8/10 + 9/10

Interactive Theorem Proving Assignment 2

Martin Skilleter (u6679334)

8 April 2019

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

Q wasn't included in the PDF?

```
import data.nat.basic

-- First definition of prime (quite "prim(e)itive")
def prime (p : ℕ) := p ≥ 2 ∧ ∀ (m < p), m ∣ p → m = 1

open nat

-- A lemma to prove that 4 is not prime
lemma two_divides_four : 2 ∣ 4 :=
begin
    dsimp [(∣)],
    use 2,
    refl,
end

-- Question 1a
theorem four_is_not_prime : ¬ (prime 4) :=
begin
    unfold prime,
    simp [not_and_distrib],
    right,
    intros h,
    have h₁ : 2 ∣ 4, by exact two_divides_four,
    have h₂ : 2 < 4, by exact dec_trivial,
    have w := h 2 h₂ h₁,
    have w' : 2 ≠ 1, by exact dec_trivial,
    contradiction,
end

-- A lemma to prove that 5 is prime; dec_trivial was not used
because we can't use tactics in MathLib
lemma five_ge_two : 5 ≥ 2 :=
begin
    dsimp [(≥)],
    rw [le_iff_lt_or_eq],
    left,
    have h₁ : 3 ≠ 0, begin
        intros h,
        contradiction,
```

```
    end,
    have h₂ : 3 > 0, by exact nat.pos_of_ne_zero h₁,
    dsimp [(>)] at h₂,
    have h₃ : 1 < 4, by exact succ_lt_succ h₂,
    have h₄ : 2 < 5, by exact succ_lt_succ h₃,
    exact h₄,
end

-- This tactic gives a contradiction proof that a non-factor does not divide a number
-- I couldn't work out how to make the name n be given as an input
meta def case_bash : tactic unit :=
`[intros h, repeat {cases n, cases h, try {cases h}}]

lemma two_not_div_five (n : ℕ) : 5 ≠ 2 * n := by case_bash

lemma three_not_div_five (n : ℕ) : 5 ≠ 3 * n := by case_bash

lemma four_not_div_five (n : ℕ) : 5 ≠ 4 * n := by case_bash

-- Question 1a
theorem five_is_prime : prime 5 :=
begin
    unfold prime,
    split,
    exact five_ge_two,
    intros m h w,
    cases m,
    dsimp [(∣)] at w,
    cases w with c w,
    simp only [zero_mul] at w,
    contradiction,
    cases m,
    trivial,
    cases m,
    cases w with c w,
    have w' : 5 ≠ 2 * c, by apply two_not_div_five,
    contradiction,
```

```
    cases m,
    cases w with c w,
    have w' : 5 ≠ 3 * c, by apply three_not_div_five,
    contradiction,
    cases m,
    cases w with c w,
    have w' : 5 ≠ 4 * c, by apply four_not_div_five,
    contradiction,
    cases m,
    repeat {cases h with h h},
end

-- It is decidable that one natural number is greater than or
equal to another
instance decidable_ge {a b : ℕ} : decidable (a ≥ b) :=
begin
    dsimp [(≥)],
    apply nat.decidable_le b a,
end

-- Builds up in stages that the right-hand side of the and
statement in the definition of prime
-- is decidable. It begins by saying that it is decidable if a
number equals 1.
-- Then it is decidable if one number divides another.
Finally, a lemma from MathLib is used
-- which says that it is decidable that, if you have an upper
bound for a predicate and that
-- predicate is decidable for any specific natural number then
it is decidable for all natural
-- numbers.
instance decidable_divisors {p : ℕ} : decidable (∀ (m : ℕ), m
< p → m ∣ p → m = 1) :=
begin
    apply nat.decidable_ball_lt p (λ (m : ℕ) (a : m < p), m ∣
p → m = 1),
end
```

I would have preferred if you'd explicitly shown me the two instances that are being inferred in this step.

```
-- First instance of decidability about primes. Uses
```

```
and.decidable and the previous two
-- lemmas to say that the whole statement is decidable.
-- Note that haveI is used to update the instance cache after
the instances are declared.
instance decidable_prime : decidable_pred prime :=
begin
    intros p,
    unfold prime,
    haveI h₁ : decidable (p ≥ 2) := decidable_ge,
    haveI h₂ : decidable (∀ (m : ℕ), m < p → m ∣ p → m = 1) :=
decidable_divisors,
    apply and.decidable,
end
```

↑ commenting out these two lines results in the same proof! Similarly, if you comment those out, ~~and~~ decidable_div, the proof still works. Using 'apply' here, and relying on inference, means we can't be sure what's happening

```
-- A second definition of primality. We will prove that this
is equivalent to the
-- first definition.
def prime' (p : ℕ) := p ≥ 2 ∧ ∀ (m ≤ p/2), m ∣ p → m = 1

-- If a natural number a is less than or equal to b/2 then it
is less than b
-- Used as a lemma in prime_im_prime'
lemma lt_half_lt {a b : ℕ} : (b > 0) → a ≤ b/2 → a < b :=
begin
    intros h w,
    have h₁ : 2 > 1, by exact dec_trivial,
    have h₂ : b/2 < b, by apply div_lt_self h h₁,
    have h₃ : a < b, by apply lt_of_le_of_lt w h₂,
    exact h₃,
end

-- Essentially used to say that if p is prime then p > 0
lemma ge_two_gt_zero {p : ℕ} : p ≥ 2 → p > 0 :=
begin
    intros h,
    dsimp [(≥)] at h,
    have w₁ : 1 < p, by apply lt_of_succ_le h,
    have w₂ : 0 < 1, by exact dec_trivial,
    have w₃ : 0 < p, by apply lt_trans w₂ w₁,
```

```
    dsimp [(>)],
    exact w₃,
end
```

```
-- Non-zero factors of a number respect multiplication
lemma div_gt_zero_eq {m p k: ℕ} : m ∣ p → m > 0 → p / m = k →
p = m * k :=
begin
    intros h₁ h₂ h₃,
    have h₄ : m * p / m = p := nat.mul_div_cancel_left p h₂,
    subst h₃,
    have h₅ : m * p / m = m * (p / m) := nat.mul_div_assoc m
h₁,
    rw h₅ at h₄,
    exact h₄.symm,
end
```

```
-- The reciprocals of positive natural numbers behave as you
would
-- expect them to under ≤
lemma dvd_pos_lt {a b c : ℕ} : c ≤ b → 0 < c → a / b ≤ a /
c :=
begin
    intros h₁ h₂,
    apply (le_div_iff_mul_le (a/b) a h₂).2,
    have h₃ : a * c ≤ a * b := mul_le_mul_left a h₁,
    apply le_trans,
    swap,
    apply div_mul_le_self',
    exact b,
    apply nat.mul_le_mul_left (a/b),
    exact h₁,
end
```

```
-- The first definition prime → the second definition prime'
theorem prime_im_prime' {p : ℕ} : prime p → prime' p :=
begin
    intro h,
    unfold prime at h,
```

```
    unfold prime',
    split,
    exact h.1,
    have ge := h.1,
    have h₁ := h.2,
    intros m w₁ w₂,
    have w₃ : p > 0, by apply ge_two_gt_zero ge,
    have w₄ : m < p, by apply lt_half_lt w₃ w₁,
    have w₅ : m = 1, by apply h₁ m w₄ w₂,
    exact w₅,
end

-- The second definition prime' → the first definition prime
theorem prime'_im_prime {p : ℕ} : prime' p → prime p :=
begin
    intros w,
    unfold prime' at w,
    unfold prime,
    split,
    exact w.1,
    have ge := w.1,
    intros m w₁ w₂,
    by_cases (m ≤ p/2),
    exact w.2 m h w₂,
    simp at h,
    have two_gt_zero : 2 > 0, by exact dec_trivial,
    -- Wasn't written as a lemma because I felt it would be
unwieldy to pass
    -- hypotheses around
```

✓ okay!

```
    have one_lt_half : p / 2 ≥ 1, begin
        have le : 2 ≤ p := ge,
        have z : 2/2 ≤ p / 2 := nat.div_le_div_right le,
        have h₁ : 2 / 2 = 1 := nat.div_self two_gt_zero,
        rw [h₁] at z,
        exact z,
    end,
    -- Same as above. Relies on hypotheses.
    have h₄ : m > 0, begin
        have zero_lt_one : 0 < 1, by exact dec_trivial,
```

```
        have h₂ : 0 < p / 2, by apply lt_of_lt_of_le
zero_lt_one one_lt_half,
        exact (lt_trans h₂ h),
    end,
    have m_gt_one : m > 1, by apply lt_of_le_of_lt one_lt_half
h,
    have m_ge_two : m ≥ 2, by apply succ_le_of_lt m_gt_one,
    have h₁ : p / m ≤ p / 2 := dvd_pos_lt m_ge_two
two_gt_zero,
    have h₂ : p / m ∣ p, by apply div_dvd_of_dvd w₂,
    have h₃ : p / m = 1, by apply w.2 (p/m) h₁ h₂,
    have h₅ : p = m * 1 := div_gt_zero_eq w₂ h₄ h₃,
    simp at h₅,
    subst h₅,
    have h₆ : p ≥ p, by exact le_refl p,
    have h₇ : ¬ (p < p), by simp,
    exact (absurd w₁ h₇),
end
```

Good.

```
-- Uses the constructor for ↔ and the above implications to
give an equivalence statement
theorem prime_equiv_statement {p : ℕ} : prime p ↔ prime' p :=
⟨prime_im_prime', prime'_im_prime⟩
```

```
-- See explanation above decidable_divisors. Construction is
the same but upper bound used is p/2
-- instead of p
instance decidable_divisors' {p : ℕ}: decidable (∀ (m : ℕ), m
< p / 2 → m ∣ p → m = 1) :=
nat.decidable_ball_lt (p / 2) (λ (m : ℕ) (a : m < p / 2), m ∣
p → m = 1)
```

```
-- Says that it is possible to decide if a natural number is
prime under prime'
instance decidable_prime' : decidable_pred prime' :=
begin
    intros p,
    unfold prime',
    haveI h₁ : decidable (p ≥ 2) := decidable_ge,
```

```
    haveI h₂ : decidable (∀ (m : ℕ), m < p / 2 → m ∣ p → m = 1) := decidable_divisors',
    apply and.decidable,
end

-- Gives a "better" instance for decidable_prime by applying the equivalence of
-- prime and prime'. It is better because you only need to check half as many natural numbers.
instance decidable_prime_v2 : decidable_pred prime :=
begin
    intros p,
    apply decidable_of_iff' (prime' p) (@prime_equiv_statement p),
end
```

Which one runs faster? How would you test this?

Question 1

where's Question 1?

```
import data.real.basic algebra.field_power
import algebra.pi_instances
import algebra.module
import ring_theory.algebra

-- Question 2a
structure quaternion : Type :=
(re : ℝ) (i : ℝ) (j : ℝ) (k : ℝ)

notation `ℍ` := quaternion

namespace quaternion
```

do you know why this is called 'eta'?

```
@[simp] theorem eta : ∀ p : ℍ, quaternion.mk p.re p.i p.j p.k = p
| ⟨a, b, c, d⟩ := rfl

@[extensionality] theorem ext : ∀ {p q : ℍ}, p.re = q.re → p.i = q.i →
p.j = q.j → p.k = q.k → p = q
| ⟨pr, pi, pj, pk⟩ ⟨_, _, _, _⟩ rfl rfl rfl rfl := rfl

theorem ext_iff {p q : ℍ} : p = q ↔ p.re = q.re ∧ p.i = q.i ∧ p.j = q.j ∧
p.k = q.k :=
⟨λ H, by simp [H], by {rintros ⟨a, ⟨b, ⟨c, d⟩⟩⟩, apply ext; assumption}⟩
```

I you could omit 'apply' here.

```
def of_real (r : ℝ) : ℍ := ⟨r, 0, 0, 0⟩
instance : has_coe ℝ ℍ := ⟨of_real⟩
@[simp] lemma of_real_eq_coe (r : ℝ) : of_real r = r := rfl

@[simp] lemma of_real_re (r : ℝ) : (r : ℍ).re = r := rfl
@[simp] lemma of_real_i (r : ℝ) : (r : ℍ).i = 0 := rfl
@[simp] lemma of_real_j (r : ℝ) : (r : ℍ).j = 0 := rfl
@[simp] lemma of_real_k (r : ℝ) : (r : ℍ).k = 0 := rfl

@[simp] theorem of_real_inj {z w : ℝ} : (z : ℍ) = w ↔ z = w :=
⟨congr_arg re, congr_arg _⟩

instance : has_zero ℍ := ⟨(0 : ℝ)⟩
instance : inhabited ℍ := ⟨0⟩
```

```
@[simp] lemma zero_re : (0 : ℍ).re = 0 := rfl
@[simp] lemma zero_i : (0 : ℍ).i = 0 := rfl
@[simp] lemma zero_j : (0 : ℍ).j = 0 := rfl
@[simp] lemma zero_k : (0 : ℍ).k = 0 := rfl
@[simp] lemma of_real_zero : ((0 : ℝ) : ℍ) = 0 := rfl

@[simp] theorem of_real_eq_zero {z : ℝ} : (z : ℍ) = 0 ↔ z = 0 :=
of_real_inj
@[simp] theorem of_real_ne_zero {z : ℝ} : (z : ℍ) ≠ 0 ↔ z ≠ 0 := not_congr
of_real_eq_zero

instance : has_one ℍ := ⟨(1 : ℝ)⟩

@[simp] lemma one_re : (1 : ℍ).re = 1 := rfl
@[simp] lemma one_i : (1 : ℍ).i = 0 := rfl
@[simp] lemma one_j : (1 : ℍ).j = 0 := rfl
@[simp] lemma one_k : (1 : ℍ).k = 0 := rfl
@[simp] lemma of_real_one : ((1 : ℝ) : ℍ) = 1 := rfl

def I : ℍ := ⟨0, 1, 0, 0⟩
def J : ℍ := ⟨0, 0, 1, 0⟩
def K : ℍ := ⟨0, 0, 0, 1⟩
```

not sure how useful these are.

```
@[simp] lemma I_re : I.re = 0 := rfl
@[simp] lemma I_i : I.i = 1 := rfl
@[simp] lemma I_j : I.j = 0 := rfl
@[simp] lemma I_k : I.k = 0 := rfl

@[simp] lemma J_re : J.re = 0 := rfl
@[simp] lemma J_i : J.i = 0 := rfl
@[simp] lemma J_j : J.j = 1 := rfl
@[simp] lemma J_k : J.k = 0 := rfl

@[simp] lemma K_re : K.re = 0 := rfl
@[simp] lemma K_i : K.i = 0 := rfl
@[simp] lemma K_j : K.j = 0 := rfl
@[simp] lemma K_k : K.k = 1 := rfl

instance : has_add ℍ := ⟨λ z w, ⟨z.re + w.re, z.i + w.i, z.j + w.j, z.k +
w.k⟩⟩
```

```
@[simp] lemma add_re (z w : ℍ) : (z + w).re = z.re + w.re := rfl
@[simp] lemma add_i (z w : ℍ) : (z + w).i = z.i + w.i := rfl
@[simp] lemma add_j (z w : ℍ) : (z + w).j = z.j + w.j := rfl
@[simp] lemma add_k (z w : ℍ) : (z + w).k = z.k + w.k := rfl
@[simp] lemma of_real_add (r s : ℝ) : ((r + s : ℝ) : ℍ) = r + s :=
ext_iff.2 $ by simp

@[simp] lemma of_real_bit0 (r : ℝ) : ((bit0 r : ℝ) : ℍ) = bit0 r :=
ext_iff.2 $ by simp [bit0]
@[simp] lemma of_real_bit1 (r : ℝ) : ((bit1 r : ℝ) : ℍ) = bit1 r :=
ext_iff.2 $ by simp [bit1]
```

do you know what these are for?

```
instance : has_neg ℍ := ⟨λ z, ⟨-z.re, -z.i, -z.j, -z.k⟩⟩

@[simp] lemma neg_re (z : ℍ) : (-z).re = -z.re := rfl
@[simp] lemma neg_i (z : ℍ) : (-z).i = -z.i := rfl
@[simp] lemma neg_j (z : ℍ) : (-z).j = -z.j := rfl
@[simp] lemma neg_k (z : ℍ) : (-z).k = -z.k := rfl
@[simp] lemma of_real_neg (r : ℝ) : ((-r : ℝ) : ℍ) = -r := ext_iff.2 $ by
simp
```

`by ext; simp` also works.

```
instance : has_mul ℍ := ⟨λ p q, ⟨p.re*q.re - p.i*q.i - p.j*q.j - p.k*q.k,
                                p.re*q.i + p.i*q.re + p.j*q.k - p.k*q.j,
                                p.re*q.j - p.i*q.k + p.j*q.re + p.k*q.i,
                                p.re*q.k + p.i*q.j - p.j*q.i + p.k*q.re⟩⟩

@[simp] lemma mul_re (p q : ℍ) : (p*q).re = p.re*q.re - p.i*q.i - p.j*q.j -
p.k*q.k := rfl
@[simp] lemma mul_i (p q : ℍ) : (p*q).i = p.re*q.i + p.i*q.re + p.j*q.k -
p.k*q.j := rfl
@[simp] lemma mul_j (p q : ℍ) : (p*q).j = p.re*q.j - p.i*q.k + p.j*q.re +
p.k*q.i := rfl
@[simp] lemma mul_k (p q : ℍ) : (p*q).k = p.re*q.k + p.i*q.j - p.j*q.i +
p.k*q.re := rfl
@[simp] lemma of_real_mul (r s : ℝ) : ((r * s : ℝ) : ℍ) = r * s :=
ext_iff.2 $ by simp
@[simp] lemma mul_comm_re (r : ℝ) (p : ℍ) : (r : ℍ) * p = p * r :=
by ext; {dsimp, simp [mul_comm]}

@[simp] lemma I_mul_I : I * I = -1 := ext_iff.2 $ by simp
@[simp] lemma J_mul_J : J * J = -1 := ext_iff.2 $ by simp
@[simp] lemma K_mul_K : K * K = -1 := ext_iff.2 $ by simp
```

```
lemma one_ne_zero : (1 : ℍ) ≠ 0 := by simp [ext_iff]
lemma I_ne_zero : (I : ℍ) ≠ 0 := mt (congr_arg i) zero_ne_one.symm
lemma J_ne_zero : (J : ℍ) ≠ 0 := mt (congr_arg j) zero_ne_one.symm
lemma K_ne_zero : (K : ℍ) ≠ 0 := mt (congr_arg k) zero_ne_one.symm

lemma mk_eq_add_mul_I_add_mul_J_add_mul_K (a b c d : ℝ) :
quaternion.mk a b c d = a + b * I + c * J + d * K :=
ext_iff.2 $ by simp

@[simp] lemma re_add_i_add_j_add_k (z : ℍ) : (z.re : ℍ) + z.i * I + z.j * J
+ z.k * K = z :=
ext_iff.2 $ by simp

def real_prod_equiv : ℍ ≃ (ℝ × ℝ × ℝ × ℝ) :=
{ to_fun := λ z, ⟨z.re, z.i, z.j, z.k⟩,
 inv_fun := λ p, ⟨p.1, p.2.1, p.2.2.1, p.2.2.2⟩,
 left_inv := λ ⟨a, b, c, d⟩, rfl,
 right_inv := λ ⟨a, b, c, d⟩, rfl }

@[simp] theorem real_prod_equiv_apply (z : ℍ) : real_prod_equiv z = (z.re,
z.i, z.j, z.k) := rfl

-- Question 2d
def conj (z : ℍ) : ℍ := ⟨z.re, -z.i, -z.j, -z.k⟩

@[simp] lemma conj_re (z : ℍ) : (conj z).re = z.re := rfl
@[simp] lemma conj_i (z : ℍ) : (conj z).i = -z.i := rfl
@[simp] lemma conj_j (z : ℍ) : (conj z).j = -z.j := rfl
@[simp] lemma conj_k (z : ℍ) : (conj z).k = -z.k := rfl
@[simp] lemma conj_of_real (r : ℝ) : conj r = r := ext_iff.2 $ by simp
[conj]

@[simp] lemma conj_zero : conj 0 = 0 := ext_iff.2 $ by simp [conj]
@[simp] lemma conj_one : conj 1 = 1 := ext_iff.2 $ by simp
@[simp] lemma conj_I : conj I = -I := ext_iff.2 $ by simp
@[simp] lemma conj_neg_I : conj (-I) = I := ext_iff.2 $ by simp
@[simp] lemma conj_J : conj J = -J := ext_iff.2 $ by simp
@[simp] lemma conj_neg_J : conj (-J) = J := ext_iff.2 $ by simp
@[simp] lemma conj_K : conj K = -K := ext_iff.2 $ by simp
@[simp] lemma conj_neg_K : conj (-K) = K := ext_iff.2 $ by simp
```

```
@[simp] lemma conj_add (z w : ℍ) : conj (z + w) = conj z + conj w :=
ext_iff.2 $ by simp

@[simp] lemma conj_neg (z : ℍ) : conj (-z) = -conj z := rfl

@[simp] lemma conj_mul (p q : ℍ) : conj (p * q) = conj q * conj p := by
simp [ext_iff, mul_comm, add_comm]

@[simp] lemma conj_conj (z : ℍ) : conj (conj z) = z :=
ext_iff.2 $ by simp

lemma conj_bijective : function.bijective conj :=
⟨function.injective_of_has_left_inverse ⟨conj, conj_conj⟩,
function.surjective_of_has_right_inverse ⟨conj, conj_conj⟩⟩

lemma conj_inj {z w : ℍ} : conj z = conj w ↔ z = w :=
conj_bijective.1.eq_iff

@[simp] lemma conj_eq_zero {z : ℍ} : conj z = 0 ↔ z = 0 :=
by simpa using @conj_inj z 0

@[simp] lemma eq_conj_iff_real (z : ℍ) : conj z = z ↔ ∃ r : ℝ, z = r :=
begin
   split,
   intros h,
   use z.re,
   simp [ext_iff] at h,
   rcases h with ⟨hi, ⟨hj, hk⟩⟩,
   ext,
   refl,
   repeat {apply eq_zero_of_neg_eq, assumption},
   intros h,
   cases h with r h,
   simp only [ext_iff] at h,
   rcases h with ⟨hre, ⟨hi, ⟨hj, hk⟩⟩⟩,
   ext,
   refl,
   repeat {simp, try {rw hi}, try {rw hj}, try {rw hk}, simp},
end
```

```
def norm_sq (z : ℍ) : ℝ := z.re * z.re + z.i * z.i + z.j * z.j + z.k * z.k

@[simp] lemma norm_sq_of_real (r : ℝ) : norm_sq r = r * r :=
by simp [norm_sq]

@[simp] lemma norm_sq_zero : norm_sq 0 = 0 := by simp [norm_sq]
@[simp] lemma norm_sq_one : norm_sq 1 = 1 := by simp [norm_sq]
@[simp] lemma norm_sq_I : norm_sq I = 1 := by simp [norm_sq]
@[simp] lemma norm_sq_J : norm_sq J = 1 := by simp [norm_sq]
@[simp] lemma norm_sq_K : norm_sq K = 1 := by simp [norm_sq]

lemma norm_sq_nonneg (z : ℍ) : 0 ≤ norm_sq z :=
add_nonneg ( add_nonneg ( add_nonneg (mul_self_nonneg _) (mul_self_nonneg _)) (mul_self_nonneg _)) (mul_self_nonneg _)

lemma zero_of_sum_squares_zero {a b c d : ℝ} : a*a + b*b + c*c + d*d = 0 →
a = 0 ∧ b = 0 ∧ c = 0 ∧ d = 0 :=
begin
   intros h,
   have ha : a*a ≥ 0, by apply mul_self_nonneg,
   have hb : b*b ≥ 0, by apply mul_self_nonneg,
   have hc : c*c ≥ 0, by apply mul_self_nonneg,
   have hd : d*d ≥ 0, by apply mul_self_nonneg,
   have hcd : c*c + d*d ≥ 0, by exact add_nonneg hc hd,
   have hbcd : b*b + (c*c + d*d) ≥ 0, by exact add_nonneg hb hcd,
   rw ←add_assoc at hbcd,
   have w₁ := (add_eq_zero_iff_eq_zero_and_eq_zero_of_nonneg_of_nonneg ha hbcd).1,
   have h₁ : a*a + b*b + c*c + d*d = a*a +(b*b + c*c + d*d), by ring,
   rw ←h₁ at w₁,
   have w₂ := w₁ h,
   cases w₂,
   have h₂ : b*b + c*c + d*d = b*b +(c*c + d*d), by ring,
   have w₃ := (add_eq_zero_iff_eq_zero_and_eq_zero_of_nonneg_of_nonneg hb hcd).1,
   rw ←h₂ at w₃,
   have w₄ := w₃ w₂_right,
   cases w₄,
   have w₅ := (add_eq_zero_iff_eq_zero_and_eq_zero_of_nonneg_of_nonneg hc hd).1 w₄_right,
   cases w₅,
   repeat {split},
```

```
   exact eq_zero_of_mul_self_eq_zero w₂_left,
   exact eq_zero_of_mul_self_eq_zero w₄_left,
   exact eq_zero_of_mul_self_eq_zero w₅_left,
   exact eq_zero_of_mul_self_eq_zero w₅_right,
end

@[simp] lemma norm_sq_eq_zero {z : ℍ} : norm_sq z = 0 ↔ z = 0 :=
begin
   split,
   intros h,
   dsimp [norm_sq] at h,
   have w : z.re = 0 ∧ z.i = 0 ∧ z.j = 0 ∧ z.k = 0, by exact
zero_of_sum_squares_zero h,
   rcases w with ⟨wre, ⟨wi, ⟨wj, wk⟩⟩⟩,
   ext,
   repeat {assumption},
   intros h,
   dsimp [norm_sq],
   simp only [ext_iff] at h,
   rcases h with ⟨hre, ⟨hi, ⟨hj, hk⟩⟩⟩,
   rw [hre, hi, hj, hk],
   simp,
end

@[simp] lemma norm_sq_pos {z : ℍ} : 0 < norm_sq z ↔ z ≠ 0 :=
by rw [lt_iff_le_and_ne, ne, eq_comm]; simp [norm_sq_nonneg]

@[simp] lemma norm_sq_neg (z : ℍ) : norm_sq (-z) = norm_sq z :=
by simp [norm_sq]

@[simp] lemma norm_sq_conj (z : ℍ) : norm_sq (conj z) = norm_sq z :=
by simp [norm_sq]

@[simp] lemma norm_sq_mul (z w : ℍ) : norm_sq (z * w) = norm_sq z * norm_sq
w :=
by dsimp [norm_sq]; ring

lemma norm_sq_add (z w : ℍ) : norm_sq (z + w) =
 norm_sq z + norm_sq w + 2 * (z * conj w).re :=
by dsimp [norm_sq]; ring
```

```
-- Question 2e
theorem mul_conj (z : ℍ) : z * conj z = norm_sq z :=
ext_iff.2 $ by simp [norm_sq, mul_comm]

-- Question 2e
-- I did both because, in general, quaternion multiplication does not
commute
theorem mul_conj' (z : ℍ) : conj z * z = norm_sq z :=
ext_iff.2 $ by simp [norm_sq, mul_comm]

theorem mul_conj_is_re (z : ℍ) : ∃ (r : ℝ), (r : ℍ) = z * conj z :=
by {use norm_sq z, rw mul_conj}

theorem mul_conj'_is_re (z : ℍ) : ∃ (r : ℝ), (r : ℍ) = conj z * z :=
by {use norm_sq z, rw mul_conj'}

theorem add_conj (z : ℍ) : z + conj z = (2 * z.re : ℝ) :=
ext_iff.2 $ by simp [two_mul]

lemma add_assoc (a b c : ℍ) : a + b + c = a + (b + c) :=
by {ext, repeat {simp}}

lemma add_comm (a b : ℍ) : a + b = b + a :=
by {ext, repeat {simp}}

lemma add_left_neg (a : ℍ) : -a + a = 0 :=
by {ext, repeat {simp}}

lemma zero_add (a : ℍ) : 0 + a = a :=
by {ext, repeat {simp}}

lemma add_zero (a : ℍ) : a + 0 = a :=
by {rw add_comm, exact zero_add a}

instance : add_comm_group ℍ :=
{add_assoc := add_assoc, zero_add := zero_add,
add_zero := add_zero, add_left_neg := add_left_neg, add_comm := add_comm,
..}

lemma mul_assoc (a b c : ℍ) : a * b * c = a * (b * c) :=
by {ext; {simp, ring}}
```

```
lemma one_mul (a : ℍ) : 1 * a = a :=
by {ext; {simp}}

lemma mul_one (a : ℍ) : a * 1 = a :=
by {ext; {simp}}

lemma left_distrib (a b c : ℍ) : a * (b + c) = a * b + a * c :=
by {ext; {simp, ring}}

lemma right_distrib (a b c : ℍ) : (a + b) * c = a * c + b * c :=
by {ext; {simp, ring}}

-- Question 2b
-- Previous lemmas were written explicitly to make ring instance compile
faster
instance : ring ℍ :=
{mul := (*), mul_assoc := mul_assoc, one := 1, one_mul := one_mul,
   mul_one := mul_one, left_distrib := left_distrib, right_distrib :=
right_distrib, .. (by apply_instance : add_comm_group ℍ)}
.

@[simp] lemma sub_re (z w : ℍ) : (z - w).re = z.re - w.re := rfl
@[simp] lemma sub_i (z w : ℍ) : (z - w).i = z.i - w.i := rfl
@[simp] lemma sub_j (z w : ℍ) : (z - w).j = z.j - w.j := rfl
@[simp] lemma sub_k (z w : ℍ) : (z - w).k = z.k - w.k := rfl
@[simp] lemma of_real_sub (r s : ℝ) : ((r - s : ℝ) : ℍ) = r - s :=
ext_iff.2 $ by simp
@[simp] lemma of_real_pow (r : ℝ) (n : ℕ) : ((r ^ n : ℝ) : ℍ) = r ^ n :=
by induction n; simp [*, of_real_mul, pow_succ]

theorem sub_conj (z : ℍ) : z - conj z = (2 * z.i : ℝ) * I + (2 * z.j : ℝ) *
J + (2 * z.k : ℝ) * K:=
ext_iff.2 $ by simp [two_mul]

lemma norm_sq_sub (z w : ℍ) : norm_sq (z - w) =
 norm_sq z + norm_sq w - 2 * (z * conj w).re :=
by rw [sub_eq_add_neg, norm_sq_add]; simp [-mul_re]

noncomputable instance : has_inv ℍ := ⟨λ z, conj z * ((norm_sq z)⁻¹:ℝ)⟩
```

```
theorem inv_def (z : ℍ) : z⁻¹ = conj z * ((norm_sq z)⁻¹:ℝ) := rfl
@[simp] lemma inv_re (z : ℍ) : (z⁻¹).re = z.re / norm_sq z := by simp
[inv_def, division_def]
@[simp] lemma inv_i (z : ℍ) : (z⁻¹).i = -z.i / norm_sq z := by simp
[inv_def, division_def]
@[simp] lemma inv_j (z : ℍ) : (z⁻¹).j = -z.j / norm_sq z := by simp
[inv_def, division_def]
@[simp] lemma inv_k (z : ℍ) : (z⁻¹).k = -z.k / norm_sq z := by simp
[inv_def, division_def]

@[simp] lemma of_real_inv (r : ℝ) : ((r⁻¹ : ℝ) : ℍ) = r⁻¹ :=
ext_iff.2 $ begin
 simp,
 by_cases r = 0, {simp [h]},
 rw [← div_div_eq_div_mul, div_self h, one_div_eq_inv]
end

protected lemma inv_zero : (0⁻¹ : ℍ) = 0 :=
by rw [← of_real_zero, ← of_real_inv, inv_zero]

theorem mul_inv_cancel (z : ℍ) (h : z ≠ 0) : z * z⁻¹ = 1 :=
by rw [inv_def, ← mul_assoc, mul_conj, ← of_real_mul,
 mul_inv_cancel (mt norm_sq_eq_zero.1 h), of_real_one]

theorem inv_mul_cancel (z : ℍ) (h : z ≠ 0) : z⁻¹ * z = 1 :=
by rw [inv_def, ← mul_comm_re, mul_assoc, mul_conj', ← of_real_mul,
inv_mul_cancel (mt norm_sq_eq_zero.1 h), of_real_one]

instance re.is_add_group_hom : is_add_group_hom quaternion.re :=
by refine_struct {..}; simp

instance i.is_add_group_hom : is_add_group_hom quaternion.i :=
by refine_struct {..}; simp

instance j.is_add_group_hom : is_add_group_hom quaternion.j :=
by refine_struct {..}; simp

instance k.is_add_group_hom : is_add_group_hom quaternion.k :=
by refine_struct {..}; simp

instance of_real.is_ring_hom : is_ring_hom (coe : ℝ → ℍ) :=
by {constructor, refl, apply of_real_mul, apply of_real_add}
```

indicates there's a missing simp lemma!

```
def smul : ℝ → ℍ → ℍ := λ r z, of_real r * z

infix `•` := smul

instance : has_scalar ℝ ℍ := {smul := smul}

lemma add_smul (r s : ℝ) (z : ℍ) : (r + s) • z = r • z + s • z :=
by {dsimp [(•)], ext; {simp, ring}}

lemma zero_smul (z : ℍ) : 0 • z = 0 :=
by {dsimp [(•)], ext; simp}

lemma one_smul (z : ℍ) : 1 • z = z :=
by {dsimp [(•)], simp}

lemma mul_smul (r s : ℝ) (z : ℍ) : (r * s) • z = r • (s • z) :=
by {dsimp [(•)], ext; {rw [of_real_mul, mul_assoc]}}

lemma smul_add (r : ℝ) (z w : ℍ) : r • (z + w) = r • z + r • w :=
by {dsimp [(•)], ext; {simp, ring}}

lemma smul_zero (r : ℝ) : r • 0 = 0 :=
by {dsimp [(•)], simp}

instance : semimodule ℝ ℍ :=
{add_smul := add_smul, zero_smul := zero_smul, one_smul := one_smul,
   mul_smul := mul_smul, smul_add := smul_add, smul_zero := smul_zero, ..}

instance : module ℝ ℍ := by constructor

lemma smul_def' (r : ℝ) (z : ℍ) : r • z = of_real r * z :=
begin
   dsimp [(•)],
   simp,
end

-- Question 2c
instance : algebra ℝ ℍ :=
{to_fun := of_real, commutes' := by simp, smul_def' := smul_def', hom :=
of_real.is_ring_hom}
```

```
-- Question 2g
noncomputable instance : division_ring ℍ :=
{inv := has_inv.inv, zero_ne_one := one_ne_zero.symm, mul_inv_cancel :=
mul_inv_cancel,
   inv_mul_cancel := inv_mul_cancel, .. (by apply_instance : ring ℍ)}
.

end quaternion
```

Question 2f: Normed Spaces

Note: I endeavoured to complete the proof that the quaternions were a normed vector space over the real numbers but was unable to prove the Triangle Inequality for the norm. The approach I would have taken would be to prove that the norm was induced by the standard inner product on Euclidean space (I had already proven that Q is isomorphic to R^4 as real vector spaces) but could not find inner products defined anywhere in MathLib.

```
import .Q2
import analysis.normed_space.basic

noncomputable theory

namespace quaternion

instance : has_norm ℍ := ⟨λ z, real.sqrt (norm_sq z)⟩

instance : has_dist ℍ := ⟨λ z w, ∥z-w∥⟩

lemma dist_self (z : ℍ) : dist z z = 0 :=
by {dsimp [dist], rw add_right_neg, dsimp [norm], simp}

lemma eq_of_dist_eq_zero (z w : ℍ) : dist z w = 0 → z = w :=
begin
   intros h,
   dsimp [dist, norm] at h,
   have w := (real.sqrt_eq_zero (norm_sq_nonneg (z + -w))).1 h,
   rw norm_sq_eq_zero at w,
   exact eq_of_sub_eq_zero w,
end
```

```
lemma dist_comm (z w : ℍ) : dist z w = dist w z :=
begin
   dsimp [dist, norm],
   have h₁ : norm_sq (z-w) ≥ 0 := norm_sq_nonneg (z-w),
   have h₂ : norm_sq (w-z) ≥ 0 := norm_sq_nonneg (w-z),
   apply (real.sqrt_inj h₁ h₂).2,
   dsimp [norm_sq],
   ring,
end
.

lemma le_of_eq_of_le {a b c : ℝ} : b = c → a ≤ b → a ≤ c :=
by {intros h w, rw [h] at w, exact w}

-- It turned out to be absolutely impossible to prove this without
Cauchy-Schwartz.
-- I would have proven that the norm was induced by an inner product first,
-- but inner products are not implemented in MathLib anywhere that I could
find.
lemma norm_triangle (z w : ℍ) : ∥z+w∥ ≤ ∥z∥+∥w∥ :=
begin
   dsimp [norm],
   have h₁ := add_nonneg (real.sqrt_nonneg (norm_sq z)) (real.sqrt_nonneg
(norm_sq w)),
   rw [real.sqrt_le_left h₁, pow_two],
   ring,
   rw [real.ring.right_distrib, ←pow_two, real.sqr_sqrt (norm_sq_nonneg z),
real.sqr_sqrt (norm_sq_nonneg w)],
   have h₂ := real.sqrt_mul (norm_sq_nonneg w) (norm_sq z),
   have h₃ := congr_arg (λ (x : ℝ), 2 * x) h₂,
   simp at h₃,
   rw [←real.ring.mul_assoc] at h₃,
   have h₄ : norm_sq z + 2 * real.sqrt (norm_sq w) * real.sqrt (norm_sq z) +
norm_sq w = norm_sq z + 2 * real.sqrt (norm_sq w * norm_sq z) + norm_sq w, by
{simp, exact h₃.symm},
   apply le_of_eq_of_le h₄.symm,
   clear h₃ h₄,
   rw [norm_sq_add],
   simp,
   have zero_lt_two : (0 : ℝ) < 2 := by linarith,
   apply (@mul_le_mul_left _ _ (z.i * w.i + (z.j * w.j + (z.k * w.k + z.re *
w.re))) _ 2 zero_lt_two).2,
   clear zero_lt_two,
   sorry,
```

```
end

lemma dist_triangle (x y z : ℍ) : dist x z ≤ dist x y + dist y z :=
begin
   dsimp [dist],
   have w := norm_triangle (x-y) (y-z),
   simp at w,
   exact w,
end

instance : metric_space ℍ :=
{dist := dist, dist_self := dist_self, eq_of_dist_eq_zero :=
eq_of_dist_eq_zero,
   dist_comm := dist_comm, dist_triangle := dist_triangle, ..}

lemma dist_eq (x y : ℍ) : dist x y = ∥x-y∥ :=
by {dsimp [dist], refl}

instance : normed_group ℍ :=
{dist := dist, dist_eq := dist_eq, .. (by apply_instance : metric_space ℍ)}

instance : vector_space ℝ ℍ := by constructor

lemma norm_smul (r : ℝ) (z : ℍ) : ∥r • z∥ = ∥r∥ * ∥z∥ :=
begin
   dsimp [norm],
   rw ←real.sqrt_mul_self_eq_abs,
   rw ←(real.sqrt_mul' (r*r) (norm_sq_nonneg z)),
   have w := mul_nonneg (mul_self_nonneg r) (norm_sq_nonneg z),
   rw (real.sqrt_inj (norm_sq_nonneg (r • z)) w),
   dsimp [norm_sq, (•)],
   ring,
end

-- Question 2f
instance : normed_space ℝ ℍ :=
{norm_smul := norm_smul, .. (by apply_instance : vector_space ℝ ℍ)}

end quaternion
```