:(intros
I3)
   ltac:(intro) ltac:(intro) ltac:(intro).
Tactic Notation "inverts" "keep" hyp(H) "as"
simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3)
simple_intropattern(I4) :=
  inverts_tactic H ltac:(intros I1) ltac:(intros I2) ltac:(intros
I3)
   ltac:(intros I4) ltac:(intro) ltac:(intro).
Tactic Notation "inverts" "keep" hyp(H) "as"
simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3)
simple_intropattern(I4)
 simple_intropattern(I5) :=
  inverts_tactic H ltac:(intros I1) ltac:(intros I2) ltac:(intros
I3)
   ltac:(intros I4) ltac:(intros I5) ltac:(intro).
Tactic Notation "inverts" "keep" hyp(H) "as"
simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3)
simple_intropattern(I4)
```

```
 simple_intropattern(I5) simple_intropattern(I6) :=
  inverts_tactic H ltac:(intros I1) ltac:(intros I2) ltac:(intros
I3)
   ltac:(intros I4) ltac:(intros I5) ltac:(intros I6).
```

`inverts H` is same to `inverts keep H` except that it clears hypothesis `H`.

```
Tactic Notation "inverts" hyp(H) :=
  inverts keep H; clear H.
```

`inverts H as X1 .. XN` is the same as `inverts keep H as X1 .. XN` but it also clears the hypothesis `H`.

```
Tactic Notation "inverts_tactic" hyp(H) tactic(tac) :=
  let H' := fresh in rename H into H'; tac H'; clear H'.
Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1) :=
  invert_tactic H (fun H ⇒ inverts keep H as I1).
Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
 simple_intropattern(I2) :=
  invert_tactic H (fun H ⇒ inverts keep H as I1 I2).
Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3) :=
  invert_tactic H (fun H ⇒ inverts keep H as I1 I2 I3).
Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3)
simple_intropattern(I4) :=
  invert_tactic H (fun H ⇒ inverts keep H as I1 I2 I3 I4).
Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3)
simple_intropattern(I4)
 simple_intropattern(I5) :=
  invert_tactic H (fun H ⇒ inverts keep H as I1 I2 I3 I4 I5).
Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3)
simple_intropattern(I4)
 simple_intropattern(I5) simple_intropattern(I6) :=
  invert_tactic H (fun H ⇒ inverts keep H as I1 I2 I3 I4 I5 I6).
```

`inverts H as` performs an inversion on hypothesis `H`, substitutes generated equalities, and put in the goal the other freshly-created hypotheses, for the user to name explicitly. `inverts keep H as` is the same except that it does not clear `H`. TODO: reimplement `inverts` above using this one

```
Ltac inverts_as_tactic H :=
  let rec go tt :=
    match goal with
    | |- (ltac_Mark → _) ⇒ intros _
    | |- (?x = ?y → _) ⇒ let H := fresh "TEMP" in intro H;
                           first [ subst x | subst y ];
                           go tt
    | |- (existT ?P ?p ?x = existT ?P ?p ?y → _) ⇒
```

```
          let H := fresh in intro H;
          generalize (@inj_pair2 _ P p x y H);
          clear H; go tt
     | |- (∀ _, _) ⇒
        intro; let H := get_last_hyp tt in mark_to_generalize H; go
tt
     end in
   pose ltac_mark; inversion H;
   generalize ltac_mark; gen_until_mark;
   go tt; gen_to_generalize; unfolds ltac_to_generalize;
   unfold eq' in *.

Tactic Notation "inverts" "keep" hyp(H) "as" :=
  inverts_as_tactic H.

Tactic Notation "inverts" hyp(H) "as" :=
  inverts_as_tactic H; clear H.

Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3)
simple_intropattern(I4)
 simple_intropattern(I5) simple_intropattern(I6)
simple_intropattern(I7) :=
  inverts H as; introv I1 I2 I3 I4 I5 I6 I7.
Tactic Notation "inverts" hyp(H) "as" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3)
simple_intropattern(I4)
 simple_intropattern(I5) simple_intropattern(I6)
simple_intropattern(I7)
 simple_intropattern(I8) :=
  inverts H as; introv I1 I2 I3 I4 I5 I6 I7 I8.
```

`lets_inverts E as I1 .. IN` is intuitively equivalent to `inverts E`, with the difference that it applies to any expression and not just to the name of an hypothesis.

```
Ltac lets_inverts_base E cont :=
  let H := fresh "TEMP" in lets H: E; try cont H.

Tactic Notation "lets_inverts" constr(E) :=
  lets_inverts_base E ltac:(fun H ⇒ inverts H).
Tactic Notation "lets_inverts" constr(E) "as"
simple_intropattern(I1) :=
  lets_inverts_base E ltac:(fun H ⇒ inverts H as I1).
Tactic Notation "lets_inverts" constr(E) "as"
simple_intropattern(I1)
 simple_intropattern(I2) :=
  lets_inverts_base E ltac:(fun H ⇒ inverts H as I1 I2).
Tactic Notation "lets_inverts" constr(E) "as"
simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3) :=
  lets_inverts_base E ltac:(fun H ⇒ inverts H as I1 I2 I3).
Tactic Notation "lets_inverts" constr(E) "as"
simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3)
```

```
  simple_intropattern(I4) :=
    lets_inverts_base E ltac:(fun H ⇒ inverts H as I1 I2 I3 I4).
```

Injection with Substitution

Underlying implementation of `injects`

```
Ltac injects_tactic H :=
  let rec go _ :=
    match goal with
    | |- (ltac_Mark → _) ⇒ intros _
    | |- (?x = ?y → _) ⇒ let H := fresh in intro H;
                            first [ subst x | subst y | idtac ];
                            go tt
    end in
  generalize ltac_mark; injection H; go tt.
```

`injects keep H` takes an hypothesis `H` of the form `C a1 .. aN = C b1 .. bN` and substitute all equalities `ai = bi` that have been generated.

```
Tactic Notation "injects" "keep" hyp(H) :=
  injects_tactic H.
```

`injects H` is similar to `injects keep H` but clears the hypothesis `H`.

```
Tactic Notation "injects" hyp(H) :=
  injects_tactic H; clear H.
```

`inject H as X1 .. XN` is the same as `injection` followed by `intros X1 .. XN`

```
Tactic Notation "inject" hyp(H) :=
  injection H.
Tactic Notation "inject" hyp(H) "as" ident(X1) :=
  injection H; intros X1.
Tactic Notation "inject" hyp(H) "as" ident(X1) ident(X2) :=
  injection H; intros X1 X2.
Tactic Notation "inject" hyp(H) "as" ident(X1) ident(X2) ident(X3)
:=
  injection H; intros X1 X2 X3.
Tactic Notation "inject" hyp(H) "as" ident(X1) ident(X2) ident(X3)
 ident(X4) :=
  injection H; intros X1 X2 X3 X4.
Tactic Notation "inject" hyp(H) "as" ident(X1) ident(X2) ident(X3)
 ident(X4) ident(X5) :=
  injection H; intros X1 X2 X3 X4 X5.
```

Inversion and Injection with Substitution —rough implementation

The tactics `inversions` and `injections` provided in this section are similar to `inverts` and `injects` except that they perform substitution on all equalities from the context and not only the ones freshly generated. The counterpart is that they have simpler implementations.

inversions keep H is the same as inversions H but it does not clear hypothesis H.

```
Tactic Notation "inversions" "keep" hyp(H) :=
  inversion H; subst.
```

inversions H is a shortcut for inversion H followed by subst and clear H. It is a rough implementation of inverts keep H which behave badly when the proof context already contains equalities. It is provided in case the better implementation turns out to be too slow.

```
Tactic Notation "inversions" hyp(H) :=
  inversion H; subst; clear H.
```

injections keep H is the same as injection H followed by intros and subst. It is a rough implementation of injects keep H which behave badly when the proof context already contains equalities, or when the goal starts with a forall or an implication.

```
Tactic Notation "injections" "keep" hyp(H) :=
  injection H; intros; subst.
```

injections H is the same as injection H followed by intros and clear H and subst. It is a rough implementation of injects keep H which behave badly when the proof context already contains equalities, or when the goal starts with a forall or an implication.

```
Tactic Notation "injections" "keep" hyp(H) :=
  injection H; clear H; intros; subst.
```

Case Analysis

cases is similar to case_eq E except that it generates the equality in the context and not in the goal, and generates the equality the other way round. The syntax cases E as H allows specifying the name H of that hypothesis.

```
Tactic Notation "cases" constr(E) "as" ident(H) :=
  let X := fresh "TEMP" in
  set (X := E) in *; def_to_eq_sym X H E;
  destruct X.

Tactic Notation "cases" constr(E) :=
  let H := fresh "Eq" in cases E as H.
```

case_if_post is to be defined later as a tactic to clean up goals. By defaults, it looks for obvious contradictions. Currently, this tactic is extended in LibReflect to clean up boolean propositions.

```
Ltac case_if_post := tryfalse.
```

case_if looks for a pattern of the form if ?B then ?E_1 else ?E_2 in the goal, and perform a case analysis on B by calling destruct B. Subgoals containing a contradiction are discarded. case_if looks in the goal first, and otherwise in the first hypothesis that contains and if statement. case_if in H can be used to specify which hypothesis to consider. Syntaxes case_if as Eq and case_if in H as Eq allows to name the hypothesis coming from the case analysis.

```
Ltac case_if_on_tactic_core E Eq :=
  match type of E with
  | {_}+{_} ⇒ destruct E as [Eq | Eq]
  | _ ⇒ let X := fresh in
         sets_eq <- X Eq: E;
         destruct X
  end.

Ltac case_if_on_tactic E Eq :=
  case_if_on_tactic_core E Eq; case_if_post.

Tactic Notation "case_if_on" constr(E) "as" simple_intropattern(Eq)
:=
  case_if_on_tactic E Eq.

Tactic Notation "case_if" "as" simple_intropattern(Eq) :=
  match goal with
  | |- context [if ?B then _ else _] ⇒ case_if_on B as Eq
  | K: context [if ?B then _ else _] |- _ ⇒ case_if_on B as Eq
  end.

Tactic Notation "case_if" "in" hyp(H) "as" simple_intropattern(Eq)
:=
  match type of H with context [if ?B then _ else _] ⇒
    case_if_on B as Eq end.

Tactic Notation "case_if" :=
  let Eq := fresh in case_if as Eq.

Tactic Notation "case_if" "in" hyp(H) :=
  let Eq := fresh in case_if in H as Eq.
```

`cases_if` is similar to `case_if` with two main differences: if it creates an equality of the form `x = y` and then substitutes it in the goal

```
Ltac cases_if_on_tactic_core E Eq :=
  match type of E with
  | {_}+{_} ⇒ destruct E as [Eq|Eq]; try subst_hyp Eq
  | _ ⇒ let X := fresh in
         sets_eq <- X Eq: E;
         destruct X
  end.

Ltac cases_if_on_tactic E Eq :=
  cases_if_on_tactic_core E Eq; tryfalse; case_if_post.

Tactic Notation "cases_if_on" constr(E) "as"
simple_intropattern(Eq) :=
  cases_if_on_tactic E Eq.

Tactic Notation "cases_if" "as" simple_intropattern(Eq) :=
  match goal with
  | |- context [if ?B then _ else _] ⇒ cases_if_on B as Eq
  | K: context [if ?B then _ else _] |- _ ⇒ cases_if_on B as Eq
  end.

Tactic Notation "cases_if" "in" hyp(H) "as" simple_intropattern(Eq)
:=
  match type of H with context [if ?B then _ else _] ⇒
    cases_if_on B as Eq end.
```

```
Tactic Notation "cases_if" :=
  let Eq := fresh in cases_if as Eq.

Tactic Notation "cases_if" "in" hyp(H) :=
  let Eq := fresh in cases_if in H as Eq.
```

`case_ifs` is like `repeat case_if`

```
Ltac case_ifs_core :=
  repeat case_if.

Tactic Notation "case_ifs" :=
  case_ifs_core.
```

`destruct_if` looks for a pattern of the form `if ?B then ?E1 else ?E2` in the goal, and perform a case analysis on `B` by calling `destruct B`. It looks in the goal first, and otherwise in the first hypothesis that contains and `if` statement.

```
Ltac destruct_if_post := tryfalse.

Tactic Notation "destruct_if"
 "as" simple_intropattern(Eq1) simple_intropattern(Eq2) :=
  match goal with
  | |- context [if ?B then _ else _] ⇒ destruct B as [Eq1|Eq2]
  | K: context [if ?B then _ else _] |- _ ⇒ destruct B as [Eq1|Eq2]
  end;
  destruct_if_post.

Tactic Notation "destruct_if" "in" hyp(H)
 "as" simple_intropattern(Eq1) simple_intropattern(Eq2) :=
  match type of H with context [if ?B then _ else _] ⇒
    destruct B as [Eq1|Eq2] end;
  destruct_if_post.

Tactic Notation "destruct_if" "as" simple_intropattern(Eq) :=
  destruct_if as Eq Eq.
Tactic Notation "destruct_if" "in" hyp(H) "as"
simple_intropattern(Eq) :=
  destruct_if in H as Eq Eq.

Tactic Notation "destruct_if" :=
  let Eq := fresh "C" in destruct_if as Eq Eq.
Tactic Notation "destruct_if" "in" hyp(H) :=
  let Eq := fresh "C" in destruct_if in H as Eq Eq.
```

BROKEN since v_8.5beta2.

`destruct_head_match` performs a case analysis on the argument of the head pattern matching when the goal has the form `match ?E with ...` or `match ?E with ... = _` or `_ = match ?E with ...`. Due to the limits of Ltac, this tactic will not fail if a match does not occur. Instead, it might perform a case analysis on an unspecified subterm from the goal. Warning: experimental.

```
Ltac find_head_match T :=
  match T with context [?E] ⇒
    match T with
    | E ⇒ fail 1
```

```
      | _ ⇒ constr:(E)
      end
    end.
```

```
Ltac destruct_head_match_core cont :=
  match goal with
  | |- ?T1 = ?T2 ⇒ first [ let E := find_head_match T1 in cont E
                          | let E := find_head_match T2 in cont E ]
  | |- ?T1 ⇒ let E := find_head_match T1 in cont E
  end;
  destruct_if_post.
```

```
Tactic Notation "destruct_head_match" "as" simple_intropattern(I)
:=
  destruct_head_match_core ltac:(fun E ⇒ destruct E as I).
```

```
Tactic Notation "destruct_head_match" :=
  destruct_head_match_core ltac:(fun E ⇒ destruct E).
```

```
(**--provided for compatibility with remember *)
```

`cases' E` is similar to `case_eq E` except that it generates the equality in the context and not in the goal. The syntax `cases E as H` allows specifying the name `H` of that hypothesis.

```
Tactic Notation "cases'" constr(E) "as" ident(H) :=
  let X := fresh "TEMP" in
  set (X := E) in *; def_to_eq X H E;
  destruct X.
```

```
Tactic Notation "cases'" constr(E) :=
  let x := fresh "Eq" in cases' E as H.
```

`cases_if'` is similar to `cases_if` except that it generates the symmetric equality.

```
Ltac cases_if_on' E Eq :=
  match type of E with
  | {_}+{_} ⇒ destruct E as [Eq|Eq]; try subst_hyp Eq
  | _ ⇒ let X := fresh in
         sets_eq X Eq: E;
         destruct X
  end; case_if_post.
```

```
Tactic Notation "cases_if'" "as" simple_intropattern(Eq) :=
  match goal with
  | |- context [if ?B then _ else _] ⇒ cases_if_on' B Eq
  | K: context [if ?B then _ else _] |- _ ⇒ cases_if_on' B Eq
  end.
```

```
Tactic Notation "cases_if'" :=
  let Eq := fresh in cases_if' as Eq.
```

Induction

`inductions E` is a shorthand for `dependent induction E`. `inductions E gen X1 .. XN` is a shorthand for `dependent induction E generalizing X1 .. XN`.

```
Require Import Coq.Program.Equality.

Ltac inductions_post :=
  unfold eq' in *.

Tactic Notation "inductions" ident(E) :=
  dependent induction E; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) :=
  dependent induction E generalizing X1; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2) :=
  dependent induction E generalizing X1 X2; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2)
 ident(X3) :=
  dependent induction E generalizing X1 X2 X3; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2)
 ident(X3) ident(X4) :=
  dependent induction E generalizing X1 X2 X3 X4; inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2)
 ident(X3) ident(X4) ident(X5) :=
  dependent induction E generalizing X1 X2 X3 X4 X5;
inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2)
 ident(X3) ident(X4) ident(X5) ident(X6) :=
  dependent induction E generalizing X1 X2 X3 X4 X5 X6;
inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2)
 ident(X3) ident(X4) ident(X5) ident(X6) ident(X7) :=
  dependent induction E generalizing X1 X2 X3 X4 X5 X6 X7;
inductions_post.
Tactic Notation "inductions" ident(E) "gen" ident(X1) ident(X2)
 ident(X3) ident(X4) ident(X5) ident(X6) ident(X7) ident(X8) :=
  dependent induction E generalizing X1 X2 X3 X4 X5 X6 X7 X8;
inductions_post.
```

`induction_wf IH: E X` is used to apply the well-founded induction principle, for a given well-founded relation. It applies to a goal `PX` where `PX` is a proposition on `X`. First, it sets up the goal in the form `(fun a ⇒ P a) X`, using `pattern X`, and then it applies the well-founded induction principle instantiated on `E`, where `E` is a term of type `well_founded R`, and `R` is a binary relation. Syntaxes `induction_wf: E X` and `induction_wf E X`.

```
Tactic Notation "induction_wf" ident(IH) ":" constr(E) ident(X) :=
  pattern X; apply (well_founded_ind E); clear X; intros X IH.
Tactic Notation "induction_wf" ":" constr(E) ident(X) :=
  let IH := fresh "IH" in induction_wf IH: E X.
Tactic Notation "induction_wf" ":" constr(E) ident(X) :=
  induction_wf: E X.
```

Induction on the height of a derivation: the helper tactic `induct_height` helps proving the equivalence of the auxiliary judgment that includes a counter for the maximal height (see LibTacticsDemos for an example)

```
Require Import Compare_dec Omega.
```

```
Lemma induct_height_max2 : ∀ n1 n2 : nat,
  ∃ n, n1 < n ∧ n2 < n.
Proof using.
  intros. destruct (lt_dec n1 n2).
  ∃ (S n2). omega.
  ∃ (S n1). omega.
Qed.

Ltac induct_height_step x :=
  match goal with
  | H: ∃ _, _ |- _ ⇒
     let n := fresh "n" in let y := fresh "x" in
     destruct H as [n ?];
     forwards (y&?&?): induct_height_max2 n x;
     induct_height_step y
  | _ ⇒ ∃ (S x); eauto
 end.

Ltac induct_height := induct_height_step O.
```

Coinduction

Tactic `cofixs IH` is like `cofix IH` except that the coinduction hypothesis is tagged in the form `IH: COIND P` instead of being just `IH: P`. This helps other tactics clearing the coinduction hypothesis using `clear_coind`

```
Definition COIND (P:Prop) := P.

Tactic Notation "cofixs" ident(IH) :=
  cofix IH;
  match type of IH with ?P ⇒ change P with (COIND P) in IH end.
```

Tactic `clear_coind` clears all the coinduction hypotheses, assuming that they have been tagged

```
Ltac clear_coind :=
  repeat match goal with H: COIND _ |- _ ⇒ clear H end.
```

Tactic `abstracts tac` is like `abstract tac` except that it clears the coinduction hypotheses so that the productivity check will be happy. For example, one can use `abstracts omega` to obtain the same behavior as `omega` but with an auxiliary lemma being generated.

```
Tactic Notation "abstracts" tactic(tac) :=
  clear_coind; tac.
```

Decidable Equality

`decides_equality` is the same as `decide equality` excepts that it is able to unfold definitions at head of the current goal.

```
Ltac decides_equality_tactic :=
  first [ decide equality | progress(unfolds);
decides_equality_tactic ].

Tactic Notation "decides_equality" :=
  decides_equality_tactic.
```

Equivalence

`iff H` can be used to prove an equivalence `P ↔ Q` and name `H` the hypothesis obtained in each case. The syntaxes `iff` and `iff H1 H2` are also available to specify zero or two names. The tactic `iff <- H` swaps the two subgoals, i.e., produces (Q -> P) as first subgoal.

```
Lemma iff_intro_swap : ∀ (P Q : Prop),
  (Q → P) → (P → Q) → (P ↔ Q).
Proof using. intuition. Qed.

Tactic Notation "iff" simple_intropattern(H1)
simple_intropattern(H2) :=
  split; [ intros H1 | intros H2 ].
Tactic Notation "iff" simple_intropattern(H) :=
  iff H H.
Tactic Notation "iff" :=
  let H := fresh "H" in iff H.

Tactic Notation "iff" "<-" simple_intropattern(H1)
simple_intropattern(H2) :=
  apply iff_intro_swap; [ intros H1 | intros H2 ].
Tactic Notation "iff" "<-" simple_intropattern(H) :=
  iff <- H H.
Tactic Notation "iff" "<-" :=
  let H := fresh "H" in iff <- H.
```

N-ary Conjunctions and Disjunctions

N-ary Conjunctions Splitting in Goals

Underlying implementation of `splits`.

```
Ltac splits_tactic N :=
  match N with
  | O ⇒ fail
  | S O ⇒ idtac
  | S ?N' ⇒ split; [| splits_tactic N']
  end.

Ltac unfold_goal_until_conjunction :=
  match goal with
  | |- _ ∧ _ ⇒ idtac
```

```
    | _ ⇒ progress(unfolds); unfold_goal_until_conjunction
    end.

Ltac get_term_conjunction_arity T :=
  match T with
  | _ ∧ _ ∧ _ ∧ _ ∧ _ ∧ _ ∧ _ ∧ _ ⇒ constr:(8)
  | _ ∧ _ ∧ _ ∧ _ ∧ _ ∧ _ ∧ _ ⇒ constr:(7)
  | _ ∧ _ ∧ _ ∧ _ ∧ _ ∧ _ ⇒ constr:(6)
  | _ ∧ _ ∧ _ ∧ _ ∧ _ ⇒ constr:(5)
  | _ ∧ _ ∧ _ ∧ _ ⇒ constr:(4)
  | _ ∧ _ ∧ _ ⇒ constr:(3)
  | _ ∧ _ ⇒ constr:(2)
  | _ → ?T' ⇒ get_term_conjunction_arity T'
  | _ ⇒ let P := get_head T in
        let T' := eval unfold P in T in
        match T' with
        | T ⇒ fail 1
        | _ ⇒ get_term_conjunction_arity T'
        end
        (* todo: warning this can loop... *)
  end.

Ltac get_goal_conjunction_arity :=
  match goal with |- ?T ⇒ get_term_conjunction_arity T end.
```

`splits` applies to a goal of the form `(T1 ∧ .. ∧ TN)` and destruct it into `N` subgoals `T1` .. `TN`. If the goal is not a conjunction, then it unfolds the head definition.

```
Tactic Notation "splits" :=
  unfold_goal_until_conjunction;
  let N := get_goal_conjunction_arity in
  splits_tactic N.
```

`splits N` is similar to `splits`, except that it will unfold as many definitions as necessary to obtain an `N`-ary conjunction.

```
Tactic Notation "splits" constr(N) :=
  let N := nat_from_number N in
  splits_tactic N.
```

`splits_all` will recursively split any conjunction, unfolding definitions when necessary. Warning: this tactic will loop on goals of the form `well_founded R`. Todo: fix this

```
Ltac splits_all_base := repeat split.

Tactic Notation "splits_all" :=
  splits_all_base.
```

N-ary Conjunctions Deconstruction

Underlying implementation of `destructs`.

```
Ltac destructs_conjunction_tactic N T :=
  match N with
  | 2 ⇒ destruct T as [? ?]
  | 3 ⇒ destruct T as [? [? ?]] | 4 ⇒ destruct T as [? [? [? ?]]]
| 5 ⇒ destruct T as [? [? [? [? ?]]]] | 6 ⇒ destruct T as [? [? [?
[? [? ?]]]]] | 7 ⇒ destruct T as [? [? [? [? [? [? ?]]]]]] end.
```

`destructs T` allows destructing a term `T` which is a N-ary conjunction. It is equivalent to `destruct T as (H1 .. HN)`, except that it does not require to manually specify N different names.

```
Tactic Notation "destructs" constr(T) :=
  let TT := type of T in
  let N := get_term_conjunction_arity TT in
  destructs_conjunction_tactic N T.
```

`destructs N T` is equivalent to `destruct T as (H1 .. HN)`, except that it does not require to manually specify N different names. Remark that it is not restricted to N-ary conjunctions.

```
Tactic Notation "destructs" constr(N) constr(T) :=
  let N := nat_from_number N in
  destructs_conjunction_tactic N T.
```

Proving goals which are N-ary disjunctions

Underlying implementation of `branch`.

```
Ltac branch_tactic K N :=
  match constr:((K,N)) with
  | (_,0) ⇒ fail 1
  | (0,_) ⇒ fail 1
  | (1,1) ⇒ idtac
  | (1,_) ⇒ left
  | (S ?K', S ?N') ⇒ right; branch_tactic K' N'
  end.

Ltac unfold_goal_until_disjunction :=
  match goal with
  | |- _ ∨ _ ⇒ idtac
  | _ ⇒ progress(unfolds); unfold_goal_until_disjunction
  end.

Ltac get_term_disjunction_arity T :=
  match T with
  | _ ∨ _ ∨ _ ∨ _ ∨ _ ∨ _ ∨ _ ∨ _ ⇒ constr:(8)
  | _ ∨ _ ∨ _ ∨ _ ∨ _ ∨ _ ∨ _ ⇒ constr:(7)
  | _ ∨ _ ∨ _ ∨ _ ∨ _ ∨ _ ⇒ constr:(6)
  | _ ∨ _ ∨ _ ∨ _ ∨ _ ⇒ constr:(5)
  | _ ∨ _ ∨ _ ∨ _ ⇒ constr:(4)
  | _ ∨ _ ∨ _ ⇒ constr:(3)
  | _ ∨ _ ⇒ constr:(2)
  | _ → ?T' ⇒ get_term_disjunction_arity T'
  | _ ⇒ let P := get_head T in
        let T' := eval unfold P in T in
        match T' with
        | T ⇒ fail 1
        | _ ⇒ get_term_disjunction_arity T'
        end
  end.

Ltac get_goal_disjunction_arity :=
  match goal with |- ?T ⇒ get_term_disjunction_arity T end.
```

branch N applies to a goal of the form P_1 ∨ ... ∨ PK ∨ ... ∨ PN and leaves the goal PK. It only able to unfold the head definition (if there is one), but for more complex unfolding one should use the tactic branch K of N.

```
Tactic Notation "branch" constr(K) :=
  let K := nat_from_number K in
  unfold_goal_until_disjunction;
  let N := get_goal_disjunction_arity in
  branch_tactic K N.
```

branch K of N is similar to branch K except that the arity of the disjunction N is given manually, and so this version of the tactic is able to unfold definitions. In other words, applies to a goal of the form P_1 ∨ ... ∨ PK ∨ ... ∨ PN and leaves the goal PK.

```
Tactic Notation "branch" constr(K) "of" constr(N) :=
  let N := nat_from_number N in
  let K := nat_from_number K in
  branch_tactic K N.
```

N-ary Disjunction Deconstruction

Underlying implementation of branches.

```
Ltac destructs_disjunction_tactic N T :=
  match N with
  | 2 ⇒ destruct T as [? | ?]
  | 3 ⇒ destruct T as [? | [? | ?]] | 4 ⇒ destruct T as [? | [? |
[? | ?]]] | 5 ⇒ destruct T as [? | [? | [? | [? | ?]]]] end.
```

branches T allows destructing a term T which is a N-ary disjunction. It is equivalent to destruct T as [H_1 | .. | HN] , and produces N subgoals corresponding to the N possible cases.

```
Tactic Notation "branches" constr(T) :=
  let TT := type of T in
  let N := get_term_disjunction_arity TT in
  destructs_disjunction_tactic N T.
```

branches N T is the same as branches T except that the arity is forced to N. This version is useful to unfold definitions on the fly.

```
Tactic Notation "branches" constr(N) constr(T) :=
  let N := nat_from_number N in
  destructs_disjunction_tactic N T.
```

N-ary Existentials

```
(* Underlying implementation of ∃. *)

Ltac get_term_existential_arity T :=
  match T with
  | ∃ x1 x2 x3 x4 x5 x6 x7 x8, _ ⇒ constr:(8)
  | ∃ x1 x2 x3 x4 x5 x6 x7, _ ⇒ constr:(7)
  | ∃ x1 x2 x3 x4 x5 x6, _ ⇒ constr:(6)
  | ∃ x1 x2 x3 x4 x5, _ ⇒ constr:(5)
```

```
    | ∃ x1 x2 x3 x4, _ ⇒ constr:(4)
    | ∃ x1 x2 x3, _ ⇒ constr:(3)
    | ∃ x1 x2, _ ⇒ constr:(2)
    | ∃ x1, _ ⇒ constr:(1)
    | _ → ?T' ⇒ get_term_existential_arity T'
    | _ ⇒ let P := get_head T in
          let T' := eval unfold P in T in
          match T' with
          | T ⇒ fail 1
          | _ ⇒ get_term_existential_arity T'
          end
    end.

Ltac get_goal_existential_arity :=
  match goal with |- ?T ⇒ get_term_existential_arity T end.
```

`∃T1 ... TN` is a shorthand for `∃T1; ...; ∃TN`. It is intended to prove goals of the form `exist X1 .. XN, P`. If an argument provided is __ (double underscore), then an evar is introduced. `∃T1 .. TN ___` is equivalent to `∃T1 .. TN __ __ __` with as many __ as possible.

```
Tactic Notation "exists_original" constr(T1) :=
  ∃ T1.
Tactic Notation "exists" constr(T1) :=
  match T1 with
  | ltac_wild ⇒ esplit
  | ltac_wilds ⇒ repeat esplit
  | _ ⇒ ∃ T1
  end.
Tactic Notation "exists" constr(T1) constr(T2) :=
  ∃ T1; ∃ T2.
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) :=
  ∃ T1; ∃ T2; ∃ T3.
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) constr(T4)
:=
  ∃ T1; ∃ T2; ∃ T3; ∃ T4.
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) constr(T4)
 constr(T5) :=
  ∃ T1; ∃ T2; ∃ T3; ∃ T4; ∃ T5.
Tactic Notation "exists" constr(T1) constr(T2) constr(T3) constr(T4)
 constr(T5) constr(T6) :=
  ∃ T1; ∃ T2; ∃ T3; ∃ T4; ∃ T5; ∃ T6.

(* The tactic exists___ N is short for ∃__ ... __
   with N double-underscores. The tactic ∃is equivalent
   to calling exists___ N, where the value of N is obtained
   by counting the number of existentials syntactically present
   at the head of the goal. The behaviour of ∃differs
   from that of ∃___ is the case where the goal is a
   definition which yields an existential only after unfolding. *)
```

```
Tactic Notation "exists___" constr(N) :=
  let rec aux N :=
    match N with
    | 0 ⇒ idtac
    | S ?N' ⇒ esplit; aux N'
    end in
  let N := nat_from_number N in aux N.

  (* todo: deprecated *)
Tactic Notation "exists___" :=
  let N := get_goal_existential_arity in
  exists___ N.

  (* todo: does not seem to work *)
Tactic Notation "exists" :=
  exists___.

  (* todo: exists_all is the new syntax for exists___ *)
Tactic Notation "exists_all" := exists___.
```

Existentials and conjunctions in hypotheses

`unpack` or `unpack H` destructs conjunctions and existentials in all or one hypothesis.

```
Ltac unpack_core :=
  repeat match goal with
  | H: _ ∧ _ |- _ ⇒ destruct H
  | H: ∃ a, _ |- _ ⇒ destruct H
  end.

Ltac unpack_from H :=
  first [ progress (unpack_core)
        | destruct H; unpack_core ].

Tactic Notation "unpack" :=
  unpack_core.
Tactic Notation "unpack" constr(H) :=
  unpack_from H.
```

Tactics to Prove Typeclass Instances

`typeclass` is an automation tactic specialized for finding typeclass instances.

```
Tactic Notation "typeclass" :=
  let go _ := eauto with typeclass_instances in
  solve [ go tt | constructor; go tt ].
```

`solve_typeclass` is a simpler version of `typeclass`, to use in hint tactics for resolving instances

```
Tactic Notation "solve_typeclass" :=
  solve [ eauto with typeclass_instances ].
```

Tactics to Invoke Automation

Definitions for Parsing Compatibility

```
Tactic Notation "f_equal" :=
  f_equal.
Tactic Notation "constructor" :=
  constructor.
Tactic Notation "simple" :=
  simpl.

Tactic Notation "split" :=
  split.

Tactic Notation "right" :=
  right.
Tactic Notation "left" :=
  left.
```

`hint` to Add Hints Local to a Lemma

`hint E` adds `E` as an hypothesis so that automation can use it. Syntax `hint E1,..,EN` is available

```
Tactic Notation "hint" constr(E) :=
  let H := fresh "Hint" in lets H: E.
Tactic Notation "hint" constr(E1) "," constr(E2) :=
  hint E1; hint E2.
Tactic Notation "hint" constr(E1) "," constr(E2) "," constr(E3) :=
  hint E1; hint E2; hint(E3).
Tactic Notation "hint" constr(E1) "," constr(E2) "," constr(E3) ","
constr(E4) :=
  hint E1; hint E2; hint(E3); hint(E4 ).
```

`jauto`, a New Automation Tactic

`jauto` is better at `intuition eauto` because it can open existentials from the context. In the same time, `jauto` can be faster than `intuition eauto` because it does not destruct disjunctions from the context. The strategy of `jauto` can be summarized as follows:

- open all the existentials and conjunctions from the context
- call esplit and split on the existentials and conjunctions in the goal
- call eauto.

```
Tactic Notation "jauto" :=
  try solve [ jauto_set; eauto ].

Tactic Notation "jauto_fast" :=
  try solve [ auto | eauto | jauto ].
```

`iauto` is a shorthand for `intuition eauto`

```
Tactic Notation "iauto" := try solve [intuition eauto].
```

Definitions of Automation Tactics

The two following tactics defined the default behaviour of "light automation" and "strong automation". These tactics may be redefined at any time using the syntax `Ltac .. ::= ...`.

`auto_tilde` is the tactic which will be called each time a symbol `¬` is used after a tactic.

```
Ltac auto_tilde_default := auto.
Ltac auto_tilde := auto_tilde_default.
```

`auto_star` is the tactic which will be called each time a symbol `*` is used after a tactic.

```
(* SPECIAL VERSION FOR SF*)
Ltac auto_star_default := try solve [ jauto ].
Ltac auto_star := auto_star_default.
```

`autos¬` is a notation for tactic `auto_tilde`. It may be followed by lemmas (or proofs terms) which auto will be able to use for solving the goal. `autos` is an alias for `autos¬`

```
Tactic Notation "autos" :=
  auto_tilde.
Tactic Notation "autos" "¬" :=
  auto_tilde.
Tactic Notation "autos" "¬" constr(E1) :=
  lets: E1; auto_tilde.
Tactic Notation "autos" "¬" constr(E1) constr(E2) :=
  lets: E1; lets: E2; auto_tilde.
Tactic Notation "autos" "¬" constr(E1) constr(E2) constr(E3) :=
  lets: E1; lets: E2; lets: E3; auto_tilde.
```

`autos*` is a notation for tactic `auto_star`. It may be followed by lemmas (or proofs terms) which auto will be able to use for solving the goal.

```
Tactic Notation "autos" "*" :=
  auto_star.
Tactic Notation "autos" "*" constr(E1) :=
  lets: E1; auto_star.
Tactic Notation "autos" "*" constr(E1) constr(E2) :=
  lets: E1; lets: E2; auto_star.
Tactic Notation "autos" "*" constr(E1) constr(E2) constr(E3) :=
  lets: E1; lets: E2; lets: E3; auto_star.
```

`auto_false` is a version of `auto` able to spot some contradictions. There is an ad-hoc support for goals in `↔`: split is called first. `auto_false¬` and `auto_false*` are also available.

```
Ltac auto_false_base cont :=
  try solve [
    intros_all; try match goal with |- _ ↔ _ ⇒ split end;
    solve [ cont tt | intros_all; false; cont tt ] ].

Tactic Notation "auto_false" :=
   auto_false_base ltac:(fun tt ⇒ auto).
Tactic Notation "auto_false" "¬" :=
   auto_false_base ltac:(fun tt ⇒ auto_tilde).
Tactic Notation "auto_false" "*" :=
   auto_false_base ltac:(fun tt ⇒ auto_star).
```

Parsing for Light Automation

Any tactic followed by the symbol ¬ will have `auto_tilde` called on all of its subgoals. Three exceptions:

- `cuts` and `asserts` only call `auto` on their first subgoal,
- `apply¬` relies on `sapply` rather than `apply`,
- `tryfalse¬` is defined as `tryfalse by auto_tilde`.

Some builtin tactics are not defined using tactic notations and thus cannot be extended, e.g., `simpl` and `unfold`. For these, notation such as `simpl¬` will not be available.

```
Tactic Notation "equates" "¬" constr(E) :=
   equates E; auto_tilde.
Tactic Notation "equates" "¬" constr(n1) constr(n2) :=
  equates n1 n2; auto_tilde.
Tactic Notation "equates" "¬" constr(n1) constr(n2) constr(n3) :=
  equates n1 n2 n3; auto_tilde.
Tactic Notation "equates" "¬" constr(n1) constr(n2) constr(n3)
constr(n4) :=
  equates n1 n2 n3 n4; auto_tilde.

Tactic Notation "applys_eq" "¬" constr(H) constr(E) :=
  applys_eq H E; auto_tilde.
Tactic Notation "applys_eq" "¬" constr(H) constr(n1) constr(n2) :=
  applys_eq H n1 n2; auto_tilde.
Tactic Notation "applys_eq" "¬" constr(H) constr(n1) constr(n2)
constr(n3) :=
  applys_eq H n1 n2 n3; auto_tilde.
Tactic Notation "applys_eq" "¬" constr(H) constr(n1) constr(n2)
constr(n3) constr(n4) :=
  applys_eq H n1 n2 n3 n4; auto_tilde.

Tactic Notation "apply" "¬" constr(H) :=
  sapply H; auto_tilde.

Tactic Notation "destruct" "¬" constr(H) :=
  destruct H; auto_tilde.
Tactic Notation "destruct" "¬" constr(H) "as"
simple_intropattern(I) :=
  destruct H as I; auto_tilde.
Tactic Notation "f_equal" "¬" :=
  f_equal; auto_tilde.
Tactic Notation "induction" "¬" constr(H) :=
  induction H; auto_tilde.
Tactic Notation "inversion" "¬" constr(H) :=
  inversion H; auto_tilde.
Tactic Notation "split" "¬" :=
  split; auto_tilde.
Tactic Notation "subst" "¬" :=
  subst; auto_tilde.
Tactic Notation "right" "¬" :=
  right; auto_tilde.
Tactic Notation "left" "¬" :=
  left; auto_tilde.
```

```
Tactic Notation "constructor" "¬" :=
  constructor; auto_tilde.
Tactic Notation "constructors" "¬" :=
  constructors; auto_tilde.

Tactic Notation "false" "¬" :=
  false; auto_tilde.
Tactic Notation "false" "¬" constr(E) :=
  false_then E ltac:(fun _ ⇒ auto_tilde).
Tactic Notation "false" "¬" constr(E0) constr(E1) :=
  false¬ (>> E0 E1).
Tactic Notation "false" "¬" constr(E0) constr(E1) constr(E2) :=
  false¬ (>> E0 E1 E2).
Tactic Notation "false" "¬" constr(E0) constr(E1) constr(E2)
constr(E3) :=
  false¬ (>> E0 E1 E2 E3).
Tactic Notation "false" "¬" constr(E0) constr(E1) constr(E2)
constr(E3) constr(E4) :=
  false¬ (>> E0 E1 E2 E3 E4).
Tactic Notation "tryfalse" "¬" :=
  try solve [ false¬ ].

Tactic Notation "asserts" "¬" simple_intropattern(H) ":" constr(E)
:=
  asserts H: E; [ auto_tilde | idtac ].
Tactic Notation "asserts" "¬" ":" constr(E) :=
  let H := fresh "H" in asserts¬ H: E.
Tactic Notation "cuts" "¬" simple_intropattern(H) ":" constr(E) :=
  cuts H: E; [ auto_tilde | idtac ].
Tactic Notation "cuts" "¬" ":" constr(E) :=
  cuts: E; [ auto_tilde | idtac ].

Tactic Notation "lets" "¬" simple_intropattern(I) ":" constr(E) :=
  lets I: E; auto_tilde.
Tactic Notation "lets" "¬" simple_intropattern(I) ":" constr(E0)
 constr(A1) :=
  lets I: E0 A1; auto_tilde.
Tactic Notation "lets" "¬" simple_intropattern(I) ":" constr(E0)
 constr(A1) constr(A2) :=
  lets I: E0 A1 A2; auto_tilde.
Tactic Notation "lets" "¬" simple_intropattern(I) ":" constr(E0)
 constr(A1) constr(A2) constr(A3) :=
  lets I: E0 A1 A2 A3; auto_tilde.
Tactic Notation "lets" "¬" simple_intropattern(I) ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) :=
  lets I: E0 A1 A2 A3 A4; auto_tilde.
Tactic Notation "lets" "¬" simple_intropattern(I) ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  lets I: E0 A1 A2 A3 A4 A5; auto_tilde.

Tactic Notation "lets" "¬" ":" constr(E) :=
  lets: E; auto_tilde.
Tactic Notation "lets" "¬" ":" constr(E0)
```

```
 constr(A1) :=
  lets: E0 A1; auto_tilde.
Tactic Notation "lets" "¬" ":" constr(E0)
 constr(A1) constr(A2) :=
  lets: E0 A1 A2; auto_tilde.
Tactic Notation "lets" "¬" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) :=
  lets: E0 A1 A2 A3; auto_tilde.
Tactic Notation "lets" "¬" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) :=
  lets: E0 A1 A2 A3 A4; auto_tilde.
Tactic Notation "lets" "¬" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  lets: E0 A1 A2 A3 A4 A5; auto_tilde.

Tactic Notation "forwards" "¬" simple_intropattern(I) ":" constr(E)
:=
  forwards I: E; auto_tilde.
Tactic Notation "forwards" "¬" simple_intropattern(I) ":"
constr(E0)
 constr(A1) :=
  forwards I: E0 A1; auto_tilde.
Tactic Notation "forwards" "¬" simple_intropattern(I) ":"
constr(E0)
 constr(A1) constr(A2) :=
  forwards I: E0 A1 A2; auto_tilde.
Tactic Notation "forwards" "¬" simple_intropattern(I) ":"
constr(E0)
 constr(A1) constr(A2) constr(A3) :=
  forwards I: E0 A1 A2 A3; auto_tilde.
Tactic Notation "forwards" "¬" simple_intropattern(I) ":"
constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) :=
  forwards I: E0 A1 A2 A3 A4; auto_tilde.
Tactic Notation "forwards" "¬" simple_intropattern(I) ":"
constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  forwards I: E0 A1 A2 A3 A4 A5; auto_tilde.

Tactic Notation "forwards" "¬" ":" constr(E) :=
  forwards: E; auto_tilde.
Tactic Notation "forwards" "¬" ":" constr(E0)
 constr(A1) :=
  forwards: E0 A1; auto_tilde.
Tactic Notation "forwards" "¬" ":" constr(E0)
 constr(A1) constr(A2) :=
  forwards: E0 A1 A2; auto_tilde.
Tactic Notation "forwards" "¬" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) :=
  forwards: E0 A1 A2 A3; auto_tilde.
```

```
Tactic Notation "forwards" "¬" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) :=
  forwards: E0 A1 A2 A3 A4; auto_tilde.
Tactic Notation "forwards" "¬" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  forwards: E0 A1 A2 A3 A4 A5; auto_tilde.

Tactic Notation "applys" "¬" constr(H) :=
  sapply H; auto_tilde. (*todo?*)
Tactic Notation "applys" "¬" constr(E0) constr(A1) :=
  applys E0 A1; auto_tilde.
Tactic Notation "applys" "¬" constr(E0) constr(A1) :=
  applys E0 A1; auto_tilde.
Tactic Notation "applys" "¬" constr(E0) constr(A1) constr(A2) :=
  applys E0 A1 A2; auto_tilde.
Tactic Notation "applys" "¬" constr(E0) constr(A1) constr(A2)
constr(A3) :=
  applys E0 A1 A2 A3; auto_tilde.
Tactic Notation "applys" "¬" constr(E0) constr(A1) constr(A2)
constr(A3) constr(A4) :=
  applys E0 A1 A2 A3 A4; auto_tilde.
Tactic Notation "applys" "¬" constr(E0) constr(A1) constr(A2)
constr(A3) constr(A4) constr(A5) :=
  applys E0 A1 A2 A3 A4 A5; auto_tilde.

Tactic Notation "specializes" "¬" hyp(H) :=
  specializes H; auto_tilde.
Tactic Notation "specializes" "¬" hyp(H) constr(A1) :=
  specializes H A1; auto_tilde.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) :=
  specializes H A1 A2; auto_tilde.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2)
constr(A3) :=
  specializes H A1 A2 A3; auto_tilde.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2)
constr(A3) constr(A4) :=
  specializes H A1 A2 A3 A4; auto_tilde.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2)
constr(A3) constr(A4) constr(A5) :=
  specializes H A1 A2 A3 A4 A5; auto_tilde.

Tactic Notation "fapply" "¬" constr(E) :=
  fapply E; auto_tilde.
Tactic Notation "sapply" "¬" constr(E) :=
  sapply E; auto_tilde.

Tactic Notation "logic" "¬" constr(E) :=
  logic_base E ltac:(fun _ ⇒ auto_tilde).

Tactic Notation "intros_all" "¬" :=
  intros_all; auto_tilde.
```

```
Tactic Notation "unfolds" "¬" :=
  unfolds; auto_tilde.
Tactic Notation "unfolds" "¬" constr(F1) :=
  unfolds F1; auto_tilde.
Tactic Notation "unfolds" "¬" constr(F1) "," constr(F2) :=
  unfolds F1, F2; auto_tilde.
Tactic Notation "unfolds" "¬" constr(F1) "," constr(F2) ","
constr(F3) :=
  unfolds F1, F2, F3; auto_tilde.
Tactic Notation "unfolds" "¬" constr(F1) "," constr(F2) ","
constr(F3) ","
 constr(F4) :=
  unfolds F1, F2, F3, F4; auto_tilde.

Tactic Notation "simple" "¬" :=
  simpl; auto_tilde.
Tactic Notation "simple" "¬" "in" hyp(H) :=
  simpl in H; auto_tilde.
Tactic Notation "simpls" "¬" :=
  simpls; auto_tilde.
Tactic Notation "hnfs" "¬" :=
  hnfs; auto_tilde.
Tactic Notation "hnfs" "¬" "in" hyp(H) :=
  hnf in H; auto_tilde.
Tactic Notation "substs" "¬" :=
  substs; auto_tilde.
Tactic Notation "intro_hyp" "¬" hyp(H) :=
  subst_hyp H; auto_tilde.
Tactic Notation "intro_subst" "¬" :=
  intro_subst; auto_tilde.
Tactic Notation "subst_eq" "¬" constr(E) :=
  subst_eq E; auto_tilde.

Tactic Notation "rewrite" "¬" constr(E) :=
  rewrite E; auto_tilde.
Tactic Notation "rewrite" "¬" "<-" constr(E) :=
  rewrite <- E; auto_tilde.
Tactic Notation "rewrite" "¬" constr(E) "in" hyp(H) :=
  rewrite E in H; auto_tilde.
Tactic Notation "rewrite" "¬" "<-" constr(E) "in" hyp(H) :=
  rewrite <- E in H; auto_tilde.

Tactic Notation "rewrites" "¬" constr(E) :=
  rewrites E; auto_tilde.
Tactic Notation "rewrites" "¬" constr(E) "in" hyp(H) :=
  rewrites E in H; auto_tilde.
Tactic Notation "rewrites" "¬" constr(E) "in" "*" :=
  rewrites E in *; auto_tilde.
Tactic Notation "rewrites" "¬" "<-" constr(E) :=
  rewrites <- E; auto_tilde.
Tactic Notation "rewrites" "¬" "<-" constr(E) "in" hyp(H) :=
  rewrites <- E in H; auto_tilde.
Tactic Notation "rewrites" "¬" "<-" constr(E) "in" "*" :=
  rewrites <- E in *; auto_tilde.

Tactic Notation "rewrite_all" "¬" constr(E) :=
  rewrite_all E; auto_tilde.
```

```
Tactic Notation "rewrite_all" "¬" "<-" constr(E) :=
  rewrite_all <- E; auto_tilde.
Tactic Notation "rewrite_all" "¬" constr(E) "in" ident(H) :=
  rewrite_all E in H; auto_tilde.
Tactic Notation "rewrite_all" "¬" "<-" constr(E) "in" ident(H) :=
  rewrite_all <- E in H; auto_tilde.
Tactic Notation "rewrite_all" "¬" constr(E) "in" "*" :=
  rewrite_all E in *; auto_tilde.
Tactic Notation "rewrite_all" "¬" "<-" constr(E) "in" "*" :=
  rewrite_all <- E in *; auto_tilde.

Tactic Notation "asserts_rewrite" "¬" constr(E) :=
  asserts_rewrite E; auto_tilde.
Tactic Notation "asserts_rewrite" "¬" "<-" constr(E) :=
  asserts_rewrite <- E; auto_tilde.
Tactic Notation "asserts_rewrite" "¬" constr(E) "in" hyp(H) :=
  asserts_rewrite E in H; auto_tilde.
Tactic Notation "asserts_rewrite" "¬" "<-" constr(E) "in" hyp(H) :=
  asserts_rewrite <- E in H; auto_tilde.
Tactic Notation "asserts_rewrite" "¬" constr(E) "in" "*" :=
  asserts_rewrite E in *; auto_tilde.
Tactic Notation "asserts_rewrite" "¬" "<-" constr(E) "in" "*" :=
  asserts_rewrite <- E in *; auto_tilde.

Tactic Notation "cuts_rewrite" "¬" constr(E) :=
  cuts_rewrite E; auto_tilde.
Tactic Notation "cuts_rewrite" "¬" "<-" constr(E) :=
  cuts_rewrite <- E; auto_tilde.
Tactic Notation "cuts_rewrite" "¬" constr(E) "in" hyp(H) :=
  cuts_rewrite E in H; auto_tilde.
Tactic Notation "cuts_rewrite" "¬" "<-" constr(E) "in" hyp(H) :=
  cuts_rewrite <- E in H; auto_tilde.

Tactic Notation "erewrite" "¬" constr(E) :=
  erewrite E; auto_tilde.

Tactic Notation "fequal" "¬" :=
  fequal; auto_tilde.
Tactic Notation "fequals" "¬" :=
  fequals; auto_tilde.
Tactic Notation "pi_rewrite" "¬" constr(E) :=
  pi_rewrite E; auto_tilde.
Tactic Notation "pi_rewrite" "¬" constr(E) "in" hyp(H) :=
  pi_rewrite E in H; auto_tilde.

Tactic Notation "invert" "¬" hyp(H) :=
  invert H; auto_tilde.
Tactic Notation "inverts" "¬" hyp(H) :=
  inverts H; auto_tilde.
Tactic Notation "inverts" "¬" hyp(E) "as" :=
  inverts E as; auto_tilde.
Tactic Notation "injects" "¬" hyp(H) :=
  injects H; auto_tilde.
Tactic Notation "inversions" "¬" hyp(H) :=
  inversions H; auto_tilde.

Tactic Notation "cases" "¬" constr(E) "as" ident(H) :=
  cases E as H; auto_tilde.
Tactic Notation "cases" "¬" constr(E) :=
  cases E; auto_tilde.
```

```
Tactic Notation "case_if" "¬" :=
  case_if; auto_tilde.
Tactic Notation "case_ifs" "¬" :=
  case_ifs; auto_tilde.
Tactic Notation "case_if" "¬" "in" hyp(H) :=
  case_if in H; auto_tilde.
Tactic Notation "cases_if" "¬" :=
  cases_if; auto_tilde.
Tactic Notation "cases_if" "¬" "in" hyp(H) :=
  cases_if in H; auto_tilde.
Tactic Notation "destruct_if" "¬" :=
  destruct_if; auto_tilde.
Tactic Notation "destruct_if" "¬" "in" hyp(H) :=
  destruct_if in H; auto_tilde.
Tactic Notation "destruct_head_match" "¬" :=
  destruct_head_match; auto_tilde.

Tactic Notation "cases'" "¬" constr(E) "as" ident(H) :=
  cases' E as H; auto_tilde.
Tactic Notation "cases'" "¬" constr(E) :=
  cases' E; auto_tilde.
Tactic Notation "cases_if'" "¬" "as" ident(H) :=
  cases_if' as H; auto_tilde.
Tactic Notation "cases_if'" "¬" :=
  cases_if'; auto_tilde.

Tactic Notation "decides_equality" "¬" :=
  decides_equality; auto_tilde.

Tactic Notation "iff" "¬" :=
  iff; auto_tilde.
Tactic Notation "splits" "¬" :=
  splits; auto_tilde.
Tactic Notation "splits" "¬" constr(N) :=
  splits N; auto_tilde.
Tactic Notation "splits_all" "¬" :=
  splits_all; auto_tilde.

Tactic Notation "destructs" "¬" constr(T) :=
  destructs T; auto_tilde.
Tactic Notation "destructs" "¬" constr(N) constr(T) :=
  destructs N T; auto_tilde.

Tactic Notation "branch" "¬" constr(N) :=
  branch N; auto_tilde.
Tactic Notation "branch" "¬" constr(K) "of" constr(N) :=
  branch K of N; auto_tilde.

Tactic Notation "branches" "¬" constr(T) :=
  branches T; auto_tilde.
Tactic Notation "branches" "¬" constr(N) constr(T) :=
  branches N T; auto_tilde.

Tactic Notation "exists" "¬" :=
  ∃; auto_tilde.
Tactic Notation "exists___" "¬" :=
  exists___; auto_tilde.
Tactic Notation "exists" "¬" constr(T1) :=
  ∃ T1; auto_tilde.
Tactic Notation "exists" "¬" constr(T1) constr(T2) :=
```

```
  ∃ T1 T2; auto_tilde.
Tactic Notation "exists" "¬" constr(T1) constr(T2) constr(T3) :=
  ∃ T1 T2 T3; auto_tilde.
Tactic Notation "exists" "¬" constr(T1) constr(T2) constr(T3)
constr(T4) :=
  ∃ T1 T2 T3 T4; auto_tilde.
Tactic Notation "exists" "¬" constr(T1) constr(T2) constr(T3)
constr(T4)
 constr(T5) :=
  ∃ T1 T2 T3 T4 T5; auto_tilde.
Tactic Notation "exists" "¬" constr(T1) constr(T2) constr(T3)
constr(T4)
 constr(T5) constr(T6) :=
  ∃ T1 T2 T3 T4 T5 T6; auto_tilde.
```

Parsing for Strong Automation

Any tactic followed by the symbol `*` will have `auto*` called on all of its subgoals. The exceptions to these rules are the same as for light automation.

Exception: use `subs*` instead of `subst*` if you import the library `Coq.Classes.Equivalence`.

```
Tactic Notation "equates" "*" constr(E) :=
   equates E; auto_star.
Tactic Notation "equates" "*" constr(n1) constr(n2) :=
  equates n1 n2; auto_star.
Tactic Notation "equates" "*" constr(n1) constr(n2) constr(n3) :=
  equates n1 n2 n3; auto_star.
Tactic Notation "equates" "*" constr(n1) constr(n2) constr(n3)
constr(n4) :=
  equates n1 n2 n3 n4; auto_star.

Tactic Notation "applys_eq" "*" constr(H) constr(E) :=
  applys_eq H E; auto_star.
Tactic Notation "applys_eq" "*" constr(H) constr(n1) constr(n2) :=
  applys_eq H n1 n2; auto_star.
Tactic Notation "applys_eq" "*" constr(H) constr(n1) constr(n2)
constr(n3) :=
  applys_eq H n1 n2 n3; auto_star.
Tactic Notation "applys_eq" "*" constr(H) constr(n1) constr(n2)
constr(n3) constr(n4) :=
  applys_eq H n1 n2 n3 n4; auto_star.

Tactic Notation "apply" "*" constr(H) :=
  sapply H; auto_star.

Tactic Notation "destruct" "*" constr(H) :=
  destruct H; auto_star.
Tactic Notation "destruct" "*" constr(H) "as"
simple_intropattern(I) :=
  destruct H as I; auto_star.
```

```
Tactic Notation "f_equal" "*" :=
  f_equal; auto_star.
Tactic Notation "induction" "*" constr(H) :=
  induction H; auto_star.
Tactic Notation "inversion" "*" constr(H) :=
  inversion H; auto_star.
Tactic Notation "split" "*" :=
  split; auto_star.
Tactic Notation "subs" "*" :=
  subst; auto_star.
Tactic Notation "subst" "*" :=
  subst; auto_star.
Tactic Notation "right" "*" :=
  right; auto_star.
Tactic Notation "left" "*" :=
  left; auto_star.
Tactic Notation "constructor" "*" :=
  constructor; auto_star.
Tactic Notation "constructors" "*" :=
  constructors; auto_star.

Tactic Notation "false" "*" :=
  false; auto_star.
Tactic Notation "false" "*" constr(E) :=
  false_then E ltac:(fun _ ⇒ auto_star).
Tactic Notation "false" "*" constr(E0) constr(E1) :=
  false* (>> E0 E1).
Tactic Notation "false" "*" constr(E0) constr(E1) constr(E2) :=
  false* (>> E0 E1 E2).
Tactic Notation "false" "*" constr(E0) constr(E1) constr(E2)
constr(E3) :=
  false* (>> E0 E1 E2 E3).
Tactic Notation "false" "*" constr(E0) constr(E1) constr(E2)
constr(E3) constr(E4) :=
  false* (>> E0 E1 E2 E3 E4).
Tactic Notation "tryfalse" "*" :=
  try solve [ false* ].

Tactic Notation "asserts" "*" simple_intropattern(H) ":" constr(E)
:=
  asserts H: E; [ auto_star | idtac ].
Tactic Notation "asserts" "*" ":" constr(E) :=
  let H := fresh "H" in asserts* H: E.
Tactic Notation "cuts" "*" simple_intropattern(H) ":" constr(E) :=
  cuts H: E; [ auto_star | idtac ].
Tactic Notation "cuts" "*" ":" constr(E) :=
  cuts: E; [ auto_star | idtac ].

Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E) :=
  lets I: E; auto_star.
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
 constr(A1) :=
  lets I: E0 A1; auto_star.
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
 constr(A1) constr(A2) :=
  lets I: E0 A1 A2; auto_star.
```

```
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
 constr(A1) constr(A2) constr(A3) :=
  lets I: E0 A1 A2 A3; auto_star.
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) :=
  lets I: E0 A1 A2 A3 A4; auto_star.
Tactic Notation "lets" "*" simple_intropattern(I) ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  lets I: E0 A1 A2 A3 A4 A5; auto_star.

Tactic Notation "lets" "*" ":" constr(E) :=
  lets: E; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
 constr(A1) :=
  lets: E0 A1; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
 constr(A1) constr(A2) :=
  lets: E0 A1 A2; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) :=
  lets: E0 A1 A2 A3; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) :=
  lets: E0 A1 A2 A3 A4; auto_star.
Tactic Notation "lets" "*" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  lets: E0 A1 A2 A3 A4 A5; auto_star.

Tactic Notation "forwards" "*" simple_intropattern(I) ":" constr(E)
:=
  forwards I: E; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":"
constr(E0)
 constr(A1) :=
  forwards I: E0 A1; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":"
constr(E0)
 constr(A1) constr(A2) :=
  forwards I: E0 A1 A2; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":"
constr(E0)
 constr(A1) constr(A2) constr(A3) :=
  forwards I: E0 A1 A2 A3; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":"
constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) :=
  forwards I: E0 A1 A2 A3 A4; auto_star.
Tactic Notation "forwards" "*" simple_intropattern(I) ":"
constr(E0)
```

```
 constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  forwards I: E0 A1 A2 A3 A4 A5; auto_star.

Tactic Notation "forwards" "*" ":" constr(E) :=
  forwards: E; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
 constr(A1) :=
  forwards: E0 A1; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
 constr(A1) constr(A2) :=
  forwards: E0 A1 A2; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) :=
  forwards: E0 A1 A2 A3; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) :=
  forwards: E0 A1 A2 A3 A4; auto_star.
Tactic Notation "forwards" "*" ":" constr(E0)
 constr(A1) constr(A2) constr(A3) constr(A4) constr(A5) :=
  forwards: E0 A1 A2 A3 A4 A5; auto_star.

Tactic Notation "applys" "*" constr(H) :=
  sapply H; auto_star. (*todo?*)
Tactic Notation "applys" "*" constr(E0) constr(A1) :=
  applys E0 A1; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) :=
  applys E0 A1; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) constr(A2) :=
  applys E0 A1 A2; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) constr(A2)
constr(A3) :=
  applys E0 A1 A2 A3; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) constr(A2)
constr(A3) constr(A4) :=
  applys E0 A1 A2 A3 A4; auto_star.
Tactic Notation "applys" "*" constr(E0) constr(A1) constr(A2)
constr(A3) constr(A4) constr(A5) :=
  applys E0 A1 A2 A3 A4 A5; auto_star.

Tactic Notation "specializes" "*" hyp(H) :=
  specializes H; auto_star.
Tactic Notation "specializes" "¬" hyp(H) constr(A1) :=
  specializes H A1; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2) :=
  specializes H A1 A2; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2)
constr(A3) :=
  specializes H A1 A2 A3; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2)
constr(A3) constr(A4) :=
```

```
  specializes H A1 A2 A3 A4; auto_star.
Tactic Notation "specializes" hyp(H) constr(A1) constr(A2)
constr(A3) constr(A4) constr(A5) :=
  specializes H A1 A2 A3 A4 A5; auto_star.

Tactic Notation "fapply" "*" constr(E) :=
  fapply E; auto_star.
Tactic Notation "sapply" "*" constr(E) :=
  sapply E; auto_star.

Tactic Notation "logic" constr(E) :=
  logic_base E ltac:(fun _ ⇒ auto_star).

Tactic Notation "intros_all" "*" :=
  intros_all; auto_star.

Tactic Notation "unfolds" "*" :=
  unfolds; auto_star.
Tactic Notation "unfolds" "*" constr(F1) :=
  unfolds F1; auto_star.
Tactic Notation "unfolds" "*" constr(F1) "," constr(F2) :=
  unfolds F1, F2; auto_star.
Tactic Notation "unfolds" "*" constr(F1) "," constr(F2) ","
constr(F3) :=
  unfolds F1, F2, F3; auto_star.
Tactic Notation "unfolds" "*" constr(F1) "," constr(F2) ","
constr(F3) ","
 constr(F4) :=
  unfolds F1, F2, F3, F4; auto_star.

Tactic Notation "simple" "*" :=
  simpl; auto_star.
Tactic Notation "simple" "*" "in" hyp(H) :=
  simpl in H; auto_star.
Tactic Notation "simpls" "*" :=
  simpls; auto_star.
Tactic Notation "hnfs" "*" :=
  hnfs; auto_star.
Tactic Notation "hnfs" "*" "in" hyp(H) :=
  hnf in H; auto_star.
Tactic Notation "substs" "*" :=
  substs; auto_star.
Tactic Notation "intro_hyp" "*" hyp(H) :=
  subst_hyp H; auto_star.
Tactic Notation "intro_subst" "*" :=
  intro_subst; auto_star.
Tactic Notation "subst_eq" "*" constr(E) :=
  subst_eq E; auto_star.

Tactic Notation "rewrite" "*" constr(E) :=
  rewrite E; auto_star.
Tactic Notation "rewrite" "*" "<-" constr(E) :=
  rewrite <- E; auto_star.
Tactic Notation "rewrite" "*" constr(E) "in" hyp(H) :=
  rewrite E in H; auto_star.
Tactic Notation "rewrite" "*" "<-" constr(E) "in" hyp(H) :=
  rewrite <- E in H; auto_star.
```

```
Tactic Notation "rewrites" "*" constr(E) :=
  rewrites E; auto_star.
Tactic Notation "rewrites" "*" constr(E) "in" hyp(H):=
  rewrites E in H; auto_star.
Tactic Notation "rewrites" "*" constr(E) "in" "*":=
  rewrites E in *; auto_star.
Tactic Notation "rewrites" "*" "<-" constr(E) :=
  rewrites <- E; auto_star.
Tactic Notation "rewrites" "*" "<-" constr(E) "in" hyp(H):=
  rewrites <- E in H; auto_star.
Tactic Notation "rewrites" "*" "<-" constr(E) "in" "*":=
  rewrites <- E in *; auto_star.

Tactic Notation "rewrite_all" "*" constr(E) :=
  rewrite_all E; auto_star.
Tactic Notation "rewrite_all" "*" "<-" constr(E) :=
  rewrite_all <- E; auto_star.
Tactic Notation "rewrite_all" "*" constr(E) "in" ident(H) :=
  rewrite_all E in H; auto_star.
Tactic Notation "rewrite_all" "*" "<-" constr(E) "in" ident(H) :=
  rewrite_all <- E in H; auto_star.
Tactic Notation "rewrite_all" "*" constr(E) "in" "*" :=
  rewrite_all E in *; auto_star.
Tactic Notation "rewrite_all" "*" "<-" constr(E) "in" "*" :=
  rewrite_all <- E in *; auto_star.

Tactic Notation "asserts_rewrite" "*" constr(E) :=
  asserts_rewrite E; auto_star.
Tactic Notation "asserts_rewrite" "*" "<-" constr(E) :=
  asserts_rewrite <- E; auto_star.
Tactic Notation "asserts_rewrite" "*" constr(E) "in" hyp(H) :=
  asserts_rewrite E; auto_star.
Tactic Notation "asserts_rewrite" "*" "<-" constr(E) "in" hyp(H) :=
  asserts_rewrite <- E; auto_star.
Tactic Notation "asserts_rewrite" "*" constr(E) "in" "*" :=
  asserts_rewrite E in *; auto_tilde.
Tactic Notation "asserts_rewrite" "*" "<-" constr(E) "in" "*" :=
  asserts_rewrite <- E in *; auto_tilde.

Tactic Notation "cuts_rewrite" "*" constr(E) :=
  cuts_rewrite E; auto_star.
Tactic Notation "cuts_rewrite" "*" "<-" constr(E) :=
  cuts_rewrite <- E; auto_star.
Tactic Notation "cuts_rewrite" "*" constr(E) "in" hyp(H) :=
  cuts_rewrite E in H; auto_star.
Tactic Notation "cuts_rewrite" "*" "<-" constr(E) "in" hyp(H) :=
  cuts_rewrite <- E in H; auto_star.

Tactic Notation "erewrite" "*" constr(E) :=
  erewrite E; auto_star.

Tactic Notation "fequal" "*" :=
  fequal; auto_star.
Tactic Notation "fequals" "*" :=
  fequals; auto_star.
Tactic Notation "pi_rewrite" "*" constr(E) :=
  pi_rewrite E; auto_star.
Tactic Notation "pi_rewrite" "*" constr(E) "in" hyp(H) :=
  pi_rewrite E in H; auto_star.
```

```
Tactic Notation "invert" "*" hyp(H) :=
  invert H; auto_star.
Tactic Notation "inverts" "*" hyp(H) :=
  inverts H; auto_star.
Tactic Notation "inverts" "*" hyp(E) "as" :=
  inverts E as; auto_star.
Tactic Notation "injects" "*" hyp(H) :=
  injects H; auto_star.
Tactic Notation "inversions" "*" hyp(H) :=
  inversions H; auto_star.

Tactic Notation "cases" "*" constr(E) "as" ident(H) :=
  cases E as H; auto_star.
Tactic Notation "cases" "*" constr(E) :=
  cases E; auto_star.
Tactic Notation "case_if" "*" :=
  case_if; auto_star.
Tactic Notation "case_ifs" "*" :=
  case_ifs; auto_star.
Tactic Notation "case_if" "*" "in" hyp(H) :=
  case_if in H; auto_star.
Tactic Notation "cases_if" "*" :=
  cases_if; auto_star.
Tactic Notation "cases_if" "*" "in" hyp(H) :=
  cases_if in H; auto_star.
 Tactic Notation "destruct_if" "*" :=
  destruct_if; auto_star.
Tactic Notation "destruct_if" "*" "in" hyp(H) :=
  destruct_if in H; auto_star.
Tactic Notation "destruct_head_match" "*" :=
  destruct_head_match; auto_star.

Tactic Notation "cases'" "*" constr(E) "as" ident(H) :=
  cases' E as H; auto_star.
Tactic Notation "cases'" "*" constr(E) :=
  cases' E; auto_star.
Tactic Notation "cases_if'" "*" "as" ident(H) :=
  cases_if' as H; auto_star.
Tactic Notation "cases_if'" "*" :=
  cases_if'; auto_star.

Tactic Notation "decides_equality" "*" :=
  decides_equality; auto_star.

Tactic Notation "iff" "*" :=
  iff; auto_star.
Tactic Notation "iff" "*" simple_intropattern(I) :=
  iff I; auto_star.
Tactic Notation "splits" "*" :=
  splits; auto_star.
Tactic Notation "splits" "*" constr(N) :=
  splits N; auto_star.
Tactic Notation "splits_all" "*" :=
  splits_all; auto_star.

Tactic Notation "destructs" "*" constr(T) :=
  destructs T; auto_star.
Tactic Notation "destructs" "*" constr(N) constr(T) :=
  destructs N T; auto_star.
```

```
Tactic Notation "branch" "*" constr(N) :=
  branch N; auto_star.
Tactic Notation "branch" "*" constr(K) "of" constr(N) :=
  branch K of N; auto_star.

Tactic Notation "branches" "*" constr(T) :=
  branches T; auto_star.
Tactic Notation "branches" "*" constr(N) constr(T) :=
  branches N T; auto_star.

Tactic Notation "exists" "*" :=
  ∃; auto_star.
Tactic Notation "exists___" "*" :=
  exists___; auto_star.
Tactic Notation "exists" "*" constr(T1) :=
  ∃ T1; auto_star.
Tactic Notation "exists" "*" constr(T1) constr(T2) :=
  ∃ T1 T2; auto_star.
Tactic Notation "exists" "*" constr(T1) constr(T2) constr(T3) :=
  ∃ T1 T2 T3; auto_star.
Tactic Notation "exists" "*" constr(T1) constr(T2) constr(T3)
constr(T4) :=
  ∃ T1 T2 T3 T4; auto_star.
Tactic Notation "exists" "*" constr(T1) constr(T2) constr(T3)
constr(T4)
 constr(T5) :=
  ∃ T1 T2 T3 T4 T5; auto_star.
Tactic Notation "exists" "*" constr(T1) constr(T2) constr(T3)
constr(T4)
 constr(T5) constr(T6) :=
  ∃ T1 T2 T3 T4 T5 T6; auto_star.
```

Tactics to Sort Out the Proof Context

Hiding Hypotheses

```
(* Implementation *)

Definition ltac_something (P:Type) (e:P) := e.

Notation "'Something'" :=
  (@ltac_something _ _).

Lemma ltac_something_eq : ∀ (e:Type),
  e = (@ltac_something _ e).
Proof using. auto. Qed.

Lemma ltac_something_hide : ∀ (e:Type),
  e → (@ltac_something _ e).
Proof using. auto. Qed.
```

```
Lemma ltac_something_show : ∀ (e:Type),
  (@ltac_something _ e) → e.
Proof using. auto. Qed.
```

`hide_def x` and `show_def x` can be used to hide/show the body of the definition `x`.

```
Tactic Notation "hide_def" hyp(x) :=
  let x' := constr:(x) in
  let T := eval unfold x in x' in
  change T with (@ltac_something _ T) in x.

Tactic Notation "show_def" hyp(x) :=
  let x' := constr:(x) in
  let U := eval unfold x in x' in
  match U with @ltac_something _ ?T ⇒
    change U with T in x end.
```

`show_def` unfolds `Something` in the goal

```
Tactic Notation "show_def" :=
  unfold ltac_something.
Tactic Notation "show_def" "in" hyp(H) :=
  unfold ltac_something in H.
Tactic Notation "show_def" "in" "*" :=
  unfold ltac_something in *.
```

`hide_defs` and `show_defs` applies to all definitions

```
Tactic Notation "hide_defs" :=
  repeat match goal with H := ?T |- _ ⇒
    match T with
    | @ltac_something _ _ ⇒ fail 1
    | _ ⇒ change T with (@ltac_something _ T) in H
    end
  end.

Tactic Notation "show_defs" :=
  repeat match goal with H := (@ltac_something _ ?T) |- _ ⇒
    change (@ltac_something _ T) with T in H end.
```

`hide_hyp H` replaces the type of `H` with the notation `Something` and `show_hyp H` reveals the type of the hypothesis. Note that the hidden type of `H` remains convertible the real type of `H`.

```
Tactic Notation "show_hyp" hyp(H) :=
  apply ltac_something_show in H.

Tactic Notation "hide_hyp" hyp(H) :=
  apply ltac_something_hide in H.
```

`hide_hyps` and `show_hyps` can be used to hide/show all hypotheses of type `Prop`.

```
Tactic Notation "show_hyps" :=
  repeat match goal with
    H: @ltac_something _ _ |- _ ⇒ show_hyp H end.

Tactic Notation "hide_hyps" :=
  repeat match goal with H: ?T |- _ ⇒
    match type of T with
```

```
      | Prop ⇒
        match T with
        | @ltac_something _ _ ⇒ fail 2
        | _ ⇒ hide_hyp H
        end
      | _ ⇒ fail 1
      end
    end.
```

`hide H` and `show H` automatically select between `hide_hyp` or `hide_def`, and `show_hyp` or `show_def`. Similarly `hide_all` and `show_all` apply to all.

```
Tactic Notation "hide" hyp(H) :=
  first [hide_def H | hide_hyp H].

Tactic Notation "show" hyp(H) :=
  first [show_def H | show_hyp H].

Tactic Notation "hide_all" :=
  hide_hyps; hide_defs.

Tactic Notation "show_all" :=
  unfold ltac_something in *.
```

`hide_term E` can be used to hide a term from the goal. `show_term` or `show_term E` can be used to reveal it. `hide_term E in H` can be used to specify an hypothesis.

```
Tactic Notation "hide_term" constr(E) :=
  change E with (@ltac_something _ E).
Tactic Notation "show_term" constr(E) :=
  change (@ltac_something _ E) with E.
Tactic Notation "show_term" :=
  unfold ltac_something.

Tactic Notation "hide_term" constr(E) "in" hyp(H) :=
  change E with (@ltac_something _ E) in H.
Tactic Notation "show_term" constr(E) "in" hyp(H) :=
  change (@ltac_something _ E) with E in H.
Tactic Notation "show_term" "in" hyp(H) :=
  unfold ltac_something in H.
```

`show_unfold R` unfolds the definition of `R` and reveals the hidden definition of R. — todo:test, and implement using unfold simply

```
    (* todo: change "unfolds" *)

Tactic Notation "show_unfold" constr(R1) :=
  unfold R1; show_def.
Tactic Notation "show_unfold" constr(R1) "," constr(R2) :=
  unfold R1, R2; show_def.
```

Sorting Hypotheses

`sort` sorts out hypotheses from the context by moving all the propositions (hypotheses of type Prop) to the bottom of the context.

```
Ltac sort_tactic :=
  try match goal with H: ?T |- _ ⇒
```

```
    match type of T with Prop ⇒
      generalizes H; (try sort_tactic); intro
    end end.

  Tactic Notation "sort" :=
    sort_tactic.
```

Clearing Hypotheses

`clears X1 ... XN` is a variation on `clear` which clears the variables `X1..XN` as well as all the hypotheses which depend on them. Contrary to `clear`, it never fails.

```
  Tactic Notation "clears" ident(X1) :=
    let rec doit _ :=
    match goal with
    | H:context[X1] |- _ ⇒ clear H; try (doit tt)
    | _ ⇒ clear X1
    end in doit tt.
  Tactic Notation "clears" ident(X1) ident(X2) :=
    clears X1; clears X2.
  Tactic Notation "clears" ident(X1) ident(X2) ident(X3) :=
    clears X1; clears X2; clears X3.
  Tactic Notation "clears" ident(X1) ident(X2) ident(X3) ident(X4) :=
    clears X1; clears X2; clears X3; clears X4.
  Tactic Notation "clears" ident(X1) ident(X2) ident(X3) ident(X4)
   ident(X5) :=
    clears X1; clears X2; clears X3; clears X4; clears X5.
  Tactic Notation "clears" ident(X1) ident(X2) ident(X3) ident(X4)
   ident(X5) ident(X6) :=
    clears X1; clears X2; clears X3; clears X4; clears X5; clears X6.
```

`clears` (without any argument) clears all the unused variables from the context. In other words, it removes any variable which is not a proposition (i.e., not of type Prop) and which does not appear in another hypothesis nor in the goal.

```
    (* todo: rename to clears_var ? *)

  Ltac clears_tactic :=
    match goal with H: ?T |- _ ⇒
    match type of T with
    | Prop ⇒ generalizes H; (try clears_tactic); intro
    | ?TT ⇒ clear H; (try clears_tactic)
    | ?TT ⇒ generalizes H; (try clears_tactic); intro
    end end.

  Tactic Notation "clears" :=
    clears_tactic.
```

`clears_all` clears all the hypotheses from the context that can be cleared. It leaves only the hypotheses that are mentioned in the goal.

```
  Ltac clears_or_generalizes_all_core :=
    repeat match goal with H: _ |- _ ⇒
            first [ clear H | generalizes H] end.
```

```
Tactic Notation "clears_all" :=
  generalize ltac_mark;
  clears_or_generalizes_all_core;
  intro_until_mark.
```

`clears_but H1 H2 .. HN` clears all hypotheses except the one that are mentioned and those that cannot be cleared.

```
Ltac clears_but_core cont :=
  generalize ltac_mark;
  cont tt;
  clears_or_generalizes_all_core;
  intro_until_mark.

Tactic Notation "clears_but" :=
  clears_but_core ltac:(fun _ ⇒ idtac).
Tactic Notation "clears_but" ident(H1) :=
  clears_but_core ltac:(fun _ ⇒ gen H1).
Tactic Notation "clears_but" ident(H1) ident(H2) :=
  clears_but_core ltac:(fun _ ⇒ gen H1 H2).
Tactic Notation "clears_but" ident(H1) ident(H2) ident(H3) :=
  clears_but_core ltac:(fun _ ⇒ gen H1 H2 H3).
Tactic Notation "clears_but" ident(H1) ident(H2) ident(H3) ident(H4)
:=
  clears_but_core ltac:(fun _ ⇒ gen H1 H2 H3 H4).
Tactic Notation "clears_but" ident(H1) ident(H2) ident(H3) ident(H4)
ident(H5) :=
  clears_but_core ltac:(fun _ ⇒ gen H1 H2 H3 H4 H5).

Lemma demo_clears_all_and_clears_but :
  ∀ x y:nat, y < 2 → x = x → x ≥ 2 → x < 3 → True.
Proof using.
  introv M1 M2 M3. dup 6.
  (* clears_all clears all hypotheses. *)
  clears_all. auto.
  (* clears_but H clears all but H *)
  clears_but M3. auto.
  clears_but y. auto.
  clears_but x. auto.
  clears_but M2 M3. auto.
  clears_but x y. auto.
Qed.
```

`clears_last` clears the last hypothesis in the context. `clears_last N` clears the last `N` hypotheses in the context.

```
Tactic Notation "clears_last" :=
  match goal with H: ?T |- _ ⇒ clear H end.

Ltac clears_last_base N :=
  match nat_from_number N with
  | 0 ⇒ idtac
  | S ?p ⇒ clears_last; clears_last_base p
  end.
```

```
Tactic Notation "clears_last" constr(N) :=
  clears_last_base N.
```

Tactics for Development Purposes

Skipping Subgoals

DEPRECATED: the new "admit" tactics now works fine.

The `skip` tactic can be used at any time to admit the current goal. Using `skip` is much more efficient than using the `Focus` top-level command to reach a particular subgoal.

There are two possible implementations of `skip`. The first one relies on the use of an existential variable. The second one relies on an axiom of type `False`. Remark that the builtin tactic `admit` is not applicable if the current goal contains uninstantiated variables.

The advantage of the first technique is that a proof using `skip` must end with `Admitted`, since `Qed` will be rejected with the message "`uninstantiated existential variables`". It is thereafter clear that the development is incomplete.

The advantage of the second technique is exactly the converse: one may conclude the proof using `Qed`, and thus one saves the pain from renaming `Qed` into `Admitted` and vice-versa all the time. Note however, that it is still necessary to instantiate all the existential variables introduced by other tactics in order for `Qed` to be accepted.

The two implementation are provided, so that you can select the one that suits you best. By default `skip'` uses the first implementation, and `skip` uses the second implementation.

```
Ltac skip_with_existential :=
  match goal with |- ?G ⇒
    let H := fresh in evar(H:G); eexact H end.

(* TO BE DEPRECATED: *)
Parameter skip_axiom : False.
  (* To obtain a safe development, change to skip_axiom : True *)
Ltac skip_with_axiom :=
  elimtype False; apply skip_axiom.

Tactic Notation "skip" :=
   skip_with_axiom.
Tactic Notation "skip'" :=
   skip_with_existential.

(* SF DOES NOT NEED THIS
(* For backward compatibility *)
Tactic Notation "admit" :=
  skip.
*)
```

`demo` is like `admit` but it documents the fact that admit is intended

```
Tactic Notation "demo" :=
  skip.
```

`skip H: T` adds an assumption named `H` of type `T` to the current context, blindly assuming that it is true. `skip: T` and `skip H_asserts: T` and `skip_asserts: T` are other possible syntax. Note that H may be an intro pattern. The syntax `skip H1 .. HN: T` can be used when `T` is a conjunction of `N` items.

```
Tactic Notation "skip" simple_intropattern(I) ":" constr(T) :=
  asserts I: T; [ skip | ].
Tactic Notation "skip" ":" constr(T) :=
  let H := fresh in skip H: T.
Tactic Notation "skip" "¬" ":" constr(T) :=
  skip: T; auto_tilde.
Tactic Notation "skip" "*" ":" constr(T) :=
  skip: T; auto_star.

Tactic Notation "skip" simple_intropattern(I1)
 simple_intropattern(I2) ":" constr(T) :=
  skip [I1 I2]: T.
Tactic Notation "skip" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3) ":" constr(T) :=
  skip [I1 [I2 I3]]: T.
Tactic Notation "skip" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3)
 simple_intropattern(I4) ":" constr(T) :=
  skip [I1 [I2 [I3 I4]]]: T.
Tactic Notation "skip" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3)
 simple_intropattern(I4) simple_intropattern(I5) ":" constr(T) :=
  skip [I1 [I2 [I3 [I4 I5]]]]: T.
Tactic Notation "skip" simple_intropattern(I1)
 simple_intropattern(I2) simple_intropattern(I3)
 simple_intropattern(I4) simple_intropattern(I5)
 simple_intropattern(I6) ":" constr(T) :=
  skip [I1 [I2 [I3 [I4 [I5 I6]]]]]: T.

Tactic Notation "skip_asserts" simple_intropattern(I) ":" constr(T)
:=
  skip I: T.
Tactic Notation "skip_asserts" ":" constr(T) :=
  skip: T.
```

`skip_cuts T` simply replaces the current goal with `T`.

```
Tactic Notation "skip_cuts" constr(T) :=
  cuts: T; [ skip | ].
```

`skip_goal H` applies to any goal. It simply assumes the current goal to be true. The assumption is named "H". It is useful to set up proof by induction or coinduction. Syntax `skip_goal` is also accepted.

```
Tactic Notation "skip_goal" ident(H) :=
  match goal with |- ?G ⇒ skip H: G end.

Tactic Notation "skip_goal" :=
  let IH := fresh "IH" in skip_goal IH.
```

`skip_rewrite T` can be applied when `T` is an equality. It blindly assumes this equality to be true, and rewrite it in the goal.

```
Tactic Notation "skip_rewrite" constr(T) :=
  let M := fresh in skip_asserts M: T; rewrite M; clear M.
```

`skip_rewrite T in H` is similar as `rewrite_skip`, except that it rewrites in hypothesis `H`.

```
Tactic Notation "skip_rewrite" constr(T) "in" hyp(H) :=
  let M := fresh in skip_asserts M: T; rewrite M in H; clear M.
```

`skip_rewrites_all T` is similar as `rewrite_skip`, except that it rewrites everywhere (goal and all hypotheses).

```
Tactic Notation "skip_rewrite_all" constr(T) :=
  let M := fresh in skip_asserts M: T; rewrite_all M; clear M.
```

`skip_induction E` applies to any goal. It simply assumes the current goal to be true (the assumption is named "IH" by default), and call `destruct E` instead of `induction E`. It is useful to try and set up a proof by induction first, and fix the applications of the induction hypotheses during a second pass on the Proof using.

```
(* TODO: deprecated *)

Tactic Notation "skip_induction" constr(E) :=
  let IH := fresh "IH" in skip_goal IH; destruct E.

Tactic Notation "skip_induction" constr(E) "as"
simple_intropattern(I) :=
  let IH := fresh "IH" in skip_goal IH; destruct E as I.
```

Compatibility with Standard Library

The module `Program` contains definitions that conflict with the current module. If you import `Program`, either directly or indirectly (e.g., through `Setoid` or `ZArith`), you will need to import the compability definitions through the top-level command: `Import LibTacticsCompatibility`.

```
Module LibTacticsCompatibility.
  Tactic Notation "apply" "*" constr(H) :=
    sapply H; auto_star.
  Tactic Notation "subst" "*" :=
    subst; auto_star.
End LibTacticsCompatibility.

Open Scope nat_scope.
```