```
-- @file   jensen.lean
-- @author Brandon H. Gomes
-- @affil  Rutgers University

/-!
# Jensen's Inequality

In this file, we prove a famous and important inequality. The file is self-contained and
requires no other definitions other than those in the Lean core.

We first state the classical Jensen's Inequality:

- Theorem. Let `μ` be a positive measure on a σ-algebra `𝔐` in a set `Ω`, so that
  `μ(Ω) = 1`. If `f` is a real function in `L¹(μ)` such that `∀x ∈ Ω, f(x) ∈ (a,b)` and if
  `ϕ` is convex on `(a,b)`, then `ϕ (∫_Ω f dμ) ≤ ∫_Ω (ϕ ∘ f) dμ`. More succinctly,
  `ϕ (∫ f) ≤ ∫ (ϕ ∘ f)`.

The inequality follows from certain facts about integration, real numbers, convex
functions, ... etc. The theorem we prove is a strict generalization in which we state the
theorem in as high of an abstract form as possible with facts about subtraction and certain
kinds of relations. We will define a type-theoretic analogue of integration theory and of
the theory of convex functions. More details are discussed below.

We begin our discussion with power sets.
-/

--————————————————————————————————————————————————————————————————————————————————--
/--
`Power X` (`𝒫 X`) The powerset of a type.

Given a type `X : 𝒰i` we can consider the type of functions from `X` into a fixed type
universe `𝒰j`.

One can view this type of functions `𝒫 X` as the space of characteristic functions on `X`.
Classically, we can associate any subset `A ⊆ S` to an appropriate characteristic function
`χ_A : S → {0,1}` which takes value `1` on `A` and `0` elsewhere in `S`. This association
is a "bijective correspondence", that is, we can freely change our interpretation of the
subset `A` either as an actual subset or as a characteristic function. In the language of
types there is no natural notion of "subset" but there is a natural (but non-canonical
since it depends on the universe level `j`) notion of a characteristic function.

In general `𝒫 X` behaves like the classical power set of `X`, i.e. a complete lattice:
  - Bottom:                `⊥ := λ x, 𝟘`
  - Top:                   `⊤ := λ x, 𝟙`
  - Union:                 `P ∪ Q := λ x, P x ∨ Q x`
  - Intersection:          `P ∩ Q := λ x, P x ∧ Q x`
  - Aribtrary Union:       `⋃ 𝓕 := λ x, Σ i, (𝓕 i) x`
  - Aribtrary Intersection: `⋂ 𝓕 := λ x, Π i, (𝓕 i) x`

where `𝟘` and `𝟙` are the empty type (the type with no terms) and unit type (the type with
one canonical term) respectively. The `⊥` element can be called the "empty set" and the
`⊤` element can be called the "total space" and can be associated to `X` itself.
Classically one can think of the union as the sum of characteristic functions `χ_P + χ_Q`
and the intersection as the product `χ_P * χ_Q`. There is a similar story for the
arbitrary union/intersection.
```

NB: To know that a term `x : X` is contained in the subset `A : 𝒫 X`, we need a witness of the form `w : A x`, since `A` is a function and concatenation is function application.

We will be using the `Power` construction many times throughout since partial functions play an important role in the `jensen_inequality` and also in the theory of integration in general; partial functions only make sense if we have defined the notion of power set.
-/
```
definition {𝓤i 𝓤j} Power (X : Type 𝓤i)
    := X → Type 𝓤j
notation `𝒫` X := Power X

/--
`const b` (`↓b`) The constant function at a point.

Given a type `X` and a pointed type `⟨Y, b⟩`, we can consider the constant function which takes every point of `X` to the basepoint `b`.

Such functions are important for the `jensen_inequality`, and are also important in the study of pointed spaces in general.
-/
definition {𝓤x 𝓤y} const {X : Type 𝓤x} {Y : Type 𝓤y}
    := λ b:Y, (λ x:X, b)
notation `↓`:max y:max := const y

section difference_domain --————————————————————————————————————————————————————————————--
universes 𝓤y
variables (Y : Type 𝓤y)

/--
`DifferenceDomain Y extends has_zero Y, has_sub Y` A good place to do subtraction.

A `DifferenceDomain` is a minimal structure that admits a version of subtraction like the one we are familiar with from the theory of abelian groups. The main property that distingushes subtraction is that image of the diagonal always vanishes. This property is called `vanishing_diagonal` below.

There are other axioms which would make sense to add like the following two:
  - `zero_is_right_id  : Π y, y - 0 = y`
  - `sub_associativity : Π a b c, a - (b - c) = (a - b) - (0 - c)`

which makes the subtraction more like the inverse of some addition operation like
`a + b := a - (0 - b)`, but such details are not considered here. For the
`jensen_inequality`, we need only the `vanishing_diagonal` property.
-/
class DifferenceDomain extends has_zero Y, has_sub Y
    := (vanishing_diagonal : Π y, y - y = zero)

/--
`OrderedDifferenceDomain Y extends has_le Y, DifferenceDomain Y`

We will need an order structure on `Y` to state and prove the `jensen_inequality` so we add it here. There are no asumptions on the behavior of `≤` like reflexivity/transitivity.
-/
class OrderedDifferenceDomain
    extends has_le Y, DifferenceDomain Y
```

```
end difference_domain --————————————————————————————————————————————————————--

section reduction --————————————————————————————————————————————————————————--
universes 𝓤y 𝓤x
variables {Y : Type 𝓤y} {X : Type 𝓤x} (𝓕 : 𝒫 (X → Y))

/--
`Reduction` A generalized functional.

Given types `X` and `Y`, a `Reduction` is a partial function from the type of functions
`X → Y` to the type `Y`. The domain of this function is denoted `𝓕` throughout. One reads
the definition of reduction as "a type which takes a function `f : X → Y` and a proof that
`f` is contained in the family `𝓕` (`p : 𝓕 f`, see `Power` above) and sends this pair to
term of type `Y`".

Examples of reductions:
  - evaluation at a point (`x₀ : X`, `eval : (X → Y) → Y := λ f, f x₀`)
      (`𝓕` can be taken to be the total or "everywhere true" predicate)
  - integration (`∫_Ω (—) dμ : (X → Y) → Y`)
      (`𝓕` is the appropriate subset corresponding to the integrable functions on `Ω`)
  - limit of a sequence (`lim : (ℕ → R) → R`)
      (`𝓕` is the appropriate subset corresponding to the convergent sequences)

For the `jensen_inequality` we will be using something like the second example as we want
to imitate a weak kind of integration.

For a fixed reduction we call a function `f` *reducible* if it is contained in the family
corresponding to that reduction, i.e. `reducible f := 𝓕 f`. We say that a reduction
*admits (a reduction of) a function `f`* if `f` is reducible with respect to the family
corresponding to the given reduction.
-/
definition Reduction
    := Π f, 𝓕 f → Y

/--
`left_closed_at 𝓕 f g` Left closedness of `g` at `f`.

We say that a function `g : Y → Y` is *left closed at `f` with respect to `𝓕`* when the
reducibility of `f` implies the reducibility of the composition `g ∘ f` with respect to
the fixed family `𝓕`.
-/
definition left_closed_at (f : X → Y) (g : Y → Y)
    := 𝓕 f → 𝓕 (g ∘ f)

/--
`left_closed 𝓕 g` Left closedness of `g`.

We say that a function `g : Y → Y` is *left closed with respect to `𝓕`* when it is left
closed at every function `f : X → Y`.
-/
definition left_closed (g : Y → Y)
    := Π f, left_closed_at 𝓕 f g

/--
`PointFamily` Family of functions which contains all constant functions.
```

```
A reasonable reduction family might admit constant functions as a trivial case of some
more interesting property.
-/
class PointFamily
    := (has_constants : Π y, 𝓕 ↓y)

section subtraction --————————————————————————————————————————————————————————————--

/--
`pointwise_subtraction` Lifting codomain subtraction to pointwise subtraction.

If `Y` has a subtraction structure then for any type `X`, the function space `X → Y` has a
canonical pointwise subtraction. This `instance` is added to simplify the notation below.
-/
instance pointwise_subtraction [has_sub Y] : has_sub (X → Y)
    := ⟨λ f g, (λ x, f x - g x)⟩

/--
`DifferenceFamily` Family of functions closed under pointwise subtraction.
-/
class DifferenceFamily [has_sub Y]
    := (closure : Π f g (f𝓕 : 𝓕 f) (g𝓕 : 𝓕 g), 𝓕 (f - g))

--————————————————————————————————————————————————————————————————————————————————--
/--
We fix `Int : Reduction 𝓕` which will be written symbolically `∫` to conote something like
an integral. This reduction will be used below.
-/
variables (Int : Reduction 𝓕)

/--
`left_factors 𝓕 β lc_β` Left factorizability of `β`.

We say that a left closed function `β` *left factors with respect to `Int`* when for every
reducible `f` we have the identity `∫ (β ∘ f) = β (∫ f)`.
-/
definition left_factors (β : Y → Y) (lc_β : left_closed 𝓕 β)
    := Π f (f𝓕 : 𝓕 f), Int (β ∘ f) (lc_β f f𝓕) = β (Int f f𝓕)

--————————————————————————————————————————————————————————————————————————————————--
/-- For the rest of this section we assume `𝓕` is a `PointFamily`. -/
variables [PointFamily 𝓕]

/--
`UnitalReduction` A reduction which is friendly to constant functions.

We say that a reduction is unital if it admits all constant functions and that the
reduction of a constant function is the constant which defines it, i.e. `∫ ↓y = y`.
-/
class UnitalReduction
    := (constant_reduction : Π y, Int ↓y (PointFamily.has_constants 𝓕 y) = y)

--————————————————————————————————————————————————————————————————————————————————--
/-- For the rest of this section we assume `Y` has a `≤` structure. -/
variables [has_le Y]
```

```
/--
`pointwise_le` Lifting codomain order to pointwise order.

If `Y` has an order structure `≤` then for any type `X`, the function space `X → Y` has a
canonical pointwise order. This `instance` is added to simplify the notation below.
-/
instance pointwise_le : has_le (X → Y)
    := ⟨λ f g, (Π x, f x ≤ g x)⟩

/--
`MonotonicReduction Y` A reduction which is functorial over `≤`.

We can also make a reduction functorial over the order structure on the space of
functions `X → Y` (restricted to `𝓕`) by requiring that `Int` be a monotonic operator.
-/
class MonotonicReduction
    := (monotonicity : Π f g {f𝓕 g𝓕}, (f ≤ g) → (Int f f𝓕 ≤ Int g g𝓕))

--————————————————————————————————————————————————————————————————————————————————————--
/--
For the rest of this section we assume `Y` has a zero element and subtraction structure
and that `𝓕` is a `DifferenceFamily`. We want to consider a reduction that interacts with
a subtraction structure on its codomain.
-/
variables [has_zero Y] [has_sub Y] [DifferenceFamily 𝓕]

/--
`constant_difference_property` A weak homomorphism property.

For a reduction compatible with subtraction, we would like to have that reductions are
homomorphisms of difference domains but this turns out to be too strong of a property. In
the case of classical integration we do not have in general the property
`∫ (g - f) = (∫ g) - (∫ f)` since both integrals on the right may take "value" at `±∞` so
the subtraction is not well defined. We instead use the weaker property that we can
commute with subtraction of a constant function, this will be enough for our purposes.
-/
definition constant_difference_property
    := Π f k {f𝓕},
        Int (f - ↓k) (DifferenceFamily.closure f ↓k f𝓕 (PointFamily.has_constants 𝓕 k))
      = Int f f𝓕 - Int ↓k (PointFamily.has_constants 𝓕 k)

/--
`translation_invariance_property` A weak translation property across inequalities.

For a reduction compatible with subtraction, we would like to capture the property from
the classical integral that the integral of a difference `g - f` is in the positive cone,
then `∫ f ≤ ∫ g`.
-/
definition translation_invariance_property
    := Π f g {f𝓕 g𝓕}, 0 ≤ Int (g - f) (DifferenceFamily.closure g f g𝓕 f𝓕)
                         → Int f f𝓕 ≤ Int g g𝓕

/--
`TranslativeReduction` A reduction which is compatible with subtraction.
```

```
We call a reduction a `TranslativeReduction` when it satisfies the
`constant_difference_property` and the `translation_invariance_property` defined above.
-/
class TranslativeReduction
    := (constant_difference    : constant_difference_property 𝓕 Int)
       (translation_invariance : translation_invariance_property 𝓕 Int)

end subtraction --————————————————————————————————————————————————————————————————--
end reduction --—————————————————————————————————————————————————————————————————————--

section subdifferential --———————————————————————————————————————————————————————————--
universes 𝒰y
variables {Y : Type 𝒰y} [has_zero Y] [has_sub Y] [has_le Y]
          (ϕ : Y → Y) (t : Y) (𝒩 : 𝒫 Y)

/--
`SubDifferential ϕ t 𝒩` The subdifferential property.

The primary role of convexity in the classical inequality is the following. Given a convex
function `ϕ` on `(a,b)` we know that for any `s,t,u ∈ (a,b)` such that `s < t < u`, it
follows that
    `(ϕ t - ϕ s) / (t - s) ≤ (ϕ u - ϕ t) / (u - t)`
From this we can deduce that there exists the supremum
    `β := sup_{s ∈ (a,t)} (ϕ t - ϕ s) / (t - s)`
which then gives us the following inequality:
    `β · (s - t) ≤ ϕ s - ϕ t`

To generalize this to a domain of discourse that does not have convex functions, we
instead accept the last inequality by hypothesis and generalize multiplication by `β` from
the left to function application. To ensure that this function application behaves well
with zero we need that it vanish there which is analogous to the fact that `y · 0 = 0` in
the real numbers.

Since the inequality above holds only in `(a,b)` in the classical case, not on all of `ℝ`,
we need to restrict the inequality to a specific subset of `Y`. We call this subset `𝒩`.
-/
structure SubDifferential
    := (map                  : Y → Y)
       (root_at_zero         : map 0 = 0)
       (lower_bound_property : Π s, 𝒩 s → map (s - t) ≤ (ϕ s) - (ϕ t))

--————————————————————————————————————————————————————————————————————————————————--
universes 𝒰x
variables {X : Type 𝒰x} (𝓕 : 𝒫 (X → Y)) (Int : Reduction 𝓕)

/--
`LeftFactorSubDifferential ϕ t 𝒩` A left factorizable subdifferential.

If we have `β` a subdifferential corresponding to `ϕ`, `t`, and `𝒩`, then we say that it
is a *left factor subdifferential* when `β` left factors with respect to the family `𝓕`.
-/
structure LeftFactorSubDifferential extends SubDifferential ϕ t 𝒩
    := (is_left_closed : left_closed 𝓕 map)
       (left_factors   : left_factors 𝓕 Int map is_left_closed)
```

```
end subdifferential --————————————————————————————————————————————————————--

section jensen_inequality --————————————————————————————————————————————————--
universes 𝒰y 𝒰x
variables {Y : Type 𝒰y} [OrderedDifferenceDomain Y]
          {X : Type 𝒰x}
          (𝓕 : 𝒫 (X → Y)) [PointFamily 𝓕] [DifferenceFamily 𝓕]
          (Int : Reduction 𝓕)

/--
`JensenReduction` A reduction which is strong enough to prove Jensen's Inequality.

We have the following properties inherited from the individual structures:
  - `UnitalReduction`
      - `∫ ↓t = t`
  - `MonotonicReduction`
      - `(Π x, f x ≤ g x) → (∫ f ≤ ∫ g)`
  - `TranslativeReduction`
      - `∫ (f - ↓k) = (∫ f) - (∫ ↓k)`
      - `(0 ≤ ∫ (g - f)) → (∫ f ≤ ∫ g)`
-/
class JensenReduction
    extends UnitalReduction      𝓕 Int,
            MonotonicReduction   𝓕 Int,
            TranslativeReduction 𝓕 Int

/--
`jensen_inequality` Jensen's Inequality (`ϕ (∫ f) ≤ ∫ (ϕ ∘ f)`).

The theorem exactly generalizes the classical Jensen inequality and so it is sufficient to
prove that the measure-theoretic integral is a `JensenReduction` and that convex functions
have the `LeftFactorSubDifferential` property to apply the following proof to the
classical case. Proof sketches for these two facts are discussed above.

NB: In this proof, we will be working to find a term which has the type of the goal
    instead of modifying the goal directly, so it is not necessary to look at the goals.
    Instead, follow the proof by reading the commentary which describes the change in the
    main term `inequality` which will satisfy the goal in the end. For simplicity the
    reducibility proofs are left unspecified while following the main term.
-/
theorem jensen_inequality
    -- Proof that Int is a nice reduction.
    [JensenReduction 𝓕 Int]

    -- A reducible function f.
    (f : X → Y) (f𝓕 : 𝓕 f)

    -- A distinguished superset of the image of f.
    (𝒩 : 𝒫 Y) (image_contained_in𝒩 : Π x, 𝒩 (f x))

    -- A function ϕ which is left closed at f and has a LeftFactorSubDifferential at the
    -- reduction of f with the above distinguished superset 𝒩.
    (ϕ : Y → Y) (ϕf𝓕 : 𝓕 (ϕ ∘ f))
    (subdifferential : LeftFactorSubDifferential ϕ (Int f f𝓕) 𝒩 𝓕 Int)

    -- From above, the Jensen Inequality follows:
```

```
    : ϕ (Int f f𝓕) ≤ Int (ϕ ∘ f) ϕf𝓕

:= begin -- We begin by introducing some relevant variables:

    -- The reduction of f.
    let t := Int f f𝓕,

    -- The function F := λ x, f x - t and its proof of reducibility.
    let F := f - ↓t,
    let F𝓕 := DifferenceFamily.closure f ↓t _ _,

    -- The subderivative of ϕ.
    let β := subdifferential.map,

    /-
    Π s, 𝒩 s → β (s - t) ≤ (ϕ s) - (ϕ t)
    ————————————————————————————————————— image_contained_in_𝒩
       Π x, β (F x) ≤ (ϕ (f x)) - (ϕ t)

    To begin the proof, we use the fact that the subdifferential has the lower bound
    property inside of 𝒩 which contains the image of f, so we have that the inequality
    holds for all points of X. This is the main term to follow throughout the proof.
    -/
    let inequality
        := λ x, subdifferential.lower_bound_property (f x) (image_contained_in_𝒩 x),

    /-
    Π x, β (F x) ≤ (ϕ (f x)) - (ϕ t)
    ———————————————————————————————— monotonicity
      ∫ β ∘ F ≤ ∫ (ϕ ∘ f - ↓(ϕ t))

    Next, we use the fact that the reduction is monotonic to "integrate" both sides.
    -/
    let inequality
        := MonotonicReduction.monotonicity Int (β ∘ F) (ϕ ∘ f - ↓(ϕ t)) inequality,

    /-
    ∫ β ∘ F ≤ ∫ (ϕ ∘ f - ↓(ϕ t))
    ———————————————————————————— left_factors
    β (∫ F) ≤ ∫ (ϕ ∘ f - ↓(ϕ t))

    Here we use the fact that the subdifferential left factors with respect to the
    reduction family.
    -/
    rewrite subdifferential.left_factors F F𝓕 at inequality,

    /-
         β (∫ F) ≤ ∫ (ϕ ∘ f - ↓(ϕ t))
    ———————————————————————————————————— constant_difference
    β ((∫ f) - (∫ ↓t)) ≤ ∫ (ϕ ∘ f - ↓(ϕ t))

    Now we use the fact that our reduction has the constant difference property to
    distribute the reduction over the difference of functions.
    -/
    rewrite TranslativeReduction.constant_difference Int at inequality,
```

```
/-
β ((∫ f) - (∫ ↓t)) ≤ ∫ (ϕ ∘ f - ↓(ϕ t))
————————————————————————————————————— constant_reduction
  β ((∫ f) - t) ≤ ∫ (ϕ ∘ f - ↓(ϕ t))

Since our reduction is unital we can replace the reduction of the constant function
with the defining constant.
-/
rewrite UnitalReduction.constant_reduction Int at inequality,

/-
β ((∫ f) - t) ≤ ∫ (ϕ ∘ f - ↓(ϕ t))
————————————————————————————————— vanishing_diagonal
    β 0 ≤ ∫ (ϕ ∘ f - ↓(ϕ t))

Since
    t := ∫ f
we have that,
    (∫ f) - t := (∫ f) - (∫ f)
so we can cancel like terms using the fact that Y is a difference domain.
-/
rewrite DifferenceDomain.vanishing_diagonal at inequality,

/-
β 0 ≤ ∫ (ϕ ∘ f - ↓(ϕ t))
————————————————————————— root_at_zero
 0 ≤ ∫ (ϕ ∘ f - ↓(ϕ t))

To simplify the left hand side of the inequality we use the fact that the
subderivative has a root at zero.
-/
rewrite subdifferential.root_at_zero at inequality,

/-
0 ≤ ∫ (ϕ ∘ f - ↓(ϕ t))
————————————————————— translation_invariance
 ∫ ↓(ϕ t) ≤ ∫ (ϕ ∘ f)

To split the reduction of a difference, we use the translation invariance property
because our reduction is translative.
-/
let inequality
    := TranslativeReduction.translation_invariance Int ↓(ϕ t) (ϕ ∘ f) inequality,

/-
∫ ↓(ϕ t) ≤ ∫ (ϕ ∘ f)
——————————————————— constant_reduction
  ϕ t ≤ ∫ (ϕ ∘ f)

Again we use the fact that out reduction is unital to pull out the constant on the
left hand side.
-/
rewrite UnitalReduction.constant_reduction Int at inequality,

/-
  ϕ t ≤ ∫ (ϕ ∘ f)
```

```
    ──────────────────── definition
    φ (∫ f) ≤ ∫ (φ ∘ f)

    And now the proof is complete. The inequality has the correct type to satisfy the main
    goal. All other goals can be deduced canonically by the proof assistant since the
    relevant proof terms are in the given context.
    -/
    exact inequality,
end --                                                                                     □
end jensen_inequality --─────────────────────────────────────────────────────────────────--
```