8+9 = 17

Interactive Theorem Proving Assignment 3

Martin Skilleter (u6679334)

12 May 2019

I collaborated with the following people while completing this assignment: Isabel Longbottom (u6702144) and Aaron Bryce (u6677442).

Question 1

```
import tactic.interactive
import init.meta.interaction_monad
import data.real.basic
```

This is just list.foldl nat.max

```
open tactic

-- Find the maximum element of a list of ℕ. Used to work out at what point
-- we don't need to traverse the targets list any further.
def find_max : list ℕ → ℕ → ℕ
| [] so_far := so_far
| (h :: tl) so_far := if h > so_far then find_max tl h else find_max tl so_far
```

why? I thought we said 0-indexed.

```
-- Given a list of ℕ (the 'targets'), return the goals corresponding
-- to these indices. Indexed from 1 (not 0).
meta def find_wanted_goals : list ℕ → ℕ → ℕ → list expr → tactic (list expr)
| tgts crnt max gls := if crnt > max then return []
                                   else match gls with
                                   | [] := fail "No such goals!"
                                   | (g::gs) := if crnt ∈ tgts then
                                                do out ← find_wanted_goals
tgts (crnt+1) max gs,
                                                   return ([g]++out)
                                                else do out ←
find_wanted_goals tgts (crnt+1) max gs,
                                                   return out
                                   end
```

why is this in the tactic monad?

more arguments that don't change in recursive calls before the colon.

no need to get the value, then return it!

You can tell you're doing something lame here because this is a bad recursion!

```
-- A wrapper for find_wanted_goals. Only needs to be given the targets.
meta def find_goals : list ℕ → tactic unit
| tgts := do let max := find_max tgts 0,
            gls ← get_goals,
            found_goals ← find_wanted_goals tgts 1 max gls,
            set_goals (found_goals)
```

This is quite roundabout

```
meta def set_tactic_state (new_state : tactic_state) : tactic unit := λ s, (do
skip new_state)

meta def get_tactic_state : tactic tactic_state := λ s,
interaction_monad.result.success s s
```

Good to give all tactics a doc-string: /-- ... -/

```
-- focus_goals as described in the assignment specifications
-- The tactic can be invoked using the syntax requested
-- After focus_goals has been run, we restore the other goals. However,
-- Some of these might have been solved as a consequence of what was solved inside
-- the given goal block. For example, proving commutativity of addition will also
-- tell Lean the definition of addition we are using. This is unavoidable.
```

but what happens? test?

```
meta def tactic.interactive.focus_goals (pe : interactive.parse lean.parser.pexpr) (t : tactic.interactive.itactic) : tactic unit :=
do s ← get_tactic_state,
  s' ← get_goals,
  e ← to_expr pe,
  tgts ← eval_expr (list ℕ) e,
```

error reporting if parsing goes wrong?

```
  find_goals tgts,
  s'' ← get_goals,
  let s''' := list.diff s' s'',
```

a bit awkward that you 'set_goals' then immediately 'get_goals'. You should just know already!

```
  t,
  gls ← get_goals,
  if gls ≠ [] then do set_tactic_state s, fail "Failed to discharge the goals!" else
  set_goals s'''
```

So many primes!

```
section focus_goals_examples

-- Example of failing when there are no such goals.
example : ring ℝ :=
begin
  constructor,
  success_if_fail {focus_goals [1,2,16] {simp}}, -- Error message "No such goals!"
  exact neg_add_self,
  exact add_comm,
  exact one_mul,
  exact mul_one,
  exact left_distrib,
  exact right_distrib,
end
```

```
-- Example of succeeding and restoring other goals.
example (p q : Prop) : ¬ (p ∧ q) ↔ ¬ p ∨ ¬ q :=
begin
  constructor,
  focus_goals [1] {exact classical.not_and_distrib.mp},
  exact not_and_of_not_or_not,
end

-- Example of discharging some goals but not others and therefore failing.
example : true ∧ (true ∨ false) :=
begin
  split,
  success_if_fail {focus_goals [1,2] {repeat {trivial}}}, -- Failed with error
message "Failed to discharge the goals!"
  trivial,
  constructor,
  trivial,
end

-- Example of focus_goals succeeding
example : true ∧ true :=
begin
  split,
  focus_goals [1,2] {repeat {trivial}},
end

end focus_goals_examples
```

usually actually enter the namespace instead of just prefixing: easier to read.

```
-- work_on_goals as described in the assignment specifications
meta def tactic.interactive.work_on_goals (pe : interactive.parse
lean.parser.pexpr) (t : tactic.interactive.itactic) : tactic unit :=
do s ← get_tactic_state,
  s' ← get_goals,
  e ← to_expr pe,
  tgts ← eval_expr (list ℕ) e,
  find_goals tgts,
  s'' ← get_goals,
  let s''' := list.diff s' s'',
  t,
  gls ← get_goals,
  do set_goals (gls ++ s''')
```

lots of repetition here can you factor it out?

```
section work_on_goals_examples

open real

-- Example of solving some goals and then prepending these to the remaining
goals.
example : ring ℝ :=
begin
  constructor,
  work_on_goals [1, 3, 5, 11] {exact neg_add_self,
                              exact one_mul},
  exact left_distrib,
  exact add_comm,
  exact mul_one,
  exact right_distrib,
end

end work_on_goals_examples
```

Question 3

```
import tactic.interactive tactic.basic
import tactic.basic
import tactic.ring
import data.real.basic

open tactic real

meta def get_tactic_state : tactic tactic_state := λ s,
interaction_monad.result.success s s
```

The description of this tactic should definitely explain what it is doing to the tactic_state.

```
-- Takes a single tactic and the complexity function and checks if it is "less
complex" than the previous best
meta def run_each (c : tactic ℕ) (best : ℕ) (s₀ : tactic_state) (t : tactic
unit) : tactic ℕ
| s := match t s₀ with -- Get the current tactic state and try running the
tactic on the original tactic state
   | result.success _ s₁ := do match c s₁ with | result.success c₁ _ := -- If
it succeeds, try running our complexity function
                                                  if (c₁ < best) then
result.success c₁ s₁ -- If the result is less complex, store the new complexity

-- and the new tactic state (before the complexity function was run)
                                                              else
```

If best = 0 you don't even need to run the other tactic!

```
result.success best s -- If it is more complex, change nothing
                                                | _ := result.success best s --
If it fails, change nothing
                                end
   | _ := result.success best s -- If it fails, change nothing
   end
```

Such long lines....

```
-- Recursor function to deal with a list of tactics. Terminates if the list is
done.
-- Otherwise, use run_each to check if the head of the list is the best and
then repeat.
meta def run_list (c : tactic nat) (s₀ : tactic_state): ℕ → list (tactic unit)
→ tactic unit
| _ [] := do skip -- We have no more tactics left to try
| best (t :: ts) := do best' ← run_each c best s₀ t, -- Run the first tactic
and store its complexity
                       out ← run_list best' ts, -- Repeat on tail of list
                       return out
```

user-facing tactics should all have doc-strings: /-- This is how to use the tactic. -/.

```
-- Note that the best tactic state is carried along by being set as current
tactic state
-- Can be invoked using the syntax "run_best c [`[simp], `[refl], `[trivial,
dsimp]]"
-- Tried to remove the backtick syntax and use an interactive block, but Keeley
-- explained that these are not foundational in Lean and are instead "tacked
on",
-- so a new parser would have had to be written. This would have been
difficult,
-- hard to maintain and likely quite inefficient.
meta def run_best (c : tactic nat) (L : list (tactic unit)) : tactic unit :=
do s₀ ← get_tactic_state,
  run_list c s₀ 1000000000 L -- A large starting best value is used, because
the natural numbers have no upper bound
  -- and it needs to be larger than the output of the complexity function after
the first iteration
```

Lame!! :) So if I use numgoals * 10^{20} your tactic fails? :)

```
example : true ∨ (false ∧ true) :=
begin
   run_best (num_goals) [`[trivial, refl], `[constructor], `[ring]], --
Constructor is only tactic that makes progress, so constructor runs
   run_best (num_goals) [`[intros], `[trivial]], -- Trivial finishes the proof
end
```

Good, but even more testing would be better.