Duality theory in linear optimization and its extensions — formally verified

Martin Dvorak Vladimir Kolmogorov

Institute of Science and Technology Austria

{martin.dvorak,vnk}@ista.ac.at

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 over linearly ordered fields in Lean 4. Furthermore, we extend duality theory to the case when some coefficients are allowed to take "infinite values".

Keywords: Farkas lemma, linear programming, extended rationals, extended reals, calculus of inductive constructions

1 Introduction

When studying linear programming, we often ask whether a given system of linear inequalities has a solution. Duality theorems answer these naturally arising questions as follows: 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. When we formulate duality theorems as "either there exists a solution, or there exists a vector of coefficients that tells us how to derive $0 < 0$ from given inequalities", they are usually called *theorems of alternatives*. Two well-known theorems of alternatives are as follows (Farkas [10, 11]; Minkowski [18]).

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 $b \cdot y < 0$*

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 $b \cdot y < 0$*

For optimization problems, duality asserts a correspondence between the optimal value of a linear program and the optimal value of its dual (originally discussed in the context of zero-sum games by Dantzig and von Neumann [25]; later in Gale, Kuhn, Tucker [12]). The strong duality theorem, the cornerstone of linear programming, can be stated as follows.

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*

$$\min\{\, c \cdot x \mid x \geq 0 \ \wedge\ A \cdot x \leq b \,\} = -\min\{\, b \cdot y \mid y \geq 0 \ \wedge (-A^T) \cdot y \leq c \,\} \tag{1}$$

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

Using identity $\min\{c \cdot x \mid \ldots\} = -\max\{-c \cdot x \mid \ldots\}$ and replacing c with $-c$, eq. (1) can be equivalently transformed to

$$\max\{\, c \cdot x \mid x \geq 0 \ \wedge\ A \cdot x \leq b \,\} = \min\{\, b \cdot y \mid y \geq 0 \ \wedge\ A^T \cdot y \geq c \,\} \tag{2}$$

which is probably a more familiar formulation of strong LP duality. Later it will be clear why we chose the "min / min" rather than the more idiomatic "max / min" formulation. Also, we do not investigate "asymmetric versions" such as:

$$\max\{\, c \cdot x \mid x \geq 0 \ \wedge\ A \cdot x = b \,\} = \min\{\, b \cdot y \mid A^T \cdot y \geq c \,\} \tag{3}$$

For many more Farkas-like theorems (beyond the scope of this paper), see for example [24].
This paper makes the following contributions.

- We formally prove several existing duality results (including the three theorems above) in Lean 4. In fact, we prove a more general version of `equalityFarkas` due to Bartl [3]. Section 1.1 says more about it.
- We establish (and formally prove in Lean 4) a new generalization of `inequalityFarkas` and `StandardLP.strongDuality` to the case when some of the coefficients are allowed to have infinite values. This scenario can be motivated by discrete optimization problems with "hard" constraints (the "hard" constraints declare what has to be be satisfied, whereäs "soft" constraints declare what should be optimized). A common way to concisely write down such problems mathematically is to use infinite coefficients in front of the corresponding terms in the objective. Since infinities are not allowed in traditional LPs, "soft" and "hard" constraints need to be handled differently when formulating LP relaxations of such problems (see e.g. [14]). Our work provides a more direct way to formulate such relaxations. Section 1.2 says more about the extensions.

1.1 Bartl's generalization

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

Theorem (`coordinateFarkasBartl`)**.** *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 $(I \to R)$ to $(J \to R)$. Let b be an R-linear map 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 $b\ y < 0$*

Note that `equalityFarkas` for matrix $A : I \times J \to F$ and vector $b : I \to F$ can be obtained by applying `coordinateFarkasBartl` to the F-linear maps $(A^T \cdot\ )$ and $(b \cdot\ )$ utilizing the fact that two linear maps are equal if and only if they map the basis vectors equally.

In the next generalization (similar to [9]), the partially ordered module $I \to R$ is replaced by a general R-module W.

Theorem (`almostFarkasBartl`)**.** *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, certain occurrences of R are replaced by a linearly ordered R-module V whose order respects the order on R (for a formal definition, see the end of Section 2.4).

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 whose ordering satisfies monotonicity of scalar multiplication by nonnegative elements on the left. 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 `almostFarkasBartl` (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 4, which is where the heavy lifting comes. Our proof is based on [4].

1.2 Extension to infinite coefficients

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, negation, and scalar action on F_∞ as follows:*

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

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

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

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 $b \cdot y < 0$*

Note `extendedFarkas` has four preconditions on matrix A and vector b. In Section 7.1 we show that omitting any of them makes the theorem false. Observe that `inequalityFarkas` has condition $A^T \cdot y \geq 0$ in the second branch, while in `extendedFarkas` we changed it to $(-A^T) \cdot y \leq 0$. The two conditions are equivalent for finite-valued matrices A, but not necessarily for matrices A with infinities (e.g. condition $(-\top) \cdot 0 \geq 0$ is false but $\top \cdot 0 \leq 0$ is true). One must be careful when formulating this condition; for example, using the condition from `inequalityFarkas` would make the theorem false even if A has a single $\bot$ entry (see Section 7.1).

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 $J \to F_\infty$. 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.*

A nonnegative vector $x : J \to R$ is a solution *to P if $A \cdot x \leq b$. We say that P is* feasible *if there exists* solution x *with $c \cdot x \neq \top$. We say that P is* unbounded *if, for any $r \in F$, there exists solution x with $c \cdot x \leq r$.*

The optimum *of P, denoted as $P^\star$, is defined as follows. If P is not feasible then $P^\star = \top$. Else, if P is unbounded, then $P^\star = \bot$. Else, if there exists finite $r \in F$ such that $c \cdot x \geq r$ for all solutions x and $c \cdot x^\star = r$ for some solution $x^\star$, then $P^\star = r$. Otherwise, $P^\star$ is undefined.*

In order to state our duality results, we need a few more definitions.

Definition. *Linear Program $P = (A, b, c)$ over F_∞ is said to be* valid *if it satisfies the following six conditions:*

- *A does not have $\bot$ and $\top$ in the same row*
- *A does not have $\bot$ and $\top$ in the same column*
- *A does not have $\bot$ in any row where b has $\bot$*
- *A does not have $\top$ in any column where c has $\bot$*
- *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 the linear program $(-A^T, c, b)$ is the dual *of P.*

It is straightforward to check that the dual of a valid LP is valid.

Theorem (`ValidELP.strongDuality`)**.** *Let F be a linearly ordered field, P be a valid linear program over F_∞ and D be its dual. Then $P^\star$ and $D^\star$ are defined. If at least one of them is feasible, i.e., $(P^\star, D^\star) \neq (\top, \top)$, then $P^\star = -D^\star$.*

Similar to `extendedFarkas`, all six conditions in the definition of a valid LP are necessary; omitting any one of them makes `ValidELP.strongDuality` false (see counterexamples in Section 7.2).

Note that the conclusion of the theorem ($P^\star = -D^\star$) can be reformulated, when we use highly informal notation, as in eq. (1):

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

If all entries of (A, b, c) are finite then this equation can be stated in many equivalent ways, e.g. as in (2). This reformulation uses the facts that $(-c) \cdot x = -(c \cdot x)$, and that $(-A^T) \cdot y \leq c$ is equivalent to $A^T \cdot y \geq -c$. These facts are no longer true if infinities are allowed, so one must be careful when formulating the duality theorem for F_∞. We considered several "max / min" and "max / max" versions, but they all required stronger preconditions compared to the "min / min" version given in `ValidELP.strongDuality`.

1.3 Structure of this paper

In Section 2, we explain all underlying definitions and comment on the formalization process; following the philosophy of [21] 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 formally state theorems from Section 1 using definitions from Section 2. In Section 4, we prove `fintypeFarkasBartl` (stated in Section 1.1), from which we obtain `equalityFarkas` as a corollary. In Section 5, we prove `extendedFarkas` (stated in Section 1.2). In Section 6, we prove `ValidELP.strongDuality` (stated in Section 1.2), from which we obtain `StandardLP.strongDuality` as a corollary. In Section 7, we show what happens when various preconditions are not satisfied.

Repository `https://github.com/madvorak/duality/tree/v2` contains the 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.[1] This paper attempts to be an accurate description of the `duality` project. However, in case of any discrepancy, the code shall prevail.

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

2 Preliminaries

There are many layers of definitions that are built before say what a linearly ordered field F is (used e.g. in `equalityFarkas`), what a linearly ordered division ring R is (used e.g. in `almostFarkasBartl`), and what a linearly ordered abelian group V is (used in `fintypeFarkasBartl`). Section 2.1 mainly documents existing Mathlib definitions; we also highlight how we added linearly ordered division rings into our project. Section 2.2 then explains how we defined extended linearly ordered fields. Section 2.3 reviews vectors and matrices; we especially focus on how multiplication between them is defined. Section 2.4 explains how modules are implemented. Section 2.5 provides the formal definition of extended linear programs.

2.1 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 (please note that "adding minus itself gives zero" is not required yet; that will be required e.g. in an additive group):

```
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
  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 (please note that "multiplication by an inverse gives one" is not required yet):

```
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
```

Nonunital-nonassociative-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
```

Nonunital-semiring is a nonunital-nonassociative-semiring that forms a semigroup with zero (i.e., the semigroup-with-zero requirement makes it associative):

```
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 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 : 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, and embeds all integers:

```
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)
```

Nonassociative-semiring is a nonunital-nonassociative-semiring that has a well-behaved multiplication by both zero and one and forms an abelian monoid with one (i.e., the unit is finally defined):

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

Semiring is a nonunital-semiring and nonassociative-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 nontrivial ring whose multiplication forms a division monoid, whose nonzero elements have multiplicative inverses, whose zero is inverse to itself (if you find the equality $0^{-1}=0$ disturbing, read [1] that explains it), and embeds rational numbers:

```
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 $\leq$ and $<$ where the strict comparison $a < b$ is equivalent to $a \leq b \wedge \neg(b \leq a)$ given by the relation $\leq$ which is neither required to be symmetric nor required to be 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
```

In our project, 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`. Because of that, we provide a custom instance that converts `LinearOrderedField` to `LinearOrderedDivisionRing` as follows:

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

This instance is needed for the step from `coordinateFarkasBartl` 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⟩
```

Line-by-line, we see that:

- negating $\bot$ gives $\top$
- negating $\top$ gives $\bot$
- negating any finite value x gives $-x$ converted to the type F_∞

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

`F∞`

for F_∞ and also the notation

`F≥0`

for the type of nonnegative elements of F. We define a scalar action of the nonnegative elements on F_∞ as follows:

```
def EF.smulNN (c : F≥0) : F∞ → F∞
| ⊥ => ⊥
| ⊤ => if c = 0 then 0 else ⊤
| (f : F) => toE (c.val * f)
```

Line-by-line, we see that:

- multiplying $\bot$ by anything from the left gives $\bot$
- multiplying $\top$ by 0 from the left gives 0; multiplying $\top$ by a strictly positive element from the left gives $\top$
- multiplying any finite value f by a coefficient c from the left gives $c \cdot f$ converted to the type F_∞

Scalar action on F_∞ by negative elements from the left is undefined. Multiplication between two elements of F_∞ is also undefined.

2.3 Vectors and matrices

We distinguish two types of vectors; implicit vectors and explicit vectors. Implicit vectors (called just "vectors") are members of a vector space; they do not have any internal structure (in the informal text, we use the word "vectors" somewhat loosely; it can refer to members of any module). Explicit vectors are functions from coordinates to values (again, we use the word "explicit vector" (or just "vector" when it is clear from context that our vector is a map) not only when they form a vector space). The type of coordinates does not have to 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 type. This is 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 do not 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 do not assume any property of multiplication in the definition of multiplication between matrices and vectors; they do not 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. Explicit vectors are neither row vectors nor column vectors; they are just maps. It is only in our imagination that we treat certain occurrences of vectors as rows or columns; in reality, it is the position of the vector in the term that determines what the vector does there.

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; the 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 1.2 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 do not 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 class `Module` does not extend the class `Semiring`; instead, it requires `Semiring` as an argument. The abelian monoid is also required as argument 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
```

Adding `[PosSMulMono R V]` to the list of requirements solves the issue.

We encourage the reader to try and delete various assumptions and see which parts of the proofs get underlined in red.

TODO explain ordering (specifically, nonnegativity) on explicit vectors.

Throughout the paper, we do not distinguish between nonnegative explicit vectors and explicit vectors of nonnegative elements. The code, however, does distinguish between them.

2.5 Linear programming

Extended linear programs are defined as follows: TODO Matrix was not defined!

```
structure ExtendedLP (I J F : Type*) [LinearOrderedField F] where
  A : Matrix I J F∞
  b : I → F∞
  c : J → F∞
```

Vector x made of finite nonnegative values is a solution if and only if $A \cdot x \leq b$ holds:

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

P reaches a value r if and only if P has solution x such that $c \cdot x = r$ holds:

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

P is feasible if and only if P reaches a value different from $\top$:

```
def ExtendedLP.IsFeasible [Fintype J] (P : ExtendedLP I J F) : Prop :=
  ∃ p : F∞, P.Reaches p ∧ p ≠ ⊤
```

P is bounded by a value r (from below – we always minimize) if and only if P reaches only values greater or equal to r:

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

P is unbounded if and only if P has no finite lower bound:

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

Valid extended linear programs are defined as follows (the six conditions were introduced in Section 1.2):

```
structure ValidELP (I J F : Type*) [LinearOrderedField F] extends ExtendedLP I J F where
  hAi : ¬∃ i : I, (∃ j : J, A i j = ⊥) ∧ (∃ j : J, A i j = ⊤)
  hAj : ¬∃ j : J, (∃ i : I, A i j = ⊥) ∧ (∃ i : I, A i j = ⊤)
  hbA : ¬∃ i : I, (∃ j : J, A i j = ⊥) ∧ b i = ⊥
  hcA : ¬∃ j : J, (∃ i : I, A i j = ⊤) ∧ c j = ⊥
  hAb : ¬∃ i : I, (∃ j : J, A i j = ⊤) ∧ b i = ⊤
  hAc : ¬∃ j : J, (∃ i : I, A i j = ⊥) ∧ c j = ⊤
```

The following definition says how linear programs are dualized (first, the abbreviation `ExtendedLP.dualize` says that (A, b, c) is mapped to $(-A^T, c, b)$; then the definition `ValidELP.dualize` says that the dualization of `ValidELP` is inherited from `ExtendedLP.dualize`; the remaining six lines generate proofs that our six conditions stay satisfied after dualization, where `aeply` is a custom tactic based on `aesop` [15] and `apply`):

```
abbrev ExtendedLP.dualize (P : ExtendedLP I J F) : ExtendedLP J I F :=
  ⟨-P.Aᵀ, P.c, P.b⟩

def ValidELP.dualize (P : ValidELP I J F) : ValidELP J I F where
  toExtendedLP := P.toExtendedLP.dualize
  hAi := by aeply P.hAj
  hAj := by aeply P.hAi
  hbA := by aeply P.hcA
  hcA := by aeply P.hbA
  hAb := by aeply P.hAc
  hAc := by aeply P.hAb
```

The definition of optimum is, sadly, very complicated (will be explained below):

```
noncomputable def ExtendedLP.optimum [Fintype J] (P : ExtendedLP I J F) : Option F∞ :=
  if ¬P.IsFeasible then
    some ⊤
  else
    if P.IsUnbounded then
      some ⊥
    else
      if hr : ∃ r : F, P.Reaches (toE r) ∧ P.IsBoundedBy r then
        some (toE hr.choose)
      else
        none
```

The type `Option F∞`, which is implemented as `Option (Option (Option F))` after unfolding definitions, allows the following values:

- `none`
- `some ⊥` implemented as `some none`
- `some ⊤` implemented as `some (some none)`
- `some (toE r)` implemented as `some (some (some r))` for any `r : F`

We assign the following semantics to `Option F∞` values:

- `none` ... invalid finite value (infimum is not attained)
- `some ⊥` ... feasible unbounded
- `some ⊤` ... infeasible
- `some (toE r)` ... the minimum is a finite value r

The definition `ExtendedLP.optimum` first asks if P is feasible; if not, it returns `some ⊤` (i.e., the worst value). When P is feasible, it asks whether P is unbounded; if yes, it returns `some ⊥` (i.e., the best value). When P is feasible and bounded, it asks if there is a finite value r such that P reaches r and, at the same time, P is bounded by r; is so, it returns `some (toE r)`. Otherwise, it returns `none`. Note that we use the verbs "ask" and "return" metaphorically; `ExtendedLP.optimum` is not a computable function; it is just a mathematical definition (see the keyword `noncomputable def` above) you can prove things about.

Finally, we define what opposite values of the type `Option F∞` are (note that for any values except `none` we directly follow the definition of negation a.k.a. unary minus on the extended linearly ordered fields):

```
def OppositesOpt : Option F∞ → Option F∞ → Prop
| (p : F∞), (q : F∞) => p = -q
| _       , _       => False
```

For example, `OppositesOpt (-3) 3` and `OppositesOpt 5 (-5)` hold. The last line says that `OppositesOpt none none` is false. We later show that `none` is never the optimum of a valid[2] linear program, but we do not know it yet.

[2]In principle, we could say that `none` is never the optimum of any linear program, i.e., P^* is defined for every (extended) linear program P, but we did not bother proving it.

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∞)
    (hAi : ¬∃ i : I, (∃ j : J, A i j = ⊥) ∧ (∃ j : J, A i j = ⊤))
    (hAj : ¬∃ j : J, (∃ i : I, A i j = ⊥) ∧ (∃ i : I, A i j = ⊤))
    (hAb : ¬∃ i : I, (∃ j : J, A i j = ⊤) ∧ b i = ⊤)
    (hbA : ¬∃ 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 whose ordering satisfies monotonicity of scalar multiplication by nonnegative elements on the left. 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 from `fintypeFarkasBartl` 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 whose ordering satisfies monotonicity of scalar multiplication by nonnegative elements on the left. Assume (induction hypothesis) that for all R-linear maps $\bar{A} : W \to ([m] \to R)$ and $\bar{b} : W \to V$, the formula "$\forall \bar{y} : W,\ \bar{A}\ \bar{y} \geq 0 \implies \bar{b}\ \bar{y} \geq 0$" implies existence of a nonnegative vector family $\bar{x} : [m] \to V$ such that, for all $\bar{w} : W$, $\sum_{i:[m]} (\bar{A}\ \bar{w})_i \cdot \bar{x}_i = \bar{b}\ \bar{w}$. 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 $\sum_{i:[m+1]} (A\ w)_i \cdot x_i = b\ w$.*

Proof idea: Let $A_{<m}$ denote a function that maps $(w : W)$ to $(A\ w)|_{[m]}$, 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 $\bar{A}$, obtain $\bar{x}$, and construct a vector family x such that $x_m = 0$ and otherwise x copies $\bar{x}$. 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 $y := ((A\ y')_m)^{-1} \cdot y'$ 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 $(A_{<m} - (z \mapsto (A\ z)_m \cdot (A_{<m}\ y)))\ w \geq 0$ implies $(b - (z \mapsto (A\ z)_m \cdot (b\ y)))\ w \geq 0$

We observe that $\bar{A} := A_{<m} - (z \mapsto (A\ z)_m \cdot (A_{<m}\ y))$ and $\bar{b} := b - (z \mapsto (A\ z)_m \cdot (b\ y))$ are R-linear maps. Thanks to the last fact, we can apply induction hypothesis to $\bar{A}$ and $\bar{b}$. We obtain a nonnegative vector family x' such that, for all $\bar{w} : W$, $\sum_{i:[m]}(\bar{A}\ \bar{w})_i \cdot x'_i = \bar{b}\ \bar{w}$. 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 Proving the 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`, restated)**.** *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 $b \cdot y < 0$*

5.1 Proof idea

We need to do the following steps in the 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.

6 Proving the Extended strong LP duality

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

Lemma (`weakDuality_of_no_bot`)**.** *Let F be a linearly ordered field. Let $P = (A, b, c)$ be a valid linear program over F_∞ such that neither b nor c contains $\bot$. If P reaches p and the dual of P reaches q, then $p + q \geq 0$.*

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 (`no_bot_of_reaches`)**.** *Let F be a linearly ordered field. Let $P = (A, b, c)$ be a valid linear program over F_∞. If P reaches any value p, then b does not contain $\bot$.*

Proof idea: It would contradict the assumption that A does not have $\bot$ in any row where b has $\bot$.

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

Proof idea: Apply `no_bot_of_reaches` to P. Apply `no_bot_of_reaches` to the dual of P. Finish the proof using `weakDuality_of_no_bot`.

Lemma (`no_bot_of_feasible`)**.** *Let F be a linearly ordered field. Let P be a valid linear program over F_∞. If P is feasible, then b does not contain $\bot$.*

Proof idea: This is just a weaker version of `no_bot_of_reaches`.

Lemma (`unbounded_of_reaches_le`)**.** *Let F be a linearly ordered field. Let P be a valid linear program over F_∞. Assume that for each r in F there exists p in F_∞ such that P reaches p and $p \leq r$. 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 valid 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 $e \neq \top$. We apply `unbounded_of_reaches_le`. In case $e \leq r$, we use x_p and we are done. Otherwise, consider what $c \cdot x_0$ equals to. If $c \cdot x_0 = \bot$ then we are done. 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 $\frac{r-e}{d}$ is well defined and it is positive. Use $x_p + \frac{r-e}{d} \cdot x_0$.

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

Proof idea: Apply `extendedFarkas` to the matrix $(A\big|_{\{i:I \mid b_i \neq \top\}})^T$ and the vector c.

Lemma (`infeasible_of_unbounded`)**.** *If a valid 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 (`dualize_dualize`)**.** *Let P be a valid linear program. The dual of the dual of P is exactly P.*

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

Lemma (`strongDuality_aux`)**.** *Let P be a valid 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:

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

In the first case, we obtain a nonnegative vectors x and y such that $A \cdot x \leq b$, $-A^T \cdot y \leq c$, and $c \cdot x + b \cdot y \leq 0$. We immediately see that P reaches $c \cdot x$, that the dual of P reaches $b \cdot y$, and that the desired inequality holds.

In the second case, we obtain nonnegative vectors y and x and a nonnegative scalar z such that $-A^T \cdot y + z \cdot (-c) \leq 0$, $A \cdot x + z \cdot (-b) \leq 0$, and $b \cdot y + c \cdot x < 0$. If $z > 0$, we finish the proof using $p := z^{-1} \cdot c \cdot x$ and $q := z^{-1} \cdot b \cdot y$ (and we do not care that this case cannot happen in reality). It remains to prove that $z = 0$ cannot happen. At least one of $b \cdot y$ or $c \cdot x$ must be strictly negative. If $c \cdot x < 0$, we apply `unbounded_of_feasible_of_neg` to P, which (using `infeasible_of_unbounded`) contradicts that the dual of P is feasible. If $b \cdot y < 0$, we apply `unbounded_of_feasible_of_neg` to the dual of P, which (using `infeasible_of_unbounded` and `dualize_dualize`) contradicts that P is feasible.

Lemma `no_bot_of_feasible` is used 12 times throughout this proof.

Lemma (`strongDuality_of_both_feasible`)**.** *Let P be a valid 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 (`optimum_unique`)**.** *Let P be a valid linear program. Let r be a value reached by P such that P is bounded by r. Let s be a value reached by P such that P is bounded by s. Then $r = s$.*

Proof idea: We prove $r \leq s$ by applying "P is bounded by r" to "s is reached by P". We prove $s \leq r$ by applying "P is bounded by s" to "r is reached by P".

Lemma (`optimum_eq_of_reaches_bounded`)**.** *Let P be a valid linear program. Let r be a value 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 valid linear program that is feasible. Then the optimum of P and the optimum of dual of P are opposites.*

Proof idea: If the dual of P is feasible as well, use `strongDuality_of_both_feasible` to obtain r such that P reaches $-r$ and the dual of P reaches r. Using `optimum_eq_of_reaches_bounded` together with `weakDuality`, conclude that the optimum of P is $-r$. Using `optimum_eq_of_reaches_bounded` together with `weakDuality`, conclude that the optimum of the dual of P is r. Observe that $-r$ and r are opposites.

If the dual of P is infeasible, the optimum of the dual of P is $\top$ by definition. Using `unbounded_of_feasible_of_infeasible` we get that the optimum of P is $\bot$. Observe that $\top$ and $\bot$ are opposites.

Theorem (`optimum_neq_none`)**.** *Every valid linear program has optimum.*

Proof idea: If a valid 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 (`strongDuality_of_dual_feas`)**.** *Let P be a valid linear program whose dual is feasible. Then the optimum of P and the optimum of dual of P are opposites.*

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

Theorem (`strongDuality`, restated)**.** *Let F be a linearly ordered field. Let P be a valid linear program over F_∞. If P or its dual is feasible (at least one of them), then the optimum of P and the optimum of dual of P are opposites.*

Proof idea: Use `strongDuality_of_prim_feas` or `strongDuality_of_dual_feas`.

7 Counterexamples

7.1 Counterexamples for `extendedFarkas`

Recall that `extendedFarkas` has four preconditions on matrix A and vector b. The following examples show that omitting any of these preconditions makes the theorem false — there may exist nonnegative vectors x, y such that $A \cdot x \le b$, $(-A^T) \cdot y \le 0$, and $b \cdot y < 0$.

$A = \begin{pmatrix} \bot & \top \\ 0 & -1 \end{pmatrix}$	$b = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$	$x = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$	$y = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$	A has $\bot$ and $\top$ in the same row
$A = \begin{pmatrix} \bot \\ \top \end{pmatrix}$	$b = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$	$x = \begin{pmatrix} 0 \end{pmatrix}$	$y = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$	A has $\bot$ and $\top$ in the same column
$A = \begin{pmatrix} \top \\ -1 \end{pmatrix}$	$b = \begin{pmatrix} \top \\ -1 \end{pmatrix}$	$x = \begin{pmatrix} 1 \end{pmatrix}$	$y = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$	A has $\top$ in a row where b has $\top$
$A = \begin{pmatrix} \bot \end{pmatrix}$	$b = \begin{pmatrix} \bot \end{pmatrix}$	$x = \begin{pmatrix} 1 \end{pmatrix}$	$y = \begin{pmatrix} 0 \end{pmatrix}$	A has $\bot$ in a row where b has $\bot$

We also claimed in Section 1.2 that changing condition $(-A^T) \cdot y \le 0$ to $A^T \cdot y \ge 0$ in `extendedFarkas` would cause the theorem to fail even when A has a single $\bot$ entry. The counterexample is as follows:

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

System $A \cdot x \le b$ does not have a solution, and neither does system $A^T \cdot y \ge 0$, $b \cdot y < 0$ (over nonnegative vectors x, y).

7.2 Counterexamples for `ValidELP.strongDuality`

Recall that `ValidELP.strongDuality` is formulated for *valid* LPs, and the definition of a valid LP has six conditions. In this section we show that omitting any of these six conditions makes the theorem false.

Let us consider the following six LPs over F_∞; all of them are written in the format $P = (A, b, c)$.

$$P_1 = \left(\begin{pmatrix} \bot \\ \top \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \end{pmatrix} \right) \qquad D_1 = \left(\begin{pmatrix} \top & \bot \end{pmatrix}, \begin{pmatrix} 0 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \end{pmatrix} \right)$$

$$P_2 = \left(\begin{pmatrix} \bot \end{pmatrix}, \begin{pmatrix} \bot \end{pmatrix}, \begin{pmatrix} 0 \end{pmatrix} \right) \qquad D_2 = \left(\begin{pmatrix} \top \end{pmatrix}, \begin{pmatrix} 0 \end{pmatrix}, \begin{pmatrix} \bot \end{pmatrix} \right)$$

$$P_3 = \left(\begin{pmatrix} \top \\ -1 \end{pmatrix}, \begin{pmatrix} \top \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \end{pmatrix} \right) \qquad D_3 = \left(\begin{pmatrix} \bot & 1 \end{pmatrix}, \begin{pmatrix} 0 \end{pmatrix}, \begin{pmatrix} \top \\ -1 \end{pmatrix} \right)$$

It can be checked that D_i is the dual of P_i for $i = 1, 2, 3$. Furthermore, strong duality fails in all cases, since the optimum of P_i is 0 and the optimum of D_i is $\bot$ for $i = 1, 2, 3$. Each of $P_1, D_1, P_2, D_2, P_3, D_3$ violates exactly one of the conditions in the definition of a valid LP.

8 Related work

8.1 Farkas-like theorems

There is a substantial body of work on linear inequalities and linear programming formalized in Isabelle. While our work is focused only on proving mathematical theorems, the work in Isabelle is motivated by the development of SMT solvers.

- Bottesch, Haslbeck, Thiemann [7] proved a variant of `equalityFarkas` for δ-rationals by analyzing a specific implementation of the Simplex algorithm [16] by Marić, Spasić, Thiemann.
- Bottesch, Raynaud, Thiemann [8] proved the Fundamental theorem of linear inequalities as well as both `equalityFarkas` and `inequalityFarkas` for all linearly ordered fields, alongside with the Carathéodory's theorem and the Farkas-Minkowski-Weyl theorem. They also investigated systems of linear mixed-integer inequalities.
- Thiemann himself [23] then proved the strong LP duality in the asymmetric version (3).

Sakaguchi [22] proved a version of `equalityFarkas` for linearly ordered fields in Rocq, using the Fourier-Motzkin elimination. Allamigeon and Katz [2] made a large contribution to the study of convex polyhedra in Rocq — among other results, they proved a version of `equalityFarkas` for linearly ordered fields as well as the strong LP duality in the asymmetric version (3).

8.2 Hahn-Banach theorems

Several Hahn-Banach theorems have been formalized in Lean as a part of Mathlib [17]. In particular, it would have been be possible to prove `equalityFarkas` for reals using the Hahn-Banach separation theorem for a convex closed set and a point [6]. It would then have been possible to extend the result to other duality theorems for reals. To our knowledge, the theorem [6] cannot be used to prove `equalityFarkas` for an arbitrary linearly ordered field. Therefore, we decided to prove `equalityFarkas` without appealing to the geometry and topology.

Note that certain Hahn-Banach theorems have also been formalized in Mizar [20], Alf [19], Isabelle [5], and Rocq [13].

9 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 excellent tool for elegant expressions and organization of mathematical theorems and for proving them formally.

References

[1] Division by zero in type theory: a FAQ. `https://xenaproject.wordpress.com/2020/07/05/division-by-zero-in-type-theory-a-faq/`. Accessed: 2014-08-28.

[2] Xavier Allamigeon and Ricardo D. Katz. A formalization of convex polyhedra based on the simplex method, 2018.

[3] David Bartl. Farkas' lemma, other theorems of the alternative, and linear programming in infinite-dimensional spaces: a purely linear-algebraic approach. *Linear and Multilinear Algebra*, 55(4):327–353, July 2007.

[4] David Bartl. A very short algebraic proof of the Farkas lemma. *Mathematical Methods of Operations Research*, 75(1):101–104, December 2011.

[5] Gertrud Bauer. The Hahn-Banach Theorem for Real Vector Spaces, 2004.

[6] Mehta Bhavik and Dillies Yaël. Separation Hahn-Banach theorem. `https://github.com/leanprover-community/mathlib4/blob/ba034836fdc5fe55f9903781364a9a1a4cbb0f55/Mathlib/Analysis/NormedSpace/HahnBanach/Separation.lean#L184-L189`, 2022.

[7] Ralph Bottesch, Max W. Haslbeck, and René Thiemann. Farkas' lemma and Motzkin's transposition theorem. *Archive of Formal Proofs*, January 2019. `https://isa-afp.org/entries/Farkas.html`, Formal proof development.

[8] Ralph Bottesch, Alban Reynaud, and René Thiemann. Linear inequalities. *Archive of Formal Proofs*, June 2019. `https://isa-afp.org/entries/Linear_Inequalities.html`, Formal proof development.

[9] Sergei Nikolaevich Chernikov. Linear inequalities. *Nauka*, 1968.

[10] György Farkas. A Fourier-féle mechanikai elv alkalmazásai. *Mathematikai és Természettudományi Értesitö*, 12:457–472, 1894.

[11] György Farkas. Paraméteres módszer fourier mechanikai elvéhez. *Mathematikai és Physikai Lapok*, 7:63–71, 1898.

[12] D. Gale, H.W. Kuhn, and A.W. Tucker. Linear programming and the theory of games. In *Activity Analysis of Production and Allocation*, pages 317–329, 1951.

[13] Marie Kerjean and Assia Mahboubi. A formal Classical Proof of Hahn-Banach Theorem . In *TYPES 2019*, Oslo.

[14] Vladimir Kolmogorov, Johan Thapper, and Stanislav Živný. The power of linear programming for general-valued CSPs. *SIAM Journal on Computing*, 44(1):1–36, 2015.

[15] Jannis Limperg and Asta Halkjær From. Aesop: White-box best-first proof search for Lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP '23. ACM, January 2023.

[16] Filip Marić, Mirko Spasić, and René Thiemann. An incremental simplex algorithm with unsatisfiable core generation. *Archive of Formal Proofs*, August 2018. `https://isa-afp.org/entries/Simplex.html`, Formal proof development.

[17] The mathlib community. The Lean Mathematical Library. In Jasmin Blanchette and Cătălin Hritcu, editors, *CPP 2020*, pages 367–381. ACM, 2020.

[18] H. Minkowski. *Geometrie der Zahlen*. Teubner, Leipzig, 1910.

[19] Sara Negri, Jan Cederquist, and Thierry Coquand. The Hahn-Banach theorem in type theory. In *Twenty-five years of constructive type theory*, pages 57–72, United Kingdom, 1998. Oxford University Press.

[20] Bogdan Nowak and Andrzej Trybulec. Hahn-Banach Theorem. *Journal of Formalized Mathematics*, 1993.

[21] Robert Pollack. How to believe a machine-checked proof. *BRICS Report Series*, 4(18), January 1997.

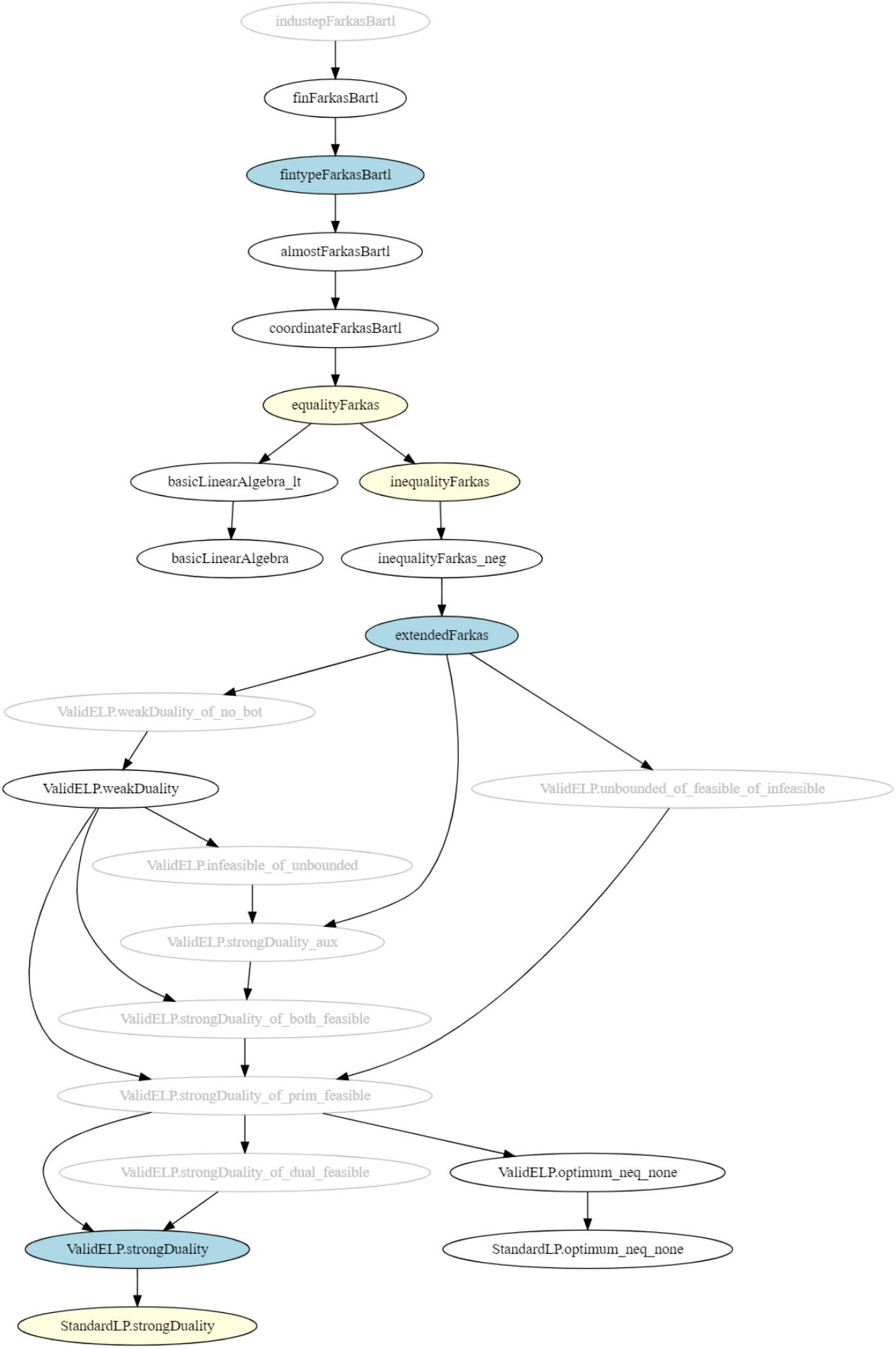
[22] K. Sakaguchi. Vass. `https://github.com/pi8027/vass`, 2016.

[23] René Thiemann. Duality of linear programming. *Archive of Formal Proofs*, February 2022. `https://isa-afp.org/entries/LP_Duality.html`, Formal proof development.

[24] Cherng tiao Perng. On a class of theorems equivalent to Farkas's lemma. *Applied Mathematical Sciences*, 11(44):2175–2184, 2017.

[25] J. von Neumann. Discussion of a maximum problem. Instutute for Advanced Study, Princeton, NJ, 1947.

Appendix: dependencies between theorems



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