Mathlib tactics

In addition to core tactics, mathlib provides a number of specific interactive tactics and commands. Here we document the mostly commonly used ones.

tfae

The `tfae` tactic suite is a set of tactics that help with proving that certain propositions are equivalent. In `data/list/basic.lean` there is a section devoted to propositions of the form

```
tfae [p1, p2, ..., pn]
```

where `p1`, `p2`, through, `pn` are terms of type `Prop`. This proposition asserts that all the `pi` are pairwise equivalent. There are results that allow to extract the equivalence of two propositions `pi` and `pj`.

To prove a goal of the form `tfae [p1, p2, ..., pn]`, there are two tactics. The first tactic is `tfae_have`. As an argument it takes an expression of the form `i arrow j`, where `i` and `j` are two positive natural numbers, and `arrow` is an arrow such as `→`, `->`, `←`, `<-`, `↔`, or `<->`. The tactic `tfae_have : i arrow j` sets up a subgoal in which the user has to prove the equivalence (or implication) of `pi` and `pj`.

The remaining tactic, `tfae_finish`, is a finishing tactic. It collects all implications and equivalences from the local context and computes their transitive closure to close the main goal.

`tfae_have` and `tfae_finish` can be used together in a proof as follows:

```
example (a b c d : Prop) : tfae [a,b,c,d] :=
begin
  tfae_have : 3 → 1,
  { /- prove c → a -/ },
  tfae_have : 2 → 3,
  { /- prove b → c -/ },
  tfae_have : 2 ← 1,
  { /- prove a → b -/ },
  tfae_have : 4 ↔ 2,
  { /- prove d ↔ b -/ },
    -- a b c d : Prop,
    -- tfae_3_to_1 : c → a,
    -- tfae_2_to_3 : b → c,
    -- tfae_1_to_2 : a → b,
    -- tfae_4_iff_2 : d ↔ b
    -- ⊢ tfae [a, b, c, d]
  tfae_finish,
end
```

rcases

The `rcases` tactic is the same as `cases`, but with more flexibility in the `with` pattern syntax to allow for recursive case splitting. The pattern syntax uses the following recursive grammar:

```
patt ::= (patt_list "|")* patt_list
patt_list ::= id | "_" | "⟨" (patt ",")* patt "⟩"
```

A pattern like `⟨a, b, c⟩ | ⟨d, e⟩` will do a split over the inductive datatype, naming the first three parameters of the first constructor as `a,b,c` and the first two of the second constructor `d,e`. If the list is not as long as the number of arguments to the constructor or the number of constructors, the remaining variables will be automatically named. If there are nested brackets such as `⟨⟨a⟩, b | c⟩ | d` then these will cause more case splits as necessary. If there are too many arguments, such as `⟨a, b, c⟩` for splitting on `∃ x, ∃ y, p x`, then it will be treated as `⟨a, ⟨b, c⟩⟩`, splitting the last parameter as necessary.

`rcases` also has special support for quotient types: quotient induction into Prop works like matching on the constructor `quot.mk`.

`rcases? e` will perform case splits on `e` in the same way as `rcases e`, but rather than accepting a pattern, it does a maximal cases and prints the pattern that would produce this case splitting. The default maximum depth is 5, but this can be modified with `rcases? e : n`.

rintro

The `rintro` tactic is a combination of the `intros` tactic with `rcases` to allow for destructuring patterns while introducing variables. See `rcases` for a description of supported patterns. For example, `rintros (a | ⟨b, c⟩) ⟨d, e⟩` will introduce two variables, and then do case splits on both of them producing two subgoals, one with variables `a d e` and the other with `b c d e`.

`rintro?` will introduce and case split on variables in the same way as `rintro`, but will also print the `rintro` invocation that would have the same result. Like `rcases?`, `rintro? : n` allows for modifying the depth of splitting; the default is 5.

simpa

This is a "finishing" tactic modification of `simp`. It has two forms.

- `simpa [rules, ...] using e` will simplify the goal and the type of `e` using `rules`, then try to close the goal using `e`.

  Simplifying the type of `e` makes it more likely to match the goal (which has also been simplified). This construction also tends to be more robust under changes to the simp lemma set.

- `simpa [rules, ...]` will simplify the goal and the type of a hypothesis `this` if present, then try to close the goal using the `assumption` tactic.

replace

Acts like `have`, but removes a hypothesis with the same name as this one. For example if the state is `h : p ⊢ goal` and `f : p → q`, then after `replace h := f h` the goal will be `h : q ⊢ goal`, where `have h :=`

`f h` would result in the state `h : p, h : q ⊢ goal`. This can be used to simulate the `specialize` and `apply at` tactics of Coq.

elide/unelide

The `elide n (at ...)` tactic hides all subterms of the target goal or hypotheses beyond depth `n` by replacing them with `hidden`, which is a variant on the identity function. (Tactics should still mostly be able to see through the abbreviation, but if you want to unhide the term you can use `unelide`.)

The `unelide (at ...)` tactic removes all `hidden` subterms in the target types (usually added by `elide`).

finish/clarify/safe

These tactics do straightforward things: they call the simplifier, split conjunctive assumptions, eliminate existential quantifiers on the left, and look for contradictions. They rely on ematching and congruence closure to try to finish off a goal at the end.

The procedures *do* split on disjunctions and recreate the smt state for each terminal call, so they are only meant to be used on small, straightforward problems.

- finish: solves the goal or fails
- clarify: makes as much progress as possible while not leaving more than one goal
- safe: splits freely, finishes off whatever subgoals it can, and leaves the rest

All accept an optional list of simplifier rules, typically definitions that should be expanded. (The equations and identities should not refer to the local context.)

ring

Evaluate expressions in the language of (semi-)rings. Based on Proving Equalities in a Commutative Ring Done Right in Coq by Benjamin Grégoire and Assia Mahboubi.

congr'

Same as the `congr` tactic, but takes an optional argument which gives the depth of recursive applications. This is useful when `congr` is too aggressive in breaking down the goal. For example, given `⊢ f (g (x + y)) = f (g (y + x))`, `congr'` produces the goals `⊢ x = y` and `⊢ y = x`, while `congr' 2` produces the intended `⊢ x + y = y + x`. If, at any point, a subgoal matches a hypothesis then the subgoal will be closed.

convert

The `exact e` and `refine e` tactics require a term `e` whose type is definitionally equal to the goal. `convert e` is similar to `refine e`, but the type of `e` is not required to exactly match the goal. Instead, new goals are created for differences between the type of `e` and the goal. For example, in the proof state

```
n : ℕ,
e : prime (2 * n + 1)
⊢ prime (n + n + 1)
```

the tactic `convert e` will change the goal to

```
⊢ n + n = 2 * n
```

In this example, the new goal can be solved using `ring`.

The syntax `convert ← e` will reverse the direction of the new goals (producing `⊢ 2 * n = n + n` in this example).

Internally, `convert e` works by creating a new goal asserting that the goal equals the type of `e`, then simplifying it using `congr'`. The syntax `convert e using n` can be used to control the depth of matching (like `congr' n`). In the example, `convert e using 1` would produce a new goal `⊢ n + n + 1 = 2 * n + 1`.

unfold_coes

Unfold coercion-related definitions

Instance cache tactics

- `resetI`: Reset the instance cache. This allows any new instances added to the context to be used in typeclass inference.
- `unfreezeI`: Unfreeze local instances, which allows us to revert instances in the context
- `introI`/`introsI`: Like `intro`/`intros`, but uses the introduced variable in typeclass inference.
- `haveI`/`letI`: Used to add typeclasses to the context so that they can be used in typeclass inference. The syntax is the same as `have`/`letI`, but the proof-omitted version of `have` is not supported (for this one must write `have : t, { <proof> }, resetI, <proof>`).
- `exactI`: Like `exact`, but uses all variables in the context for typeclass inference.

find

The `find` command from `tactic.find` allows to find lemmas using pattern matching. For instance:

```
import tactic.find

#find _ + _ = _ + _
#find (_ : ℕ) + _ = _ + _
```

solve_by_elim

The tactic `solve_by_elim` repeatedly applies assumptions to the current goal, and succeeds if this eventually discharges the main goal.

```
solve_by_elim { discharger := `[cc] }
```

also attempts to discharge the goal using congruence closure before each round of applying assumptions.

`solve_by_elim*` tries to solve all goals together, using backtracking if a solution for one goal make other goals impossible.

By default `solve_by_elim` also applies `congr_fun` and `congr_arg` against the goal.

The assumptions can be modified with similar syntax as for `simp`:

- `solve_by_elim [h₁, h₂, ..., hᵣ]` also applies the named lemmas (or all lemmas tagged with the named attributes).
- `solve_by_elim only [h₁, h₂, ..., hᵣ]` does not include the local context, `congr_fun`, or `congr_arg` unless they are explicitly included.
- `solve_by_elim [-id_1, ... -id_n]` uses the default assumptions, removing the specified ones.

ext1 / ext

- `ext1 id` selects and apply one extensionality lemma (with attribute `extensionality`), using `id`, if provided, to name a local constant introduced by the lemma. If `id` is omitted, the local constant is named automatically, as per `intro`.

- `ext` applies as many extensionality lemmas as possible;

- `ext ids`, with `ids` a list of identifiers, finds extentionality and applies them until it runs out of identifiers in `ids` to name the local constants.

When trying to prove:

```
α β : Type,
f g : α → set β
⊢ f = g
```

applying `ext x y` yields:

```
α β : Type,
f g : α → set β,
x : α,
y : β
⊢ y ∈ f x ↔ y ∈ g x
```

by applying functional extensionality and set extensionality.

A maximum depth can be provided with `ext x y z : 3`.

The `extensionality` attribute

Tag lemmas of the form:

```
@[extensionality]
lemma my_collection.ext (a b : my_collection)
  (h : ∀ x, a.lookup x = b.lookup y) :
  a = b := ...
```

The attribute indexes extensionality lemma using the type of the objects (i.e. `my_collection`) which it gets from the statement of the lemma. In some cases, the same lemma can be used to state the extensionality of multiple types that are definitionally equivalent.

```
attribute [extensionality [(→),thunk,stream]] funext
```

Those parameters are cumulative. The following are equivalent:

```
attribute [extensionality [(→),thunk]] funext
attribute [extensionality [stream]] funext
```

and

```
attribute [extensionality [(→),thunk,stream]] funext
```

One removes type names from the list for one lemma with:

```
attribute [extensionality [-stream,-thunk]] funext
```

Finally, the following:

```
@[extensionality]
lemma my_collection.ext (a b : my_collection)
  (h : ∀ x, a.lookup x = b.lookup y) :
  a = b := ...
```

is equivalent to

```
@[extensionality *]
lemma my_collection.ext (a b : my_collection)
  (h : ∀ x, a.lookup x = b.lookup y) :
  a = b := ...
```

The `*` parameter indicates to simply infer the type from the lemma's statement.

This allows us specify type synonyms along with the type that referred to in the lemma statement.

```
@[extensionality [*,my_type_synonym]]
lemma my_collection.ext (a b : my_collection)
  (h : ∀ x, a.lookup x = b.lookup y) :
  a = b := ...
```

refine_struct

`refine_struct { .. }` acts like `refine` but works only with structure instance literals. It creates a goal for each missing field and tags it with the name of the field so that `have_field` can be used to generically refer to the field currently being refined.

As an example, we can use `refine_struct` to automate the construction semigroup instances:

```
refine_struct ( { .. } : semigroup α ),
-- case semigroup, mul
-- α : Type u,
-- ⊢ α → α → α

-- case semigroup, mul_assoc
-- α : Type u,
-- ⊢ ∀ (a b c : α), a * b * c = a * (b * c)
```

`have_field`, used after `refine_struct _` poses `field` as a local constant with the type of the field of the current goal:

```
refine_struct ({ .. } : semigroup α),
{ have_field, ... },
{ have_field, ... },
```

behaves like

```
refine_struct ({ .. } : semigroup α),
{ have field := @semigroup.mul, ... },
{ have field := @semigroup.mul_assoc, ... },
```

apply_rules

`apply_rules hs n` applies the list of lemmas `hs` and `assumption` on the first goal and the resulting subgoals, iteratively, at most `n` times. `n` is optional, equal to 50 by default. `hs` can contain user attributes: in this case all theorems with this attribute are added to the list of rules.

For instance:

```
@[user_attribute]
meta def mono_rules : user_attribute :=
{ name := `mono_rules,
  descr := "lemmas usable to prove monotonicity" }

attribute [mono_rules] add_le_add mul_le_mul_of_nonneg_right

lemma my_test {a b c d e : real} (h1 : a ≤ b) (h2 : c ≤ d) (h3 : 0 ≤ e) :
a + c * e + a + c + 0 ≤ b + d * e + b + d + e :=
-- any of the following lines solve the goal:
add_le_add (add_le_add (add_le_add (add_le_add h1
(mul_le_mul_of_nonneg_right h2 h3)) h1 ) h2) h3
by apply_rules [add_le_add, mul_le_mul_of_nonneg_right]
by apply_rules [mono_rules]
by apply_rules mono_rules
```

h_generalize

`h_generalize Hx : e == x` matches on `cast _ e` in the goal and replaces it with `x`. It also adds `Hx : e == x` as an assumption. If `cast _ e` appears multiple times (not necessarily with the same proof), they are all replaced by `x`. `cast` `eq.mp`, `eq.mpr`, `eq.subst`, `eq.substr`, `eq.rec` and `eq.rec_on` are all treated as casts.

- `h_generalize Hx : e == x with h` adds hypothesis `α = β` with `e : α, x : β`;
- `h_generalize Hx : e == x with _` chooses automatically chooses the name of assumption `α = β`;
- `h_generalize! Hx : e == x` reverts `Hx`;
- when `Hx` is omitted, assumption `Hx : e == x` is not added.

pi_instance

`pi_instance` constructs an instance of `my_class (Π i : I, f i)` where we know `Π i, my_class (f i)`. If an order relation is required, it defaults to `pi.partial_order`. Any field of the instance that `pi_instance` cannot construct is left untouched and generated as a new goal.

assoc_rewrite

`assoc_rewrite [h₀, ← h₁] at ⊢ h₂` behaves like `rewrite [h₀, ← h₁] at ⊢ h₂` with the exception that associativity is used implicitly to make rewriting possible.

restate_axiom

`restate_axiom` makes a new copy of a structure field, first definitionally simplifying the type. This is useful to remove `auto_param` or `opt_param` from the statement.

As an example, we have:

```
structure A :=
(x : ℕ)
(a' : x = 1 . skip)

example (z : A) : z.x = 1 := by rw A.a' -- rewrite tactic failed, lemma is
not an equality nor a iff

restate_axiom A.a'
example (z : A) : z.x = 1 := by rw A.a
```

By default, `restate_axiom` names the new lemma by removing a trailing `'`, or otherwise appending `_lemma` if there is no trailing `'`. You can also give `restate_axiom` a second argument to specify the new name, as in

```
restate_axiom A.a f
example (z : A) : z.x = 1 := by rw A.f
```

def_replacer

`def_replacer foo` sets up a stub definition `foo : tactic unit`, which can effectively be defined and re-defined later, by tagging definitions with `@[foo]`.

- `@[foo] meta def foo_1 : tactic unit := ...` replaces the current definition of `foo`.
- `@[foo] meta def foo_2 (old : tactic unit) : tactic unit := ...` replaces the current definition of `foo`, and provides access to the previous definition via `old`. (The argument can also be an `option (tactic unit)`, which is provided as `none` if this is the first definition tagged with `@[foo]` since `def_replacer` was invoked.)

`def_replacer foo : α → β → tactic γ` allows the specification of a replacer with custom input and output types. In this case all subsequent redefinitions must have the same type, or the type `α → β → tactic γ → tactic γ` or `α → β → option (tactic γ) → tactic γ` analogously to the previous cases.

tidy

`tidy` attempts to use a variety of conservative tactics to solve the goals. In particular, `tidy` uses the `chain` tactic to repeatedly apply a list of tactics to the goal and recursively on new goals, until no tactic makes further progress.

`tidy` can report the tactic script it found using `tidy?`. As an example

```
example : ∀ x : unit, x = unit.star :=
begin
  tidy? -- Prints the trace message: "intros x, exact dec_trivial"
end
```

The default list of tactics can be found by looking up the definition of `default_tidy_tactics`.

This list can be overriden using `tidy { tactics := ... }`. (The list must be a list of `tactic string`, so that `tidy?` can report a usable tactic script.)

linarith

`linarith` attempts to find a contradiction between hypotheses that are linear (in)equalities. Equivalently, it can prove a linear inequality by assuming its negation and proving `false`.

In theory, `linarith` should prove any goal that is true in the theory of linear arithmetic over the rationals. While there is some special handling for non-dense orders like `nat` and `int`, this tactic is not complete for these theories and will not prove every true goal.

An example:

```
example (x y z : ℚ) (h1 : 2*x  < 3*y) (h2 : -4*x + 2*z < 0)
        (h3 : 12*y - 4* z < 0)  : false :=
by linarith
```

`linarith` will use all appropriate hypotheses and the negation of the goal, if applicable.

`linarith h1 h2 h3` will ohly use the local hypotheses `h1`, `h2`, `h3`.

`linarith using [t1, t2, t3]` will add `t1`, `t2`, `t3` to the local context and then run `linarith`.

`linarith {discharger := tac, restrict_type := tp, exfalso := ff}` takes a config object with three optional arguments.

- `discharger` specifies a tactic to be used for reducing an algebraic equation in the proof stage. The default is `ring`. Other options currently include `ring SOP` or `simp` for basic problems.
- `restrict_type` will only use hypotheses that are inequalities over `tp`. This is useful if you have e.g. both integer and rational valued inequalities in the local context, which can sometimes confuse the tactic.
- If `exfalso` is false, `linarith` will fail when the goal is neither an inequality nor `false`. (True by default.)

choose

`choose a b h using hyp` takes an hypothesis `hyp` of the form `∀ (x : X) (y : Y), ∃ (a : A) (b : B), P x y a b` for some `P : X → Y → A → B → Prop` and outputs into context a function `a : X → Y → A`, `b : X → Y → B` and a proposition `h` stating `∀ (x : X) (y : Y), P x y (a x y) (b x y)`. It presumably also works with dependent versions.

Example:

```
example (h : ∀n m : ℕ, ∃i j, m = n + i ∨ m + j = n) : true :=
begin
  choose i j h using h,
  guard_hyp i := ℕ → ℕ → ℕ,
  guard_hyp j := ℕ → ℕ → ℕ,
  guard_hyp h := ∀ (n m : ℕ), m = n + i n m ∨ m + j n m = n,
```

```
  trivial
end
```

squeeze_simp / squeeze_simpa

`squeeze_simp` and `squeeze_simpa` perform the same task with the difference that `squeeze_simp` relates to `simp` while `squeeze_simpa` relates to `simpa`. The following applies to both `squeeze_simp` and `squeeze_simpa`.

`squeeze_simp` behaves like `simp` (including all its arguments) and prints a `simp only` invokation to skip the search through the `simp` lemma list.

For instance, the following is easily solved with `simp`:

```
example : 0 + 1 = 1 + 0 := by simp
```

To guide the proof search and speed it up, we may replace `simp` with `squeeze_simp`:

```
example : 0 + 1 = 1 + 0 := by squeeze_simp
-- prints: simp only [add_zero, eq_self_iff_true, zero_add]
```

`squeeze_simp` suggests a replacement which we can use instead of `squeeze_simp`.

```
example : 0 + 1 = 1 + 0 := by simp only [add_zero, eq_self_iff_true,
zero_add]
```

`squeeze_simp only` prints nothing as it already skips the `simp` list.

This tactic is useful for speeding up the compilation of a complete file. Steps:

1. search and replace `simp` with `squeeze_simp` (the space helps avoid the replacement of `simp` in `@[simp]`) throughout the file.
2. Starting at the beginning of the file, go to each printout in turn, copy the suggestion in place of `squeeze_simp`.
3. after all the suggestions were applied, search and replace `squeeze_simp` with `simp` to remove the occurrences of `squeeze_simp` that did not produce a suggestion.

Known limitation(s):

- in cases where `squeeze_simp` is used after a `;` (e.g. `cases x; squeeze_simp`), `squeeze_simp` will produce as many suggestions as the number of goals it is applied to. It is likely that none of the suggestion is a good replacement but they can all be combined by concatenating their list of lemmas.

fin_cases

`fin_cases h` performs case analysis on a hypothesis of the form

1. `h : A`, where `[fintype A]` is available, or
2. `h ∈ A`, where `A : finset X`, `A : multiset X` or `A : list X`.

`fin_cases *` performs case analysis on all suitable hypotheses.

As an example, in

```
example (f : ℕ → Prop) (p : fin 3) (h0 : f 0) (h1 : f 1) (h2 : f 2) : f
p.val :=
begin
  fin_cases p; simp,
  all_goals { assumption }
end
```

after `fin_cases p; simp`, there are three goals, `f 0`, `f 1`, and `f 2`.

conv

The `conv` tactic is built-in to lean. Currently mathlib additionally provides

- `erw`,
- `ring` and `ring2`, and
- `norm_num` inside `conv` blocks. Also, as a shorthand `conv_lhs` and `conv_rhs` are provided, so that

```
example : 0 + 0 = 0 :=
begin
  conv_lhs {simp}
end
```

just means

```
example : 0 + 0 = 0 :=
begin
  conv {to_lhs, simp}
end
```

and likewise for `to_rhs`.

mono

- `mono` applies a monotonicity rule.
- `mono*` applies monotonicity rules repetitively.
- `mono with x ≤ y` or `mono with [0 ≤ x,0 ≤ y]` creates an assertion for the listed propositions. Those help to select the right monotonicity rule.

- `mono left` or `mono right` is useful when proving strict orderings: for `x + y < w + z` could be broken down into either
  - left: `x ≤ w` and `y < z` or
  - right: `x < w` and `y ≤ z`

To use it, first import `tactic.monotonicity`.

Here is an example of mono:

```
example (x y z k : ℤ)
  (h : 3 ≤ (4 : ℤ))
  (h' : z ≤ y) :
  (k + 3 + x) - y ≤ (k + 4 + x) - z :=
begin
  mono, -- unfold `(-)`, apply add_le_add
  { -- ⊢ k + 3 + x ≤ k + 4 + x
    mono, -- apply add_le_add, refl
    -- ⊢ k + 3 ≤ k + 4
    mono },
  { -- ⊢ -y ≤ -z
    mono /- apply neg_le_neg -/ }
end
```

More succinctly, we can prove the same goal as:

```
example (x y z k : ℤ)
  (h : 3 ≤ (4 : ℤ))
  (h' : z ≤ y) :
  (k + 3 + x) - y ≤ (k + 4 + x) - z :=
by mono*
```

ac_mono

`ac_mono` reduces the `f x ⊑ f y`, for some relation `⊑` and a monotonic function `f` to `x < y`.

`ac_mono*` unwraps monotonic functions until it can't.

`ac_mono^k`, for some literal number `k` applies monotonicity `k` times.

`ac_mono h`, with `h` a hypothesis, unwraps monotonic functions and uses `h` to solve the remaining goal. Can be combined with `*` or `^k`: `ac_mono* h`

`ac_mono : p` asserts `p` and uses it to discharge the goal result unwrapping a series of monotonic functions. Can be combined with * or ^k: `ac_mono* : p`

In the case where `f` is an associative or commutative operator, `ac_mono` will consider any possible permutation of its arguments and use the one the minimizes the difference between the left-hand side and the right-hand side.

To use it, first import `tactic.monotonicity`.

`ac_mono` can be used as follows:

```
example (x y z k m n : ℕ)
  (h₀ : z ≥ 0)
  (h₁ : x ≤ y) :
  (m + x + n) * z + k ≤ z * (y + n + m) + k :=
begin
  ac_mono,
  -- ⊢ (m + x + n) * z ≤ z * (y + n + m)
  ac_mono,
  -- ⊢ m + x + n ≤ y + n + m
  ac_mono,
end
```

As with `mono*`, `ac_mono*` solves the goal in one go and so does `ac_mono* h₁`. The latter syntax becomes especially interesting in the following example:

```
example (x y z k m n : ℕ)
  (h₀ : z ≥ 0)
  (h₁ : m + x + n ≤ y + n + m) :
  (m + x + n) * z + k ≤ z * (y + n + m) + k :=
by ac_mono* h₁.
```

By giving `ac_mono` the assumption `h₁`, we are asking `ac_refl` to stop earlier than it would normally would.

use

Similar to `existsi`. `use x` will instantiate the first term of an `∃` or `Σ` goal with `x`. Unlike `existsi`, `x` is elaborated with respect to the expected type. Equivalent to `refine ⟨x, _⟩`.

`use` will alternatively take a list of terms `[x0, ..., xn]`.

Examples:

```
example (α : Type) : ∃ S : set α, S = S :=
by use ∅

example : ∃ x : ℤ, x = x :=
by use 42

example : ∃ a b c : ℤ, a + b + c = 6 :=
by use [1, 2, 3]

example : ∃ p : ℤ × ℤ, p.1 = 1 :=
by use ⟨1, 42⟩
```

clear_aux_decl

`clear_aux_decl` clears every `aux_decl` in the local context for the current goal. This includes the induction hypothesis when using the equation compiler and `_let_match` and `_fun_match`.

It is useful when using a tactic such as `finish`, `simp *` or `subst` that may use these auxiliary declarations, and produce an error saying the recursion is not well founded.

```
example (n m : ℕ) (h₁ : n = m) (h₂ : ∃ a : ℕ, a = n ∧ a = m) : 2 * m = 2 *
n :=
let ⟨a, ha⟩ := h₂ in
begin
  clear_aux_decl, -- subst will fail without this line
  subst h₁
end

example (x y : ℕ) (h₁ : ∃ n : ℕ, n * 1 = 2) (h₂ : 1 + 1 = 2 → x * 1 = y) :
x = y :=
let ⟨n, hn⟩ := h₁ in
begin
  clear_aux_decl, -- finish produces an error without this line
  finish
end
```

set

`set a := t with h` is a variant of `let a := t`. It adds the hypothesis `h : a = t` to the local context and replaces `t` with `a` everywhere it can.

`set a := t with ←h` will add `h : t = a` instead.

`set! a := t with h` does not do any replacing.

```
example (x : ℕ) (h : x = 3)  : x + x + x = 9 :=
begin
  set y := x with ←h_xy,
/-
x : ℕ,
y : ℕ := x,
h_xy : x = y,
h : y = 3
⊢ y + y + y = 9
-/
end
```