Duality theory in linear optimization and its extensions — formally verified

Martin Dvorak, Vladimir Kolmogorov

2024

Abstract: Farkas established that a system of linear inequalities has a solution if and only if we cannot obtain a contradiction by taking a linear combination of the inequalities. We state and formally prove several Farkas-like theorems in Lean 4. Furthermore, we consider a linearly ordered field extended with two special elements denoted by $\bot$ and $\top$ where $\bot$ is below every element and $\top$ is above every element. We define $\bot + a = \bot = a + \bot$ for all a and we define $\top + b = \top = b + \top$ for all $b \neq \bot$. Instead of multiplication, we define scalar action $c \cdot \bot = \bot$ for every $c \geq 0$ but we define $d \cdot \top = \top$ only for $d > 0$ because $0 \cdot \top = 0$. We extend certain Farkas-like theorems to a setting where coefficients are from an extended linearly ordered field.

1 Introduction

We do not use set theory. We write everything in type theory. Juxtaposition never denotes multiplication. Juxtaposition is always function application. We often talk about linearly ordered fields. The two main examples are rationals and reals. The paper starts with famous corollaries and then proceeds to state more general theorems. Proofs are postponed to respective sections. Theorems come with names instead of numbers.

There are two famous theorems characterizing that a system of linear inequalities has a solution if and only if we cannot obtain a contradiction by taking a linear combination of the inequalities. TODO cite Farkas.

Theorem (equalityFarkas): Let I and J be finite types. Let F be a linearly ordered field. Let A be a matrix of type $(I \times J) \to F$. Let b be a vector of type $I \to F$. Exactly one of the following exists:

- nonnegative vector $x : J \to F$ such that $A \cdot x = b$
- vector $y : I \to F$ such that $A^T \cdot y \geq 0$ and $b \cdot y < 0$

TODO cite Minkowski?

Theorem (inequalityFarkas): Let I and J be finite types. Let F be a linearly ordered field. Let A be a matrix of type $(I \times J) \to F$. Let b be a vector of type $I \to F$. Exactly one of the following exists:

- nonnegative vector $x : J \to F$ such that $A \cdot x \leq b$
- nonnegative vector $y : I \to F$ such that $A^T \cdot y \geq 0$ and $b \cdot y < 0$

TODO cite John von Neumann? (any maybe Oskar Morgenstern and maybe George Dantzig and maybe Albert W. Tucker?)

Theorem (StandardLP.strongDuality): Let I and J be finite types. Let F be a linearly ordered field. Let A be a matrix of type $(I \times J) \to F$. Let b be a vector of type $I \to F$. Let c be a vector of type $J \to F$. Then

$$\min \{\, c \cdot x \mid x \geq 0 \ \wedge \ A \cdot x \leq b \,\} = -\min \{\, b \cdot y \mid y \geq 0 \ \wedge \ A^T \cdot y \leq c \,\}$$

holds if at least one of the systems has a solution (very roughly paraphrased).

1.1 Generalizations

The next theorem generalizes equalityFarkas to structures where multiplication does not have to be commutative. Furthermore, it supports infinitely many equations.

Theorem (coordinateFarkas): Let I be any type. Let J be a finite type. Let R be a linearly ordered division ring. Let A be an R-linear map from from $(I \to R)$ to $(J \to R)$. Let b be an R-linear map from from $(I \to R)$ to R. Exactly one of the following exists:

- nonnegative vector $x : J \to R$ such that, for all $w : I \to R$, we have $\sum_{j:J} (A\ w)_j \cdot x_j = b\ w$
- vector $y : I \to R$ such that $A\ y \geq 0$ and $b\ y < 0$

In the next generalization, we replace the partially ordered module $I \to R$ by a general R-module W.

Theorem (scalarFarkas): Let J be a finite type. Let R be a linearly ordered division ring. Let W be an R-module. Let A be an R-linear map from from W to $(J \to R)$. Let b be an R-linear map from from W to R. Exactly one of the following exists:

- nonnegative vector $x : J \to R$ such that, for all $w : W$, we have $\sum_{j:J} (A\ w)_j \cdot x_j = b\ w$

- vector $y : W$ such that $A\ y \geq 0$ and $b\ y < 0$

In the most general theorem, stated below, we replace certain occurrences of R by a linearly ordered R-module V whose order respects order on R. This result origins from TODO cite Bartl. TODO where did PosSMulMono go?!

Theorem (fintypeFarkasBartl): Let J be a finite type. Let R be a linearly ordered division ring. Let W be an R-module. Let V be a linearly ordered R-module. Let A be an R-linear map from W to $(J \to R)$. Let b be an R-linear map from W to V. Exactly one of the following exists:

- nonnegative vector family $x : J \to V$ such that, for all $w : W$, we have $\sum_{j:J} (A\ w)_j \cdot x_j = b\ w$
- vector $y : W$ such that $A\ y \geq 0$ and $b\ y < 0$

In the last branch, $A\ y \geq 0$ uses the partial order on $(J \to R)$ whereäs $b\ y < 0$ uses the linear order on V. Note that fintypeFarkasBartl subsumes scalarFarkas (as well as the other versions based on equality), since R can be viewed as a linearly ordered module over itself. We prove fintypeFarkasBartl in Section 3, which is where the heavy lifting comes.

1.2 Extensions

Until now, we have talked about known results. What follows is a new extension of the theory.

Definition: Let F be a linearly ordered field. We define an **extended** linearly ordered field F_∞ as $F \cup \{\bot, \top\}$ with the following properties. Let p and q be numbers from F. We have $\bot < p < \top$. We define addition, scalar action, and negation on F_∞ as follows:

$+$	$\bot$	q	$\top$
$\bot$	$\bot$	$\bot$	$\bot$
p	$\bot$	$p+q$	$\top$
$\top$	$\bot$	$\top$	$\top$

$\cdot$	$\bot$	q	$\top$
0	$\bot$	0	0
$p > 0$	$\bot$	$p \cdot q$	$\top$

$-$	$\bot$	q	$\top$
	$\top$	$-q$	$\bot$

When we talk about elements of F_∞, we say that values from F are **finite**.

Informally speaking, $\top$ represents the positive infinity, $\bot$ represents the negative infinity, and we say that $\bot$ is "stronger" than $\top$ in all arithmetic operations. The surprising parts are $\bot + \top = \bot$ and $0 \cdot \bot = \bot$. Because of them, F_∞ is not a field. In fact, F_∞ is not even a group. However, F_∞ is still a densely linearly ordered abelian monoid with characteristic zero.

Theorem (extendedFarkas): Let I and J be finite types. Let F be a linearly ordered field. Let A be a matrix of type $(I \times J) \to F_\infty$. Let b be a vector of type $I \to F_\infty$. Assume that A does not have $\bot$ and $\top$ in the same row. Assume that A does not have $\bot$ and $\top$ in the same column. Assume that A does not have $\top$ in any row where b has $\top$. Assume that A does not have $\bot$ in any row where b has $\bot$. Exactly one of the following exists:

- nonnegative vector $x : J \to F$ such that $A \cdot x \leq b$
- nonnegative vector $y : I \to F$ such that $(-A^T) \cdot y \leq 0$ and $b \cdot y < 0$

Next we define an extended notion of linear program, i.e., linear programming over extended linearly ordered fields. The implicit intention is that the linear program is to be minimized.

Definition: Let I and J be finite types. Let F be a linearly ordered field. Let A be a matrix of type $(I \times J) \to F_\infty$, let b be a vector of type $I \to F_\infty$, and c be a vector of type $J \to F_\infty$ such that the following six conditions hold:

- A does not have $\bot$ and $\top$ in the same row
- A does not have $\bot$ and $\top$ in the same column
- b does not contain $\bot$
- c does not contain $\bot$
- A does not have $\top$ in any row where b has $\top$
- A does not have $\bot$ in any column where c has $\top$

We say that $P = (A, b, c)$ is a **linear program** over F_∞ whose constraints are indexed by I and variables are indexed by J. We say that a nonnegative vector $x : J \to R$ is a **solution** to P if and only if $A \cdot x \leq b$. We say that P **reaches** an objective value r if and only if there exists x such that x is a solution to P and $c \cdot x = r$. We say that P is **feasible** if and only if P reaches a finite[1] value. We say that P is **bounded by** a finite value r if and only if, for every value p reached by P, we have $r \leq p$. We say that P is **unbounded** if and only if there is no finite value r such that P is bounded by r. We say that the linear program $(-A^T, c, b)$ is the **dual** of P.

Theorem (ExtendedLP.weakDuality): Let F be a linearly ordered field. Let P be a linear program over F_∞. If P reaches p and the dual of P reaches q, then $p + q \geq 0$.

[1] It would be perhaps more natural to say that P reaches a value different from $\top$. However, since $\bot$ cannot be reached because of the way linear programming is defined, it is equivalent to our definition by reaching a finite value.

Definition: Let F be a linearly ordered field. Let P be a linear program over F_∞. We define the **optimum** of P as follows. If P is feasible and unbounded, its optimum is $\bot$. If P is not feasible, its optimum is $\top$. In all other cases, we ask whether P reaches a finite value r such that P is bounded by r. If so, its optimum is r. Otherwise, P does not have optimum.[2]

Theorem (ExtendedLP.strongDuality):[3] Let F be a linearly ordered field. Let P be a linear program over F_∞. If P or its dual is feasible (at least one of them), then there exists p in F_∞ such that P has optimum p and the dual of P has optimum $-p$.

1.3 Structure of this paper

In Section 2, we explain all underlying definitions and comment on the formalization process; following the philosophy of (TODO cite How to believe a formal proof) we review almost all the declarations needed for the reader to believe our results, leaving out many declarations that were used in order to prove the results. In Section 3, we prove fintypeFarkasBartl (stated in Section 1.1), from which be obtain equalityFarkas as a corollary. In Section 4, we prove extendedFarkas (stated in Section 1.2). In Section 5, we prove ExtendedLP.strongDuality (stated in Section 1.2), from which we obtain StandardLP.strongDuality as a corollary. TODO autogenerate correct section numbers.

Repository `https://github.com/madvorak/duality` contains full version of all definitions, statements, and proofs. They are written in a formal language called Lean 4, which provides a guarantee that every step of every proof follows from valid logical axioms.[4] This paper attempts to be an accurate description of the `duality` project. However, in case of any discrepancy, the code shall prevail.

2 Preliminaries

There are many layers of definitions that are built before say what a linearly ordered field is, what a linearly ordered division ring is, and what linearly ordered abelian group is. We need these three algebraic structures for stating our main results. This section mainly documents existing Mathlib definitions. We also highlight custom definitions added to our project only.

2.1 We start with a review of algebraic typeclasses that our project depends on

Additive semigroup is a structure on any type with addition where the addition is associative:

```
class AddSemigroup (G : Type u) extends Add G where
  add_assoc : ∀ a b c : G, (a + b) + c = a + (b + c)
```

Similarly, semigroup is a structure on any type with multiplication where the multiplication is associative:

```
class Semigroup (G : Type u) extends Mul G where
  mul_assoc : ∀ a b c : G, (a * b) * c = a * (b * c)
```

Additive monoid is an additive semigroup with the "zero" element that is neutral with respect to addition from both left and right, thanks to which we can define a scalar multiplication by the natural numbers:

```
class AddZeroClass (M : Type u) extends Zero M, Add M where
  zero_add : ∀ a : M, 0 + a = a
  add_zero : ∀ a : M, a + 0 = a
class AddMonoid (M : Type u) extends AddSemigroup M, AddZeroClass M where
  nsmul : ℕ → M → M
  nsmul_zero : ∀ x : M, nsmul 0 x = 0
  nsmul_succ : ∀ (n : ℕ) (x : M), nsmul (n + 1) x = nsmul n x + x
```

Similarly, monoid is a semigroup with the "one" element that is neutral with respect to multiplication from both left and right, thanks to which we can define a power to the natural numbers:

```
class MulOneClass (M : Type u) extends One M, Mul M where
  one_mul : ∀ a : M, 1 * a = a
  mul_one : ∀ a : M, a * 1 = a
class Monoid (M : Type u) extends Semigroup M, MulOneClass M where
  npow : ℕ → M → M
  npow_zero : ∀ x : M, npow 0 x = 1
  npow_succ : ∀ (n : ℕ) (x : M), npow (n + 1) x = npow n x * x
```

Subtractive monoid is an additive monoid that adds two more operations (unary and binary minus) that satisfy some basic properties:

```
class SubNegMonoid (G : Type u) extends AddMonoid G, Neg G, Sub G where
  sub := SubNegMonoid.sub'
  sub_eq_add_neg : ∀ a b : G, a - b = a + -b
  zsmul : ℤ → G → G
  zsmul_zero' : ∀ a : G, zsmul 0 a = 0
```

[2] By the end of the paper, we will have proved that optimum always exists, i.e., it cannot happen that the set of objective values reached by P has a finite infimum that is not attained. However, because we do not have the theorem now, the optimum is defined as a partial function from linear programs to F_∞.

[3] For simplicity, we rephrased the theorem without mentioning partial functions. Also note that it would be incorrect to say the following: P has optimum p, the dual of P has optimum q, and $p + q = 0$. It would fail for unbounded linear programs because the arithmetics of F_∞ defines $\top + \bot = \bot$.

[4] The only axioms used in our proofs are propext, Classical.choice, and Quot.sound, which you can check by the `#print axioms` command.

```
  zsmul_succ' (n : ℕ) (a : G) : zsmul (Int.ofNat n.succ) a = zsmul (Int.ofNat n) a + a
  zsmul_neg' (n : ℕ) (a : G) : zsmul (Int.negSucc n) a = -(zsmul n.succ a)
```

Similarly, division monoid is a monoid that adds two more operations (inverse and divide) that satisfy some basic properties:

```
class DivInvMonoid (G : Type u) extends Monoid G, Inv G, Div G where
  div := DivInvMonoid.div'
  div_eq_mul_inv : ∀ a b : G, a / b = a * b⁻¹
  zpow : ℤ → G → G
  zpow_zero' : ∀ a : G, zpow 0 a = 1
  zpow_succ' (n : ℕ) (a : G) : zpow (Int.ofNat n.succ) a = zpow (Int.ofNat n) a  * a
  zpow_neg' (n : ℕ) (a : G) : zpow (Int.negSucc n) a = (zpow n.succ a)⁻¹
```

Additive group is a subtractive monoid in which the unary minus acts as a left inverse with respect to addition:

```
class AddGroup (A : Type u) extends SubNegMonoid A where
  add_left_neg : ∀ a : A, -a + a = 0
```

Abelian magma is a structure on any type that has commutative addition:

```
class AddCommMagma (G : Type u) extends Add G where
  add_comm : ∀ a b : G, a + b = b + a
```

Similarly, commutative magma is a structure on any type that has commutative multiplication:

```
class CommMagma (G : Type u) extends Mul G where
  mul_comm : ∀ a b : G, a * b = b * a
```

Abelian semigroup is an abelian magma and an additive semigroup at the same time:

```
class AddCommSemigroup (G : Type u) extends AddSemigroup G, AddCommMagma G
```

Similarly, commutative semigroup is a commutative magma and a semigroup at the same time:

```
class CommSemigroup (G : Type u) extends Semigroup G, CommMagma G
```

Abelian monoid is an additive monoid and an abelian semigroup at the same time:

```
class AddCommMonoid (M : Type u) extends AddMonoid M, AddCommSemigroup M
```

Similarly, commutative monoid is a monoid and a commutative semigroup at the same time:

```
class CommMonoid (M : Type u) extends Monoid M, CommSemigroup M
```

Abelian group is an additive group and an abelian monoid at the same time:

```
class AddCommGroup (G : Type u) extends AddGroup G, AddCommMonoid G
```

Distrib is a structure on any type with addition and multiplication where both left distributivity and right distributivity hold:

```
class Distrib (R : Type*) extends Mul R, Add R where
  left_distrib : ∀ a b c : R, a * (b + c) = a * b + a * c
  right_distrib : ∀ a b c : R, (a + b) * c = a * c + b * c
```

Non-unital-non-associative-semiring is an abelian monoid with distributive multiplication and well-behaved zero:

```
class MulZeroClass (M₀ : Type u) extends Mul M₀, Zero M₀ where
  zero_mul : ∀ a : M₀, 0 * a = 0
  mul_zero : ∀ a : M₀, a * 0 = 0
class NonUnitalNonAssocSemiring (α : Type u) extends AddCommMonoid α, Distrib α, MulZeroClass α
```

Non-unital-semiring is a non-unital-non-associative-semiring that forms a semigroup with zero:

```
class SemigroupWithZero (S₀ : Type u) extends Semigroup S₀, MulZeroClass S₀
class NonUnitalSemiring (α : Type u) extends NonUnitalNonAssocSemiring α, SemigroupWithZero α
```

Additive monoid with one and abelian monoid with one are defined defined from additive monoid and abelian monoid, equipped with the symbol "one" and embedding of natural numbers (please ignore the difference between `Type u` and `Type*` as it is the same thing for all our purposes):

```
class AddMonoidWithOne (R : Type*) extends NatCast R, AddMonoid R, One R where
  natCast := Nat.unaryCast
  natCast_zero : natCast 0 = 0
  natCast_succ : ∀ n, natCast (n + 1) = natCast n + 1
class AddCommMonoidWithOne (R : Type*) extends AddMonoidWithOne R, AddCommMonoid R
```

Additive group with one is an additive monoid with one and additive group at the same time:

```
class AddGroupWithOne (R : Type u) extends IntCast R, AddMonoidWithOne R, AddGroup R where
  intCast := Int.castDef
  intCast_ofNat : ∀ n : ℕ, intCast (n : ℕ) = Nat.cast n
  intCast_negSucc : ∀ n : ℕ, intCast (Int.negSucc n) = - Nat.cast (n + 1)
```

Non-associative-semiring is a non-unital-non-associative-semiring that has a well-behaved multiplication by both zero and one and forms an abelian monoid with one:

```
class MulZeroOneClass (M₀ : Type u) extends MulOneClass M₀, MulZeroClass M₀
class NonAssocSemiring (α : Type u) extends NonUnitalNonAssocSemiring α, MulZeroOneClass α,
  AddCommMonoidWithOne α
```

Semiring is the combination of non-unital-semiring and non-associative-semiring that forms a monoid with zero:

```
class MonoidWithZero (M₀ : Type u) extends Monoid M₀, MulZeroOneClass M₀, SemigroupWithZero M₀
class Semiring (α : Type u) extends NonUnitalSemiring α, NonAssocSemiring α, MonoidWithZero α
```

Ring is a semiring and an abelian group at the same time that has "one" that behaves well:

```
class Ring (R : Type u) extends Semiring R, AddCommGroup R, AddGroupWithOne R
```

Commutative ring is a ring (guarantees commutative addition) and a commutative monoid (guarantees commutative multiplication) at the same time.

```
class CommRing (α : Type u) extends Ring α, CommMonoid α
```

Division ring is a ring whose multiplication forms a division monoid, zero is inverse to itself, and (TODO explain NNRatCast and RatCast; give a link to the article that explains why $0^{-1}=0$ is OK):

```
class Nontrivial (α : Type*) : Prop where
  exists_pair_ne : ∃ x y : α, x ≠ y
class DivisionRing (α : Type*) extends Ring α, DivInvMonoid α, Nontrivial α, NNRatCast α, RatCast α where
  mul_inv_cancel : ∀ (a : α), a ≠ 0 → a * a⁻¹ = 1
  inv_zero : (0 : α)⁻¹ = 0
  nnratCast := NNRat.castRec
```

Field is a commutative ring and a division ring at the same time:

```
class Field (K : Type u) extends CommRing K, DivisionRing K
```

Preorder is a reflexive & transitive relation on any structure with binary relational symbols $\le$ and $<$ where the strict comparison $a < b$ is equivalent to $a \le b \wedge \neg(b \le a)$ given by the relation $\le$ which is not necessarily antisymmetric:

```
class Preorder (α : Type u) extends LE α, LT α where
  le_refl : ∀ a : α, a ≤ a
  le_trans : ∀ a b c : α, a ≤ b → b ≤ c → a ≤ c
  lt_iff_le_not_le : ∀ a b : α, a < b ⟺ a ≤ b ∧ ¬b ≤ a
```

Partial order is a reflexive & antisymmetric & transitive relation:

```
class PartialOrder (α : Type u) extends Preorder α where
  le_antisymm : ∀ a b : α, a ≤ b → b ≤ a → a = b
```

Ordered abelian group is an abelian group with partial order that respects addition:

```
class OrderedAddCommGroup (α : Type u) extends AddCommGroup α, PartialOrder α where
  add_le_add_left : ∀ a b : α, a ≤ b → ∀ c : α, c + a ≤ c + b
```

Strictly ordered ring is a nontrivial ring whose addition behaves as an ordered abelian group where zero is less or equal to one and the product of two strictly positive elements is strictly positive:

```
class StrictOrderedRing (α : Type u) extends Ring α, OrderedAddCommGroup α, Nontrivial α where
  zero_le_one : 0 ≤ (1 : α)
  mul_pos : ∀ a b : α, 0 < a → 0 < b → 0 < a * b
```

Linear order (sometimes called total order) is a partial order where every two elements are comparable; technical details are omitted:

```
class LinearOrder (α : Type u) extends PartialOrder α, (...)
  le_total (a b : α) : a ≤ b ∨ b ≤ a
  (...)
```

Linearly ordered abelian group is an ordered abelian group whose order is linear:

```
class LinearOrderedAddCommGroup (α : Type u) extends OrderedAddCommGroup α, LinearOrder α
```

Linearly ordered ring is a strictly ordered ring where every two elements are comparable:

```
class LinearOrderedRing (α : Type u) extends StrictOrderedRing α, LinearOrder α
```

Linearly ordered commutative ring is a linearly ordered ring and commutative monoid at the same time:

```
class LinearOrderedCommRing (α : Type u) extends LinearOrderedRing α, CommMonoid α
```

We define a linearly ordered division ring as a division ring that is a linearly ordered ring at the same time:

```
class LinearOrderedDivisionRing (R : Type*) extends LinearOrderedRing R, DivisionRing R
```

Linearly ordered field is defined in Mathlib as a linearly ordered commutative ring that is a field at the same time:

```
class LinearOrderedField (α : Type*) extends LinearOrderedCommRing α, Field α
```

Note that LinearOrderedDivisionRing is not a part of the algebraic hierarchy provided by Mathlib, hence LinearOrderedField does not inherit LinearOrderedDivisionRing, thus we provide a custom instance that converts LinearOrderedField to LinearOrderedDivisionRing:

```
instance LinearOrderedField.toLinearOrderedDivisionRing {F : Type*} [instF : LinearOrderedField F] :
  LinearOrderedDivisionRing F := { instF with }
```

This instance is needed for the step from coordinateFarkas to equalityFarkas.

2.2 Extended linearly ordered fields

Given any type F, we construct $F \cup \{\bot, \top\}$ as follows:

```
def Extend (F : Type*) := WithBot (WithTop F)
```

From now on we assume that F is a linearly ordered field:

```
variable {F : Type*} [LinearOrderedField F]
```

The following instance defines how addition and comparison behaves on F_∞ and automatically generates a proof that F_∞ forms a linearly ordered abelian monoid:

```
instance : LinearOrderedAddCommMonoid (Extend F) :=
  inferInstanceAs (LinearOrderedAddCommMonoid (WithBot (WithTop F)))
```

The following definition embeds F in F_∞ and registers this canonical embedding as a type coercion:

```
@[coe] def toE : F → (Extend F) := some ∘ some
instance : Coe F (Extend F) := ⟨toE⟩
```

Unary minus on F_∞ is defined as follows:

```
def neg : Extend F → Extend F
| ⊥ => ⊤
| ⊤ => ⊥
| (x : F) => toE (-x)
instance : Neg (Extend F) := ⟨EF.neg⟩
```

In the file `FarkasSpecial.lean` and everything downstream, we have the notation

```
F∞
```

for F_∞ and also the notation

```
F≥0
```

for all nonnegative elements of F. TODO define scalar action and explain SMulZeroClass.

2.3 Vectors and matrices

We distinguish two types of vectors; implicit vectors and explicit vectors. Implicit vectors are members of a vector space; they don't have any internal structure. Explicit vectors are functions from coordinates to values. The set of coordinates needn't be ordered. Matrices live next to explicit vectors. They are also functions; they take a row index and a column index and they output a value at the given spot. Neither the row indices nor the column vertices are required to form an ordered set. That's why multiplication between matrices and vectors is defined only in structures where addition forms a commutative semigroup. Consider the following example:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \cdot \begin{pmatrix} 7 \\ 8 \\ 9 \end{pmatrix} = \begin{pmatrix} ? \\ - \end{pmatrix}$$

We don't know whether the value at the question mark is equal to $(1 \cdot 7 + 2 \cdot 8) + 3 \cdot 9$ or to $(2 \cdot 8 + 1 \cdot 7) + 3 \cdot 9$ or to any other ordering of summands. This is why commutativity of addition is necessary for the definition to be valid. On the other hand, we don't assume any property of multiplication in the definition of multiplication between matrices and vectors; they don't even have to be of the same type; we only require the elements of the vector to have an action on the elements of the matrix (this is not a typo — normally, we would want matrices to have an action on vectors — not in our work).

2.3.1 Mathlib definitions versus our definitions

Mathlib defines dot product (i.e., a product of an explicit vector with an explicit vector) as follows:

```
def Matrix.dotProduct [Fintype m] [Mul α] [AddCommMonoid α] (v w : m → α) : α :=
  ∑ i, v i * w i
```

Mathlib defines a product of a matrix with an explicit vector as follows:

```
def Matrix.mulVec [Fintype n] [NonUnitalNonAssocSemiring α] (M : Matrix m n α) (v : n → α) : m → α
  | i => (fun j => M i j) ⬝ᵥ v
```

Mildly confusingly, `m` and `n` are type variables here, not natural numbers. Note that, when multiplying two vectors, the left vector is not transposed — a vector is not defined as a special case of matrix, and transposition is not defined as an operation on explicit vectors, only on matrices.

Infix notation for these two operations is defined as follows (the keyword `infixl` when defining an operator $\circ$ means that `x ∘ y ∘ z` gets parsed as `(x ∘ y) ∘ z` whether or not it makes sense, whereäs the keyword `infixr` when defining an operator $\circ$ means that `A ∘ B ∘ v` gets parsed as `A ∘ (B ∘ v)` whether or not it makes sense; numbers 72 and 73 determine precedence):

```
scoped infixl:72 " ⬝ᵥ " => Matrix.dotProduct
scoped infixr:73 " *ᵥ " => Matrix.mulVec
```

These definitions are sufficient for stating results based on linearly ordered fields. However, our results from Section 1.3 require a more general notion of multiplication between matrices and vectors. They are defined in the section hetero_matrix_products_defs as follows:

```
variable {α γ : Type*} [AddCommMonoid α] [SMul γ α]

def Matrix.dotProd (v : I → α) (w : I → γ) : α :=
  ∑ i : I, w i • v i
infixl:72 " ᵥ⬝ " => Matrix.dotProd

def Matrix.mulWeig (M : Matrix I J α) (w : J → γ) (i : I) : α :=
  M i ᵥ⬝ w
infixr:73 " ₘ* " => Matrix.mulWeig
```

We start by declaring that α and γ are types from any universe (not necessarily both from the same universe). We require that α forms an abelian monoid and that γ has a scalar action on α. In this setting, we can instantiate α with F_∞ and γ with F for any linearly ordered field F.

For explicit vectors $v : I \to \alpha$ and $w : I \to \gamma$, we define their product of type α as follows. Every element of v gets multiplied from left by an element of w on the same index. Then we sum them all together (in unspecified order). For a matrix $M : (I \times J) \to \alpha$ and a vector $w : J \to \gamma$, we define their product of type $I \to \alpha$ as a function that takes an index i and outputs the dot product between the i-th row of M and the vector w.

Beware that the arguments (both in the function definition and in the infix notation) come in the opposite order from how scalar action is written. We recommend a mnemonic "vector times weights" for $v \cdot w$ and "matrix times weights" for $M \cdot w$ where arguments come in alphabetical order.

In the infix notation, you can distinguish between the standard Mathlib definitions and our definitions by observing that Mathlib operators put the letter $_v$ to the right of the symbol whereäs our operators put a letter to the left of the symbol.

We find it unfortunate that the Mathlib name of `dotProduct` is prefixed by the `Matrix` namespace. While `Matrix.mulVec` allows for the use of dot notation in places where infix notation is not used, the full name `Matrix.dotProduct` only clutters the code. Even though we don't like it, we decided to namespace our definitions in the same way for consistency.

Since we have new definitions, we have to rebuild all API (a lot of lemmas) for `Matrix.dotProd` and `Matrix.mulWeig` from scratch. This process was very tiresome. We decided not to develop a full reusable library, but prove only those lemmas we wanted to use in our project. For similar reasons, we did not generalize the Mathlib definition of "vector times matrix", as "matrix times vector" was all we needed. It was still a lot of lemmas.

2.4 Modules and how to order them

Given types α and β such that α has a scalar action on β and α forms a monoid, Mathlib defines multiplicative action where 1 of type α gives the identity action and multiplication in the monoid associates with the scalar action:

```
class MulAction (α : Type*) (β : Type*) [Monoid α] extends SMul α β where
  one_smul : ∀ b : β, (1 : α) • b = b
  mul_smul : ∀ (x y : α) (b : β), (x * y) • b = x • y • b
```

For a distributive multiplicative action, we furthermore require the latter type to form an additive monoid and two more properties are required; multiplying the zero gives the zero, and the multiplicative action is distributive with respect to the addition:

```
class DistribMulAction (M A : Type*) [Monoid M] [AddMonoid A] extends MulAction M A where
  smul_zero : ∀ a : M, a • (0 : A) = 0
  smul_add : ∀ (a : M) (x y : A), a • (x + y) = a • x + a • y
```

We can finally review the definition of a module. Here the former type must form a semiring and the latter type abelian monoid. Module requires a distributive multiplicative action and two additional properties; addition in the semiring distributes with the multiplicative action, and multiplying by the zero from the semiring gives a zero in the abelian monoid:

```
class Module (R : Type u) (M : Type v) [Semiring R] [AddCommMonoid M] extends DistribMulAction R M where
  add_smul : ∀ (r s : R) (x : M), (r + s) • x = r • x + s • x
  zero_smul : ∀ x : M, (0 : R) • x = 0
```

Note the the class Module does not extend the class Semiring; it requires Semiring as an argument. The abelian monoid is also required separately in the definition. We call such a class "mixin". Thanks to this design, we do not need to define superclasses of Module in order to require "more than a module". Instead, we use superclasses in the respective arguments, i.e., "more than a semiring" and/or "more than an abelian monoid". For example, if we replace

```
[Semiring R] [AddCommMonoid M] [Module R M]
```

in a theorem statement by

```
[Field R] [AddCommGroup M] [Module R M]
```

we require `M` to be a vector space over `R`. We do not need to extend Module in order to define what a vector space is.

In our case, to state the theorem fintypeFarkasBartl, we need R to be a linearly ordered division ring, we need W to be an R-module, and we need V to be a linearly ordered R-module. The following list of requirements is almost correct:

```
[LinearOrderedDivisionRing R] [AddCommGroup W] [Module R W] [LinearOrderedAddCommGroup V] [Module R V]
```

The only missing assumption is the relationship between how R is ordered and how V is ordered. For that, we use another mixin, defined in Mathlib as follows:

```
class PosSMulMono (α β : Type*) [SMul α β] [Preorder α] [Preorder β] : Prop where
  elim {a : α} (ha : 0 ≤ a) {b₁ b₂ : β} (hb : b₁ ≤ b₂) : a • b₁ ≤ a • b₂
```

We encourage the reader to try and delete various assumptions and see which parts of the proofs underline by red.

2.5 Linear programming

Extended linear programs are defined as follows:

```
structure ExtendedLP (I J F : Type*) [LinearOrderedField F] where
  /-- The left-hand-side matrix. -/
  A : Matrix I J F∞
  /-- The right-hand-side vector. -/
  b : I → F∞
  /-- The objective function coefficients. -/
  c : J → F∞
  /-- No '⊥' and '⊤' in the same row. -/
  hAi : ¬∃ i : I, (∃ j : J, A i j = ⊥) ∧ (∃ j : J, A i j = ⊤)
  /-- No '⊥' and '⊤' in the same column. -/
  hAj : ¬∃ j : J, (∃ i : I, A i j = ⊥) ∧ (∃ i : I, A i j = ⊤)
  /-- No '⊥' in the right-hand-side vector. -/
  hbi : ¬∃ i : I, b i = ⊥
  /-- No '⊥' in the objective function coefficients. -/
  hcj : ¬∃ j : J, c j = ⊥
  /-- No '⊤' in the row where the right-hand-side vector has '⊤'. -/
  hAb : ¬∃ i : I, (∃ j : J, A i j = ⊤) ∧ b i = ⊤
  /-- No '⊥' in the column where the objective function has '⊤'. -/
  hAc : ¬∃ j : J, (∃ i : I, A i j = ⊥) ∧ c j = ⊤
```

Solution is defined as follows:

```
def ExtendedLP.IsSolution (P : ExtendedLP I J F) (x : J → F≥0) : Prop :=
  P.A ₘ* x ≤ P.b
```

Reaching a value is defined as follows:

```
def ExtendedLP.Reaches (P : ExtendedLP I J F) (r : F∞) : Prop :=
  ∃ x : J → F≥0, P.IsSolution x ∧ P.c ᵥ· x = r
```

Feasibility is defined as follows:

```
def ExtendedLP.IsFeasible (P : ExtendedLP I J F) : Prop :=
  ∃ p : F, P.Reaches (toE p)
```

Being bounded by a value (from below – we always minimize) is defined as follows:

```
def ExtendedLP.IsBoundedBy (P : ExtendedLP I J F) (r : F) : Prop :=
  ∀ p : F∞, P.Reaches p → r ≤ p
```

Being unbounded is defined as follows:

```
def ExtendedLP.IsUnbounded (P : ExtendedLP I J F) : Prop :=
  ¬∃ r : F, P.IsBoundedBy r
```

The following definition says how linear programs are dualized:

```
def ExtendedLP.dualize (P : ExtendedLP I J F) : ExtendedLP J I F :=
  ⟨-P.Aᵀ, P.c, P.b, by aeply P.hAj, by aeply P.hAi, P.hcj, P.hbi, by aeply P.hAc, by aeply P.hAb⟩
```

The definition of optimum is, sadly, very complicated (TODO explain this shit):

```
noncomputable def ExtendedLP.optimum (P : ExtendedLP I J F) : Option F∞ :=
  if P.IsFeasible then
    if P.IsUnbounded then
      some ⊥ -- unbounded means that the minimum is `⊥`
    else
      if hf : ∃ r : F, P.Reaches (toE r) ∧ P.IsBoundedBy r then
        some (toE hf.choose) -- the minimum is finite
      else
        none -- invalid finite value (infimum is not attained)
  else
    some ⊤ -- infeasible means that the minimum is `⊤`
```

Finally, we define what opposite values are:

```
def OppositesOpt : Option F∞ → Option F∞ → Prop
| (p : F∞), (q : F∞) => p = -q  -- opposite values; includes `⊥ = -⊤` and `⊤ = -⊥`
| _       , _       => False   -- namely `OppositesOpt none none = False`
```

3 Formal statements of selected results

3.1 Main corollaries

```
theorem equalityFarkas (A : Matrix I J F) (b : I → F) :
    (∃ x : J → F, 0 ≤ x ∧ A *ᵥ x = b) ≠ (∃ y : I → F, 0 ≤ Aᵀ *ᵥ y ∧ b ⬝ᵥ y < 0)

theorem inequalityFarkas [DecidableEq I] (A : Matrix I J F) (b : I → F) :
    (∃ x : J → F, 0 ≤ x ∧ A *ᵥ x ≤ b) ≠ (∃ y : I → F, 0 ≤ y ∧ 0 ≤ Aᵀ *ᵥ y ∧ b ⬝ᵥ y < 0)

theorem StandardLP.strongDuality {P : StandardLP I J R} (hP : P.IsFeasible ∨ P.dualize.IsFeasible) :
    OppositesOpt P.optimum P.dualize.optimum
```

3.2 Main results

```
theorem fintypeFarkasBartl {J : Type*} [Fintype J] [LinearOrderedDivisionRing R]
    [LinearOrderedAddCommGroup V] [Module R V] [PosSMulMono R V] [AddCommGroup W] [Module R W]
    (A : W →ₗ[R] J → R) (b : W →ₗ[R] V) :
    (∃ x : J → V, 0 ≤ x ∧ ∀ w : W, ∑ j : J, A w j • x j = b w) ≠ (∃ y : W, 0 ≤ A y ∧ b y < 0)

theorem extendedFarkas (A : Matrix I J F∞) (b : I → F∞)
    (hA : ¬∃ i : I, (∃ j : J, A i j = ⊥) ∧ (∃ j : J, A i j = ⊤))
    (hAT : ¬∃ j : J, (∃ i : I, A i j = ⊥) ∧ (∃ i : I, A i j = ⊤))
    (hAb : ¬∃ i : I, (∃ j : J, A i j = ⊤) ∧ b i = ⊤)
    (hAb' : ¬∃ i : I, (∃ j : J, A i j = ⊥) ∧ b i = ⊥) :
    (∃ x : J → F≥0, A ₘ* x ≤ b) ≠ (∃ y : I → F≥0, -Aᵀ ₘ* y ≤ 0 ∧ b ᵥ⬝ y < 0)

theorem ExtendedLP.strongDuality {P : ExtendedLP I J F} (hP : P.IsFeasible ∨ P.dualize.IsFeasible) :
  OppositesOpt P.optimum P.dualize.optimum
```

4 Proving the Farkas-Bartl theorem

We prove finFarkasBartl and, in the end, we obtain fintypeFarkasBartl as corollary.

Theorem (finFarkasBartl): Let n be a natural number. Let R be a linearly ordered division ring. Let W be an R-module. Let V be a linearly ordered R-module. Let A be an R-linear map from W to $([n] \to R)$. Let b be an R-linear map from W to V. Exactly one of the following exists:

- nonnegative vector family $x : [n] \to V$ such that, for all $w : W$, we have $\sum_{j:[n]} (A\ w)_j \cdot x_j = b\ w$
- vector $y : W$ such that $A\ y \geq 0$ and $b\ y < 0$

The only difference is that finFarkasBartl uses $[n] = \{0, \ldots, n-1\}$ instead of an arbitrary (unordered) finite type J.

Proof idea: We first prove that both cannot exist at the same time. Assume we have x and y of said properties. We plug y for w and obtain $\sum_{j:[n]} (A\ y)_j \cdot x_j = b\ y$. On the left-hand side, we have a sum of nonnegative vectors, which contradicts $b\ y < 0$.

We prove "at least one exists" by induction on n. If $n = 0$ then $A\ y \geq 0$ is a tautology. We consider b. Either b maps everything to the zero vector, which allows x to be the empty vector family, or some w gets mapped to a nonzero vector, where we choose y to be either w or $(-w)$. Since V is linearly ordered, one of them satisfies $b\ y < 0$. Now we precisely state how the induction step will be.

Lemma (industepFarkasBartl): Let m be a natural number. Let R be a linearly ordered division ring. Let W be an R-module. Let V be a linearly ordered R-module. Assume (induction hypothesis) that for all R-linear maps $A_0 : W \to ([m] \to R)$ and $b_0 : W \to V$, the formula "$\forall y_0 : W\ ,\ A_0\ y_0 \geq 0 \implies b_0\ y_0 \geq 0$" implies existence of a nonnegative vector family $x_0 : [m] \to V$ such that, for all $w_0 : W$, $\sum_{i:[m]} (A_0\ w_0)_i \cdot (x_0)_i = b_0\ w_0$. Let A be an R-linear map from W to $([m+1] \to R)$. Let b be an R-linear map from W to V. Assume that, for all $y : W$, $A\ y \geq 0$ implies $b\ y \geq 0$. We claim there exists a nonnegative vector family $x : [m+1] \to V$ such that, for all $w : W$, we have $\sum_{i:[m+1]} (A\ w)_i \cdot x_i = b\ w$. TODO names like A_0 don't work well with the "subscript notation" on paper.

Proof idea: Let $A_{<m}$ roughly mean $A \restriction [m]$. To be more precise, $A_{<m}$ is a function that maps $(w : W)$ to $(A\ w) \restriction [m]$, i.e., $A_{<m}$ is an R-linear map from W to $([m] \to R)$ that behaves exactly like A where it is defined. We distinguish two cases. If, for all $y : W$, the inequality $A_{<m}\ y \geq 0$ implies $b\ y \geq 0$, then plug $A_{<m}$ for A_0, obtain x_0, and construct a vector family x such that $x_m = 0$ and otherwise x copies x_0. We easily check that x is nonnegative and that $\sum_{i:[m+1]} (A\ w)_i \cdot x_i = b\ w$ holds.

In the second case, we have y' such that $A_{<m}\ y' \geq 0$ holds but $b\ y' < 0$ also holds. We realize that $(A\ y')_m < 0$. We now declare $y := (A\ y')_m \cdot y'$ and observe $(A\ y)_m = 1$. We establish the following facts (proofs are omitted):

- for all $w : W$, we have $A\ (w - ((A\ w)_m \cdot y)) = 0$
- for all $w : W$, the inequality $A_{<m}\ (w - ((A\ w)_m \cdot y)) \geq 0$ implies $b\ (w - ((A\ w)_m \cdot y)) \geq 0$
- for all $w : W$, the inequality $A_{<m}\ w - A_{<m}\ ((A\ w)_m \cdot y) \geq 0$ implies $b\ w - b\ ((A\ w)_m \cdot y) \geq 0$
- for all $w : W$, the inequality $(A_{<m} - (z \mapsto (A\ z)_m \cdot (A_{<m}\ y)))\ w \geq 0$ implies $(b - (z \mapsto (A\ z)_m \cdot (b\ y)))\ w \geq 0$

We observe that $A_0 := A_{<m} - (z \mapsto (A\ z)_m \cdot (A_{<m}\ y))$ and $b_0 := b - (z \mapsto (A\ z)_m \cdot (b\ y))$ are R-linear maps. Thanks to the last fact, we can apply induction hypothesis to A_0 and b_0. We obtain a nonnegative vector family x' such that, for all $w_0 : W$, $\sum_{i:[m]} (A_0\ w_0)_i \cdot x'_i = b_0\ w_0$. It remains to construct a nonnegative vector family $x : [m+1] \to V$ such that, for all $w : W$, we have $\sum_{i:[m+1]} (A\ w)_i \cdot x_i = b\ w$. We choose $x_m = b\ y - \sum_{i:[m]} (A_{<m}\ y)_i \cdot x'_i$ and otherwise x copies x'. We check that our x has the required properties. Qed.

We complete the proof of finFarkasBartl by applying industepFarkasBartl to $A_{\leq n}$ and b. Finally, we obtain fintypeFarkasBartl from finFarkasBartl using some boring mechanisms regarding equivalence between finite types.

5 Extended Farkas theorem

We prove inequalityFarkas by applying equalityFarkas to the matrix $(1 \mid A)$ where 1 is the identity matrix of type $(I \times I) \to F$.

Theorem (inequalityFarkas_neg): Let I and J be finite types. Let F be a linearly ordered field. Let A be a matrix of type $(I \times J) \to F$. Let b be a vector of type $I \to F$. Exactly one of the following exists:

- nonnegative vector $x : J \to F$ such that $A \cdot x \leq b$
- nonnegative vector $y : I \to F$ such that $(-A^T) \cdot y \leq 0$ and $b \cdot y < 0$

Obviously, inequalityFarkas_neg is an immediate corollary of inequalityFarkas.

Theorem (extendedFarkas): Let I and J be finite types. Let F be a linearly ordered field. Let A be a matrix of type $(I \times J) \to F_\infty$. Let b be a vector of type $I \to F_\infty$. Assume that A does not have $\bot$ and $\top$ in the same row. Assume that A does not have $\bot$ and $\top$ in the same column. Assume that A does not have $\top$ in any row where b has $\top$. Assume that A does not have $\bot$ in any row where b has $\bot$. Exactly one of the following exists:

- nonnegative vector $x : J \to F$ such that $A \cdot x \leq b$
- nonnegative vector $y : I \to F$ such that $(-A^T) \cdot y \leq 0$ and $b \cdot y < 0$

(restated)

5.1 Proof idea

We need to do the following steps in given order:

1. Delete all rows of both A and b where A has $\bot$ or b has $\top$ (they are tautologies).
2. Delete all columns of A that contain $\top$ (they force respective variables to be zero).
3. If b contains $\bot$, then $A \cdot x \leq b$ cannot be satisfied, but $y = 0$ satisfies $(-A^T) \cdot y \leq 0$ and $b \cdot y < 0$. Stop here.
4. Assume there is no $\bot$ in b. Use inequalityFarkas_neg. In either case, extend x or y with zeros on all deleted positions.

5.2 Counterexamples

If A has $\bot$ and $\top$ in the same row, it may happen that both x and y exist:

$$A = \begin{pmatrix} \bot & \top \\ 0 & -1 \end{pmatrix} \qquad b = \begin{pmatrix} 0 \\ -1 \end{pmatrix} \qquad x = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \qquad y = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

If A has $\bot$ and $\top$ in the same column, it may happen that both x and y exist:

$$A = \begin{pmatrix} \bot \\ \top \end{pmatrix} \qquad b = \begin{pmatrix} -1 \\ 0 \end{pmatrix} \qquad x = \begin{pmatrix} 0 \end{pmatrix} \qquad y = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

If A has $\top$ in a row where b has $\top$, it may happen that both x and y exist:

$$A = \begin{pmatrix} \top \\ -1 \end{pmatrix} \qquad b = \begin{pmatrix} \top \\ -1 \end{pmatrix} \qquad x = \begin{pmatrix} 1 \end{pmatrix} \qquad y = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

If A has $\bot$ in a row where b has $\bot$, it may happen that both x and y exist:

$$A = \begin{pmatrix} \bot \end{pmatrix} \qquad b = \begin{pmatrix} \bot \end{pmatrix} \qquad x = \begin{pmatrix} 1 \end{pmatrix} \qquad y = \begin{pmatrix} 0 \end{pmatrix}$$

6 Extended strong LP duality

We start with the weak duality and then move to the strong duality. We will use extendedFarkas in several places.

Theorem (weakDuality): Let F be a linearly ordered field. Let P be a linear program over F_∞. If P reaches p and the dual of P reaches q, then $p + q \geq 0$. (restated)

Proof idea: There is a vector x such that $A \cdot x \leq b$ and $c \cdot x = p$. Apply extendedFarkas to the following matrix and vector:

$$\begin{pmatrix} A \\ c \end{pmatrix} \qquad \begin{pmatrix} b \\ c \cdot x \end{pmatrix}$$

Lemma (infeasible_of_unbounded): If a linear program P is unbounded, the dual of P cannot be feasible.

Proof idea: Assume that P is unbounded, but the dual of P is feasible. Obtain contradiction using weakDuality.

Lemma (unbounded_of_reaches_le): Let F be a linearly ordered field. Let P be a linear program over F_∞. Assume that for each s in F there exists p in F_∞ such that P reaches p and $p \leq s$. Then P is unbounded.

Proof idea: It suffices to prove that for each r in F there exists p' in F_∞ such that P reaches p' and $p' < r$. Apply the assumption to $r-1$.

Lemma (unbounded_of_feasible_of_neg): Let F be a linearly ordered field. Let P be a linear program over F_∞ that is feasible. Let x_0 be a nonnegative vector such that $c \cdot x_0 < 0$ and $A \cdot x_0 + 0 \cdot (-b) \leq 0$. Then P is unbounded.

Proof idea: There is a nonnegative vector x_p such that $A \cdot x_p \leq b$ and $c \cdot x_p = e$ for some e in F_∞. We apply unbounded_of_reaches_le. In case $e \leq s$, we use x_p and we are done. Otherwise, consider what $c \cdot x_0$ equals to. We cannot have $c \cdot x_0 = \bot$ because c does not contain $\bot$. We cannot have $c \cdot x_0 = \top$ because $c \cdot x_0 < 0$. Hence $c \cdot x_0 = d$ for some d in F. Observe that the fraction $\frac{s-e}{d}$ is well defined and it is positive. Use $x_p + \frac{s-e}{d} \cdot x_0$.

Lemma (strongDuality_aux): Let P be a linear program such that P is feasible and the dual of P is also feasible. There is a value p reached by P and a value q reached by the dual of P such that $p + q \leq 0$.

Proof idea: Apply extendedFarkas to the following matrix and vector:

$$\begin{pmatrix} A & 0 \\ 0 & -A^T \\ b & c \end{pmatrix} \qquad \begin{pmatrix} b \\ c \\ 0 \end{pmatrix}$$

TODO.

Lemma (strongDuality_of_both_feasible): Let P be a linear program such that P is feasible and the dual of P is also feasible. There is a finite value r such that P reaches $-r$ and the dual of P reaches r.

Proof idea: From strongDuality_aux we have a value p reached by P and a value q reached by the dual of P such that $p + q \leq 0$. We apply weakDuality to p and q to obtain $p + q \geq 0$. We set $r := q$.

Lemma (unbounded_of_feasible_of_infeasible): Let P be a linear program such that P is feasible but the dual of P is not feasible. Then P is unbounded.

Proof idea: TODO.

Lemma (optimum_unique): Let P be a linear program. Let r be a valued reached by P such that P is bounded by r. Let s be a valued reached by P such that P is bounded by s. Then $r = s$.

Proof idea: TODO.

Lemma (optimum_eq_of_reaches_bounded): Let P be a linear program. Let r be a valued reached by P such that P is bounded by r. Then the optimum of P is r.

Proof idea: Apply the axiom of choice to the definition of optimum and use optimum_unique.

Lemma (strongDuality_of_prim_feas): Let P be a linear program that is feasible. The strong duality holds.

Proof idea: TODO.

Theorem (optimum_neq_none): Every linear program has optimum.

Proof idea: If a linear program P is feasible, the existence of optimum follows from strongDuality_of_prim_feas. Otherwise, the optimum of P is $\top$ by definition.

Lemma (dualize_dualize): Let P be a linear program. The dual of the dual of P is exactly P.

Proof idea: $-(-A^T)^T = A$

Lemma (strongDuality_of_dual_feas): Let P be a linear program whose dual is feasible. The strong duality holds.

Proof idea: Apply strongDuality_of_prim_feas to the dual of P and use dualize_dualize.

Theorem (strongDuality): Let F be a linearly ordered field. Let P be a linear program over F_∞. If P or its dual is feasible (at least one of them), then there exists p in F_∞ such that P has optimum p and the dual of P has optimum $-p$. (restated)

Proof idea: Use strongDuality_of_prim_feas or strongDuality_of_dual_feas.

6.1 Counterexample (first two conditions)

If A has $\bot$ and $\top$ in the same column, the following may happen:

$$A = \begin{pmatrix} \bot \\ \top \end{pmatrix} \qquad b = \begin{pmatrix} -1 \\ 0 \end{pmatrix} \qquad c = \begin{pmatrix} 0 \end{pmatrix}$$

If A has $\bot$ and $\top$ in the same row, the following may happen:

$$A = \begin{pmatrix} \top & \bot \end{pmatrix} \qquad b = \begin{pmatrix} 0 \end{pmatrix} \qquad c = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

The two linear programs above are duals of each other. Yet the optimum of the former is 0 but the optimum of the latter is $\bot$.

6.2 Counterexample (middle two conditions)

If b contains $\bot$, the following may happen:

$$A = \begin{pmatrix} \bot \end{pmatrix} \qquad b = \begin{pmatrix} \bot \end{pmatrix} \qquad c = \begin{pmatrix} 0 \end{pmatrix}$$

If c contains $\bot$, the following may happen:

$$A = \begin{pmatrix} \top \end{pmatrix} \qquad b = \begin{pmatrix} 0 \end{pmatrix} \qquad c = \begin{pmatrix} \bot \end{pmatrix}$$

The two linear programs above are duals of each other. Yet the optimum of the former is 0 but the optimum of the latter is $\bot$.

6.3 Counterexample (last two conditions)

If A has $\top$ in a row where b has $\top$, the following may happen:

$$A = \begin{pmatrix} \top \\ -1 \end{pmatrix} \qquad b = \begin{pmatrix} \top \\ -1 \end{pmatrix} \qquad c = \begin{pmatrix} 0 \end{pmatrix}$$

If A has $\bot$ in a column where c has $\top$, the following may happen:

$$A = \begin{pmatrix} \bot & 1 \end{pmatrix} \qquad b = \begin{pmatrix} 0 \end{pmatrix} \qquad c = \begin{pmatrix} \top \\ -1 \end{pmatrix}$$

The two linear programs above are duals of each other. Yet the optimum of the former is 0 but the optimum of the latter is $\bot$.

7 Related work

TODO presents an overcomplicated proof in Isabelle by analyzing the Simplex algorithm that already had been formally verified. It took them 30 pages to get to the basic Farkas; no generalization was provided.

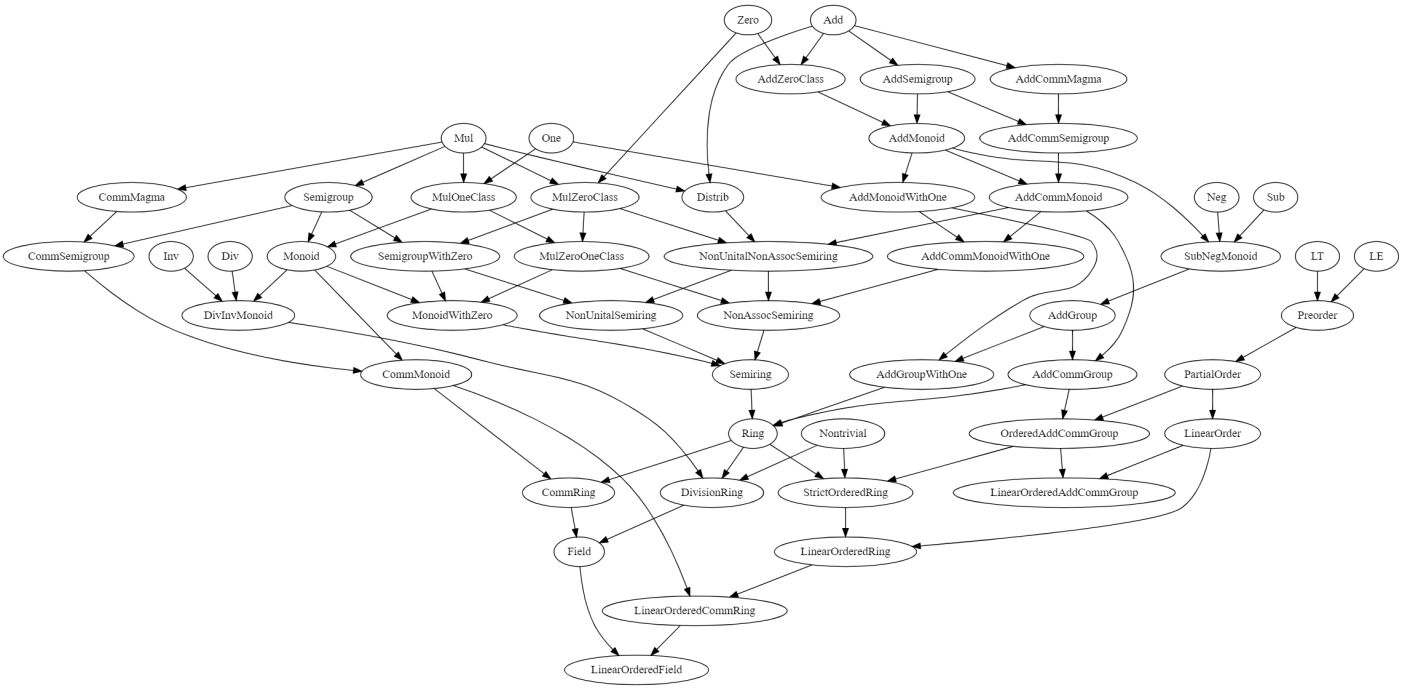
In Lean, it would be possible to prove Farkas for reals using the Hahn-Banach separation theorem. However, we do not yet know that the set of feasible solutions is closed.

TODO.

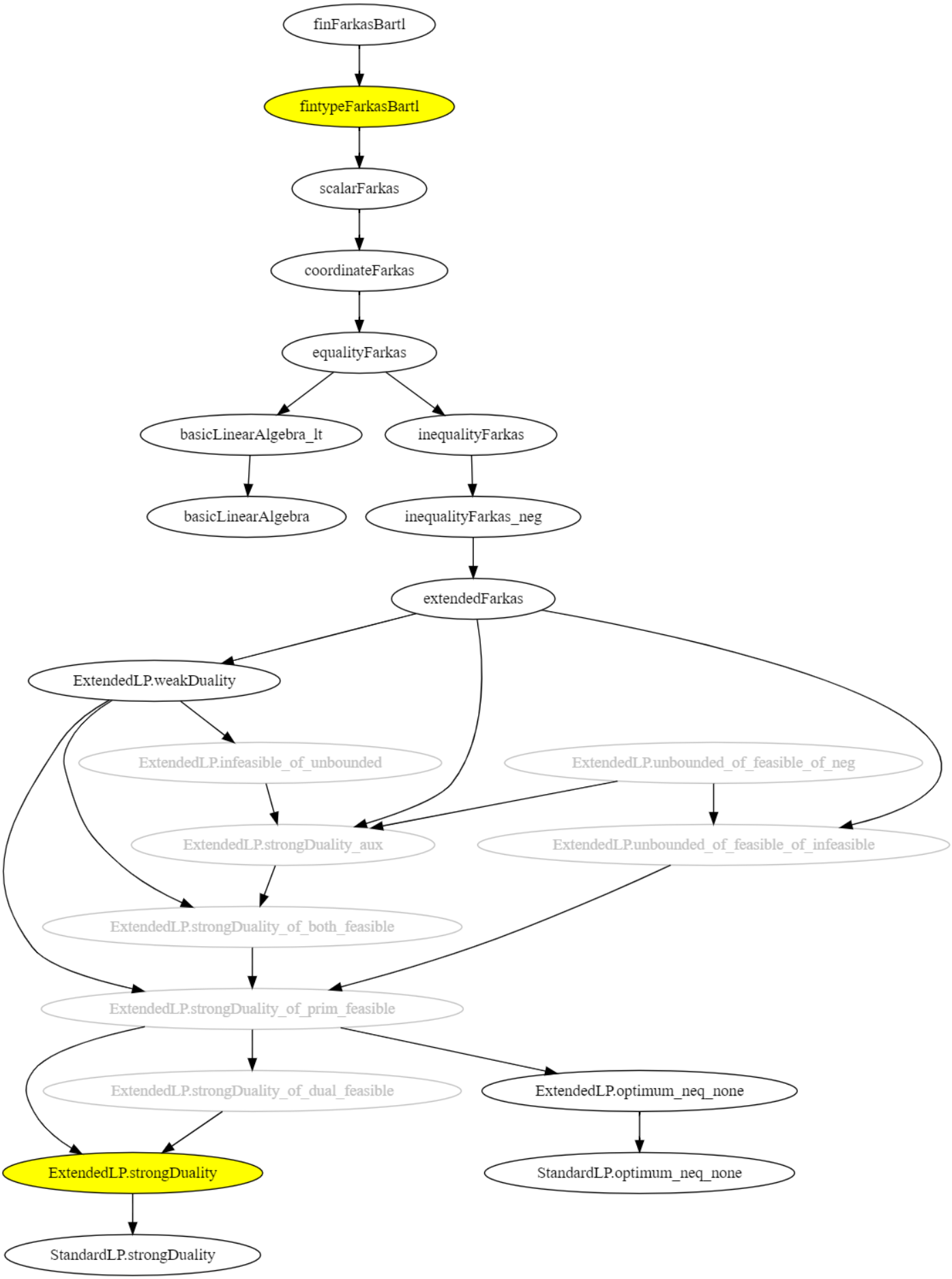
8 Conclusion

We formally verified several Farkas-like theorems in Lean 4. We extended the existing theory to a new setting where some coefficient can carry infinite values. We realized that the abstract work with modules over linearly ordered division rings and linear maps between them was fairly easy to carry on in Lean 4 thanks to the library Mathlib that is perfectly suited for such tasks. In contrast, manipulation with matrices got tiresome whenever we needed a not-fully-standard operation. It turns out Lean 4 cannot automate case analyses unless they take place in the "outer layers" of formulas. Summation over subtypes and summation of conditional expression made us developed a lot of ad-hoc machinery which we would have preferred to be handled by existing tactics. Another area where Lean 4 is not yet helpful is the search for counterexamples. Despite these difficulties, we find Lean 4 to be an extremely valuable tool for elegant expressions of mathematical formulas and for proving them formally.

9 Appendix: algebraic hierarchy up to LinearOrderedField in Mathlib



10 Appendix: dependencies between theorems



Theorems are in black. Selected lemmas are in gray. What we consider to be the main theorems are denoted by yellow background. TODO yellow extendedFarkas.