MORE PROOF TACTICS

Proof Tactics: Unfolding (dsimp)

Unfolding definitions
Informally:

```
def triple n = n + n + n

Theorem: if n + n = m
  then triple n = m + n

Proof:
  triple n
= n + n + n   (by def
               of triple)
= m + n       (by h)
```

Unfolding definitions
In Lean:

```
def triple (n : Nat) : Nat := n + n + n

theorem foo (n m : Nat)
    (h : n + n = m) :
    triple n = m + n := by

                  | |- triple n = m + n
dsimp [triple]    | |- n + n + n = m + n
rw [h]            | no goals
```

dsimp [f] ("definitionally simplify") unfolds the definition of `f`

- Informal analogue: "by definition of f"
- It also closes the goal by `rfl` when it can
- `rw [triple, h]` works too: `rw` can also unfold definitions
- Not sure what to unfold? `dsimp?` suggests a `dsimp only [...]` call

Proof Tactics: dsimp ... at h

Simplifying a hypothesis
Informally:

```
def square n = n * n

Theorem: if square n = 16
  then n * n = 16

Proof:
assume (1) square n = 16
now must prove: n * n = 16

(2) n * n = 16 (eval (1))

QED              (2 was all we need)
```

Simplifying a hypothesis
In Lean:

```
def square (n : Nat) : Nat := n * n

theorem foo (n : Nat)(h : square n = 16) :
    n * n = 16 := by

                        | h:square n = 16
                        | |- n * n = 16
dsimp [square] at h     | h:n * n = 16
                        | |- n * n = 16
exact h                 | no goals
```

at h points a tactic at hypothesis h instead of the goal

- `dsimp [f] at h`: “unfolding f in h, we know ...”
- `exact h`: “the goal is exactly h”

Proof Tactics: Nested Proofs (have)

Claims and sub-proofs
Informally:

```
Theorem:
  ((0 + n) + 0) * m = n * m

Proof:
  Claim h: (0 + n) + 0 = n
      (0 + n) + 0
    = n + 0    (zero_add)
    = n        (add_zero)
  Using the claim h:
    ((0 + n) + 0) * m
  = n * m      (by claim)
```

Claims and sub-proofs
In Lean:

```
theorem mult_zero_add' (n m : Nat) :
    ((zero + n) + zero) * m = n * m := by
  have h : (zero + n) + zero = n := by
    rw [zero_add, add_zero]
  -- now h is in the context
  rw [h]
```

have h : P := by … proves P in a nested sub-proof, then adds `h : P` to the context

- Informal analogue: "Claim: P. (proof of claim) Using the claim, …"
- For facts too small to deserve their own top-level theorem
- Indent the nested proof under the `have`

Proof Tactics: Search (? tactics)

Standard facts
Informally:

```
Theorem: n + m = m + n

Proof:
  n + m
= m + n   (by commutativity
           of addition)

  a standard fact: we
  look it up, not prove it
```

Standard facts
In Lean:

```
theorem foo (n m : Nat) :
    n + m = m + n := by
  exact?
  -- Try this: exact Nat.add_comm n m

  -- or instead:
  rw?
  -- Try this: rw [Nat.add_comm]

  -- for definitions, dsimp? suggests
  -- e.g. dsimp only [triple]
```

- `exact?` searches the library for a theorem that **closes** the goal
- `rw?` searches for theorems you could **rewrite** the goal with
- `dsimp?` simplifies, then tells you the `dsimp only [...]` call it used
- Click a suggestion to paste it in, replacing the `?` tactic

Template: Induction on lists

Informally:

Theorem: For all lists l, P(l).

Proof: By induction on lists l.

Case l = nil:

must prove: P(nil)

...

Case l = cons n l':

IH: P(l')

must prove: P(cons n l')

...

In Lean:

```
theorem foo (l : NList) :
    P l := by
  induction l with
  | nil =>
    -- |- P nil
    ...
  | cons n l' ih =>
    -- ih : P l'
    -- |- P (cons n l')
    ...
```

- `induction l with` gives one case per constructor of `NList`
- `| cons n l' ih =>` names the head `n`, the tail `l'` and the IH `ih`

Proof Tactics: induction on lists

Induction
Informally:

```
Theorem: for all lists l,
  append l nil = l

Proof: by induction on l.
Case l = nil:
  append nil nil = nil
                   (by eval)
Case l = cons n l':
  IH: append l' nil = l'
  append (cons n l') nil
= cons n (append l' nil)
                   (by def)
= cons n l'        (by IH)
```

Induction
In Lean:

```
theorem append_nil (l : NList) :
    append l nil = l := by
  induction l with
  | nil => rfl
  | cons n l' ih =>
    -- ih : append l' nil = l'
    dsimp [append]
    -- |- cons n (append l' nil)
    --       = cons n l'
    rw [ih]
```

- The IH **`ih`** is just a hypothesis: use it with **`rw [ih]`**
- Each case is an ordinary proof: **`rfl`**, **`dsimp`**, **`rw`**, ...

Induction with Library Facts

Induction
Informally:

```
Theorem:
  |l1 ++ l2| = |l1| + |l2|
(|l| = length, ++ = append)
Case l1 = nil:
  |nil ++ l2|
= |l2|                 (eval)
= 0 + |l2|       (zero_add)
= |nil| + |l2|        (eval)
Case l1 = cons n l':
  |cons n l' ++ l2|
= 1 + |l' ++ l2|      (eval)
= 1 + (|l'| + |l2|)    (IH)
= (1 + |l'|) + |l2| (assoc)
= |cons n l'| + |l2| (eval)
```

Induction
In Lean:

```
theorem append_length (l1 l2 : NList) :
    length (append l1 l2) =
    length l1 + length l2 := by
  induction l1 with
  | nil =>
    dsimp [append, length]
    rw [Nat.zero_add]
  | cons n l' ih =>
    dsimp [append, length]
    rw [ih]
    rw [Nat.add_assoc]
```

- Unfold with **`dsimp`**, use **`ih`**, finish with library facts (find them with **`rw?`**)

Lean Tactics and Informal Proof

In Lean	Informally
`rfl`	both sides are the same (by evaluation)
`rewrite [h]`	by h: replace the left side of h with its right
`rewrite [<- h]`	by h: replace the right side of h with its left
`rewrite [h] at j`	in hypothesis j, rewrite by h
`rw [h]`	let n and m be arbitrary
`intro n m`	assume P (call this assumption h)
`intro h`	this is exactly h
`exact h`	by the definition of f
`dsimp [f]`	unfolding f in h, we know ...
`dsimp [f] at h`	Claim: P. (proof of claim) ...
`have h : P`	by a standard fact (look it up!)
`exact? rw?`	(asks Lean which definitions to unfold)
`dsimp?`	by induction on l
`induction l with`	case l = cons n l', with IH: P(l')
`| cons n l' ih`	