

Linear Programming in Isabelle/HOL

Julian Parsert   

University of Oxford, United Kingdom

University of Innsbruck, Austria

Abstract

Linear programming describes the problem of optimising a linear objective function over a set of constraints on its variables. In this paper we present a solver for linear programs implemented in the proof assistant Isabelle/HOL. This allows formally proving its soundness, termination, and other properties. We base these results on a previous formalisation of the simplex algorithm which does not take optimisation problems into account. Using the weak duality theorem of linear programming we obtain an algorithm for solving linear programs. Using Isabelle's code generation mechanism we can generate an external solver for linear programs.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Linear Programming, Optimisation, Interactive Theorem Proving, Isabelle/HOL.

Acknowledgements We thank René Thiemann and Cezary Kaliszyk for their help. This work is supported by the European Research Council (ERC) grant no 714034 *SMART*.

1 Introduction

Linear programming is a methodology for solving certain types of optimisation problems. Linear programming as a part of operations research also has applications in many areas outside of pure mathematics and computer science. Examples of applications include but are not limited to: finance, transportation, management, etc. In computer science linear programming can be used in, for example, network optimisation and integer transition systems. Finally, our motivation for this work is its use in game theory. One can express a two player zero-sum game using a linear program. Solving this linear program is equivalent to solving the original game. Hence, linear programming can be used to solve two-player zero sum games [9, 17].

Due to its large amount of use cases many software suites ship with a solver for linear programs [5, 4], including popular software like Microsoft Excel. However, software is known to have bugs and undesirable behaviour and the aforementioned tools most certainly are no exception. Therefore, we believe that formal verification can be a useful tool to increase trust in the results of linear program solvers, especially when they are applied to fault critical areas. In this paper we discuss the use of a proof assistant to formalise the notion of linear programs and an algorithm for solving them. We use the proof assistant Isabelle/HOL. In particular, we formalise an algorithm for solving linear programs based on a previous formalisation of the general simplex method [14]. This algorithm is a reduction that reduces the optimisation problem to a constraint satisfaction problem. Our description of this reduction is stated in such a way that Isabelle's code generation mechanism can be utilised to generate a formally verified Haskell program which solves linear programs. To summarise, our contributions are as follows:

- We formalise linear programs using the proof assistant Isabelle/HOL and derive results related to the duality of linear optimisation.
- We describe an algorithm for solving linear programs and prove its soundness. This algorithm is stated in a way such that Isabelle's code generation mechanism can be used to obtain a verified executable program.

- In order to obtain the aforementioned results, we provide translations and equivalences between an Isabelle library for linear polynomials and one for linear algebra. We also provide correctness results for these translations.
- Using the generated program we solve an example game as a case study to show how games can be solved using linear programming.

Related Work:

Our work is based on a formalisation of the general simplex algorithm described in [8, 14]. However, the general simplex algorithm lacks the ability to optimise a function. Boulmé and Maréchal [3] describe a formalisation and implementation of Coq tactics for linear integer programming and linear arithmetic over rationals. More closely related is the formalisation by Allamigeon et al. [1] which formalises the simplex method and related results in Coq. As part of Flyspeck project Obua and Nipkow [10] created a verification mechanism for linear programs using the HOL computing library and external solvers.

Outline:

In Subsection 1.1 we introduce the proof assistant Isabelle/HOL as well as notation which will be used throughout this paper. Subsequently, in Section 2 we give a short overview of linear programming. In Section 3 we describe the formalisation and provide more details on the definitions, algorithms and theorems. In Section 4 we discuss the generated algorithm and some examples. Finally, in Section 5 we make concluding remarks and discuss future work.

1.1 Isabelle/HOL and Notation

We use the proof assistant Isabelle/HOL, which is based on simply typed higher-order logic. On top of the simple type system Isabelle/HOL provides type classes. We will also use Isabelle's standard `option` type with the constructors `Some` and `None` as well as the sum type denoted `+`. While the constructors of the sum type are `Inl` and `Inr`, we will use `Unsat` and `Sat` instead. We also use $\mathbb{N}$ and $\mathbb{Q}$ to denote Isabelle's natural and rational numbers as well as `$\alpha$  list` and `$(\alpha, \beta)$  mapping` for polymorphic lists and mappings from α to β . For a mapping M we use the notation $M_{[i]}$ to denote the value of i in M . We use $\bullet$ to denote the dot product between a row vector and a column vector. The symbols $\cdot_v$ and $\cdot^v$ describe the vector-matrix and matrix-vector multiplication respectively. The symbols $=$, $\leq$, and $\geq$ retain their standard semantics for scalars, while we use $=_{pw}$, $\leq_{pw}$, and $\geq_{pw}$ to denote the respective pointwise orders on vectors. Importantly, we also use a `constraint` type describing a constraint. The constructors of these are $[=]$, $[\leq]$, and $[\geq]$. To see the difference, note that while $x \leq y$ and $x \leq_{pw} A$ are of type `bool`, the expression $x [\leq] y$ is of type `constraint` which is the pair (x, y) in addition with one of the aforementioned constructors. We use $[m..n]$ and $\{m..n\}$ to denote lists and sets of elements from m to $n - 1$, while dropping the `<` symbol also includes n . Furthermore, $[f \ x \ . \ i \leftarrow L]$ is a short notation for `map f L`. The function `dim c`, `row A`, and `col A` return the dimension of a vector c and the number of rows and columns of a matrix A . The zero vector of dimension n is denoted 0_v^n . Vector and list concatenation are denoted with the operators $@_v$ and $@$ while $[a]$ is the singleton list containing a . $L!i$ and $V \$ i$ are list and vector access operators respectively. Both are zero indexed and are only well defined if $i < \text{length } L$ or $i < \text{dim } V$. The code snippets presented in the remainder of the paper have been formatted for readability omitting brackets, type annotations, etc.

2 Linear Programming

We will give a brief overview of linear programming. Parts of our formalisation are based on the textbook “Theory of Linear and Integer Programming” by Schrijver [13] which we also recommend for a more detailed presentation of the topic.

A linear program describes the problem where we have an objective function $f(x_1, \dots, x_n)$ that we want to optimise while the variables $x_1, \dots, x_n$ are subject to a set of constraints. These constraints can be an equality

$$\alpha_1 x_1 + \dots + \alpha_n x_n = b \quad (1)$$

or a non-strict inequality

$$\alpha_1 x_1 + \dots + \alpha_n x_n \geq b \quad (2)$$

$$\alpha_1 x_1 + \dots + \alpha_n x_n \leq b. \quad (3)$$

Note how Constraint 1 is equivalent to the combination of the Constraints 2 and 3. Furthermore, Constraint 2 is equivalent to $-(\alpha_1 x_1 + \dots + \alpha_n x_n) \leq -b$. For simplicity, we will only consider constraints of type 3 and 1 the latter of which we only keep for sake of readability.

Given a set of linear constraints one can pose the question of whether or not a variable assignment exists that satisfies these constraints. This decision problem does not take the optimisation of an objective function into account. An algorithm for deciding this is the *general simplex* algorithm. In case of success, we can obtain an arbitrary variable assignment that satisfies all constraints.

2.1 Linear Optimisation

After having introduced the general decision problem for the satisfaction of a list of constraints, we will now introduce linear programming. In linear programming we are not only interested in finding an arbitrary satisfying assignment but an assignment which is optimal with respect to a given (linear) objective function.

More precisely, a linear program is an objective function f which is subject to a list of constraints C :

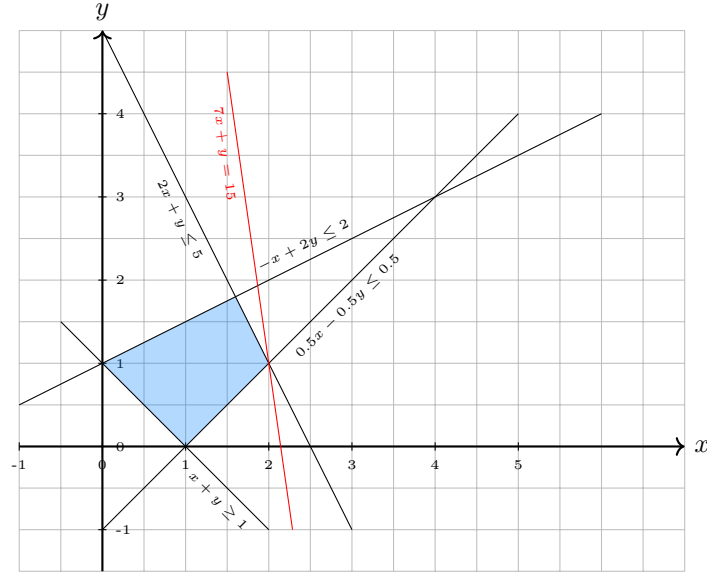
$$f(x_1, \dots, x_n) := c_1 * x_1 + \dots + c_n * x_n$$

$$C = \begin{bmatrix} A_{11} * x_1 + \dots + A_{1n} * x_n \leq b_1 \\ A_{21} * x_1 + \dots + A_{2n} * x_n \leq b_2 \\ \vdots \quad \quad \quad \vdots \\ A_{m1} * x_1 + \dots + A_{mn} * x_n \leq b_m \end{bmatrix}$$

This gives rise to a more concise notation for linear programs where $c_1, \dots, c_n$, $x_1, \dots, x_n$, $b_1, \dots, b_n$ are vectors and $A_{11}, \dots, A_{mn}$ is a matrix:

$$\begin{aligned} & \text{maximise } c \bullet x \\ & \text{subject to: } A \cdot_v x \leq_{pw} b \end{aligned} \quad (4)$$

Solving this linear program is searching for a satisfying assignment of variables that is optimal with respect to the function f . *Optimal* in this instance means either minimal or maximal. Hence, we are looking for an assignment for $x_1, \dots, x_n$ such that it satisfies the constraints



■ **Figure 1** A plot describing the optimisation problem presented in Example 1.

and $f(x_1, \dots, x_n)$ is maximal (minimal) in the set of all $f(y_1, \dots, y_n)$ where $y_1, \dots, y_n$ also satisfy the constraints. A concrete example of a linear program accompanied by a plot showing the objective function and constraints is presented in Example 1.

► **Example 1 (Linear Program).** Take the linear program consisting of the following objective function

$$f(x, y) := 7x + y$$

and the following set of constraints:

$$2x + y \leq 5 \quad -x + 2y \leq 2 \quad \frac{1}{2}x - \frac{1}{2}y \leq \frac{1}{2} \quad x + y \geq 1$$

Using basic transformations to transform the last inequality to one of the form of Inequality 3, we obtain the following matrix A , and vectors b and c :

$$A = \begin{bmatrix} 2 & 1 \\ -1 & 2 \\ \frac{1}{2} & -\frac{1}{2} \\ -1 & -1 \end{bmatrix}, \quad b = \begin{bmatrix} 5 \\ 2 \\ \frac{1}{2} \\ -1 \end{bmatrix}, \quad c = [7 \quad 1] \quad (5)$$

A plot of these inequalities is shown in Figure 1. The marked region (blue) is the region satisfying all constraints (i.e. feasible). The red line (i.e. equation $7x + y = 15$) describes the objective function going through the point $(2, 1)$ which is the optimal value within the feasible region and therefore the solution to the linear program described with A , b , and c .

2.2 Duality

The duality principle plays an important role in optimisation problems. It states that any optimisation problem, that is the *primal* problem, automatically defines a *dual* problem.

Furthermore, a solution to the dual problem is bounded by the solution of the primal problem. Our algorithm relies on this duality principle.

Since every linear program has a dual, we can write the dual of the Linear Program 4:

$$\begin{aligned} & \text{minimise } b^T \bullet y^T \\ & \text{subject to: } y \cdot^v A =_{pw} c, \ 0 \leq_{pw} y \end{aligned} \quad (6)$$

Note the switching of b and c in addition to the change from a maximisation to a minimisation problem. If the original problem is a maximisation problem, then the value (i.e. value of the optimised function at the optimal point) of the dual problem is an upper bound on the value of the primal problem. If the primal problem is a minimisation, then the dual provides a lower bound. In linear programming we know that these values are in fact equal. This is known as the Strong Duality Theorem (Theorem 2).

► **Theorem 2 (Strong Duality Theorem).** *Given linear constraints A , b , and the objective function c , we obtain x and y as the solutions to the primal and dual linear program, respectively. We can derive the following equality:*

$$c \bullet x = y \bullet b$$

To prove the correctness of our algorithm we only require the weak duality theorem, which is part of the formalisation and will be discussed in Theorem 7.

2.3 Solving Linear Programs

The most common algorithm to solve linear programs is the simplex algorithm. Unlike the *general simplex* algorithm this alternative simplex algorithm takes an objective function into account. However, we use a different approach. Using the duality theorem we can solve linear programs with the general simplex algorithm. This can be done by solving the constraints for the primal program (cf. Linear Program 4) and the dual program (cf. Linear Program 6) simultaneously:

$$A \cdot^v x \leq_{pw} b, \ y \cdot^v A =_{pw} c, \ y \geq_{pw} 0 \quad (7)$$

Due to the dual program constituting an upper bound, we know any satisfying assignment to x and y must satisfy $x \bullet c \leq y \bullet b$. Now we can add the constraint $x \bullet c \geq y \bullet b$ to the Constraints 7. Hence, by solving the following constraint satisfaction problem without explicitly maximising the objective function, we also solve the linear program:

$$x \bullet c \geq y \bullet b, \ A \cdot^v x \leq_{pw} b, \ y \cdot^v A =_{pw} c, \ y \geq_{pw} 0. \quad (8)$$

Any resulting assignment satisfying the Constraints 8 also satisfies $c \bullet x = y \bullet b$. Hence, x solves the Linear Program 4. Furthermore, we also derive that if a solution to the linear program exists this algorithm will find it, since no sub-optimal solutions get lost by adding the constraint $x \bullet c \geq y \bullet b$.

3 Formalisation

We base our work on two previous formalisation's which are part of the archive of formal proofs (AFP). The first is due to Spasić et al. [14, 8] who formalise the simplex algorithm used for checking the satisfiability of linear constraints. The second one is due to Thiemann

et al. [15, 16] and is a linear algebra library which allows us to create a relation between linear polynomials and matrices. All definitions and results in this section can be found in the formalisation unless specified otherwise.

The formalisation described in [8] provides a *sound* and *complete* implementation of the general simplex algorithm called **simplex** in Isabelle/HOL. The function **simplex** has type

`constraint list  $\Rightarrow$   $\mathbb{N}$  list + ( $\mathbb{N}$ ,  $\mathbb{Q}$ ) mapping.`

It produces either a variable assignment ($\mathbb{N}$, $\mathbb{Q}$) **mapping** (variables are modelled as naturals) satisfying the constraints or an unsatisfiable core ($\mathbb{N}$ list). At the current point in time the **simplex** algorithm only works on rational numbers. However, due to a change of the underlying libraries we will also be able to provide results for real numbers. From now on, we will simply use **simplex** as a subroutine without further consideration. For further details on this formalisation we refer to Marić et al.'s work [8].

3.1 Combining Representations

When formalising mathematics it is not uncommon to develop theories that combine existing definitions and representations. In particular, it is essential to develop methodologies that allow for switching between representations as some lend themselves better for certain tasks [6]. In our case we use two different representations for (lists of) linear polynomials.

The first representation is that used by **simplex** in [8]. Here, linear polynomials are defined as functions mapping variables to their coefficients. As variables are modelled with natural numbers, polynomials are functions of type $\mathbb{N} \Rightarrow \mathbb{Q}$ such that for each polynomial p the set $\{x \in \mathbb{N}. p\ x \neq 0\}$ is finite. The second representation is one using vectors and matrices. In particular, we use the linear algebra library described in [15]. Our motivation for combining these representations is that the vectors and matrices make stating and proving some properties easier.

First, we create a mechanism to transform vectors to function type polynomials. To this end we define a function **list_to_lpoly** that translates a list to a polynomial.

```
fun list_to_lpoly where
  list_to_lpoly cs =
    sum_list (map2 ( $\lambda i\ c.$  monom  $c\ i$ ) [0.. $\text{length}\ cs$ ] cs)
```

This function first creates a list of monomials where the index i is the vector and $cs[i]$ is the coefficient of variable i . Subsequently, we simply sum this list to obtain the function type polynomial. Now we get a function that creates linear polynomials from vectors:

```
vec_to_lpoly  $v$  = list_to_lpoly (list_of_vec  $v$ )
```

Going the other direction is a little bit more difficult. First, we define the dimension of a function-type polynomial p to be 0 if it is the zero polynomial and n if $p\ (n-1) \neq 0$ and $\forall i \geq n. p\ i = 0$. Using the vector constructor **vec** we define a function which transforms a linear polynomial into a vector;

```
lpoly_to_vec  $p$  = vec (dim_poly  $p$ ) (coeff  $p$ )
```

The curried function **coeff** p is a function that given $i \in \mathbb{N}$ returns the coefficient of i in the polynomial p .

The most important result of combining these representations of polynomials is Theorem 3.

► **Theorem 3** (`vec_to_lpoly` and `lpoly_to_vec` are (almost) inverses). *For any arbitrary linear polynomial p the equation*

$$(\text{vec_to_lpoly } (\text{lpoly_to_vec } p)) = p$$

holds. Since we lose information about the dimension of the original vector v if v ends in a sequence of zeroes we can only show the following two results:

$$\begin{aligned} \text{lpoly_to_vec } (\text{vec_to_lpoly } v) \ \$ \ i &= v \ \$ \ i \\ \dim (\text{lpoly_to_vec } (\text{vec_to_lpoly } v)) &\leq \dim v. \end{aligned}$$

Building on these definitions we also define the functions `matrix_to_lpolies` and `lpolies_to_matrix` which translate a matrix to a list of linear polynomials and vice versa. Having these two ways of representing linear polynomials we now use the vector/matrix representation for the remainder of the paper.

3.2 Creating Systems of Constraints

For the algorithm, we need to be able to create and solve a system of constraints as described in the Constraints as displayed in (8). Since we use the `simplex` subroutine as a solver, we only need to worry about creating the system of constraints. The Constraints in (8) describe two different vectors x and y with different constraints and a single intersection at the Constraint $x \bullet c \geq y \bullet b$. Since `simplex` only allows for the creation of a single solution of type $(\mathbb{N}, \mathbb{Q})$ mapping, we need to synthesise the vectors x and y in certain positions in this mapping. Hence, we introduce several definitions that allow for the creation of such constraints.

We know that the vector x has to be of length `dim c` and the vector y of length `dim b`. Assuming that the simplex subroutine terminates successfully with a resulting mapping `SOL` as a `SOLution`, we create constraints such that the first `dim c` elements in `SOL` constitute the vector x and the elements `SOL[dim c]` to `SOL[dim c + dim b - 1]` the vector y .

First, we encode the constraint $y \geq_{pw} 0$. To keep this as modular as possible we introduce the function `from_ind_geq`.

```
fun from_ind_geq ::  $\mathbb{N} \Rightarrow \text{vector} \Rightarrow \text{constraint list}$  where
  from_ind_geq ix v = [pi+ix ≥ vi. i ∈ [0.. < dim v]]
```

This allows us to specify a starting index i and a vector v , such that for all $j < \dim v$, `SOL[i+j] ≥ vi`. Therefore, given that we synthesise y in the second part of `SOL` the constraint $y \geq_{pw} 0$ can be expressed as the following:

$$\text{from_ind_geq } (\dim c) \ 0_v^{\dim b}$$

Next, we tackle the two sets of constraints $A \cdot_v x \leq_{pw} b$ and $y \cdot^v A =_{pw} c$. We will leverage the fact that $y \cdot^v A = A^T \cdot_v y^T$ in order to better represent the latter constraint. Since the two constraints are independent of each other, that is x and y do not interfere, we first introduce a way of stating them simultaneously. For that we introduce Definition 4. Note that this is a special case of a block diagonal matrix.

► **Definition 4** (Two block non interference). *Given two matrices $A^{m \times n}$ and $B^{a \times b}$ we define a matrix `two_block_non_interfere`,*

$$\text{two_block_non_interfere } A \ B = \begin{bmatrix} A_{11} & \dots & A_{1n} & & \\ \vdots & \ddots & \vdots & & 0 \\ A_{m1} & \dots & A_{mn} & & \\ & & & B_{11} & \dots & B_{1b} \\ & 0 & & \vdots & \ddots & \vdots \\ & & & B_{a1} & \dots & B_{ab} \end{bmatrix}$$

Using this definition we can show that matrix/vector multiplication of the first `col` A elements with A is independent from the multiplication of the last `col` B elements with B . This notion is captured with Theorem 5.

► **Theorem 5.** *Given matrices $A^{m \times n}$ and $B^{a \times b}$ and let x and y be n and b dimensional vectors respectively. Then we can show:*

$$\text{two_block_non_interfere } A \ B \cdot_v (x \ \mathcal{O}_v \ y) = (A \cdot_v x) \ \mathcal{O}_v (B \cdot_v y)$$

In order to state the constraints $A \cdot_v x \leq_{pw} b$ and $A^T \cdot_v y^T =_{pw} c$ simultaneously we create the following matrix:

$$\text{two_block_non_interfere } A \ A^T = \begin{bmatrix} A_{11} & \dots & A_{1n} & & \\ \vdots & \ddots & \vdots & & 0 \\ A_{m1} & \dots & A_{mn} & & \\ & & & A_{11} & \dots & A_{m1} \\ & 0 & & \vdots & \ddots & \vdots \\ & & & A_{1n} & \dots & A_{mn} \end{bmatrix}$$

Using `matrix_to_lpolies` we can translate this matrix to a list of polynomials L . Then, the list of constraints $A \cdot_v x \leq_{pw} b$ can be generated with:

$$[\text{L}li \ [\leq] \ b \ \$ \ i \ . \ i \leftarrow [0..<\text{dim } b]]$$

The second list of constraints $A^T \cdot_v y^T =_{pw} c$ is:

$$[\text{L}li \ [=] \ (b \ \mathcal{O}_v \ c) \ \$ \ i \ . \ i \leftarrow [\text{dim } b..<\text{dim } b + \text{dim } c]]$$

Combining these definitions we obtain the following Isabelle function which given a matrix A , and vectors b, c generates a list of constraints modelling $A \cdot_v x \leq_{pw} b$ and $A^T \cdot_v y^T =_{pw} c$.

```

fun mat_leq_eqc where
  mat_leq_eqc  $A \ b \ c =$ 
    let  $lst =$ 
      matrix_to_lpolies (two_block_non_interfere  $A \ A^T$ )
    in
      [  $lstli \ [\leq] \ b \ \$ \ i \ . \ i \leftarrow [0..<\text{dim } b] ] \ @$ 
      [  $lstli \ [=] \ (b \ \mathcal{O}_v \ c) \ \$ \ i \ . \ i \leftarrow [\text{dim } b..<\text{dim } b + \text{dim } c] ]$ 

```

Due to the use of `two_block_non_interfere` and by Theorem 5 the vectors x and y are generated in the correct positions.

Finally, we are left with the only constraint where x and y interfere, $x \bullet c \geq y \bullet b$.

```

fun xc_geq_yb where
  xc_geq_yb c b =
    vec_to_lpoly (c @v 0vdim b) [≥] vec_to_lpoly (0vdim c @v b)

```

This constraint ensures that after extracting x and y from the solution mapping the following condition holds ($n = \dim c$ and $m = \dim b$):

$$[c_0, \dots, c_{n-1}, 0_0, \dots, 0_{m-1}] \bullet (x @_v y) \geq [0_0, \dots, 0_{n-1}, b_0, \dots, b_{m-1}] \bullet (x @_v y)$$

This precisely corresponds to the constraint $x \bullet c \geq y \bullet b$. Now we have defined all the functions necessary to generate the System of Constraints 8 by concatenating the lists:

$$[\text{xc_geq_yb } c \ b] @ (\text{mat_leq_eqc } A \ b \ c) @ (\text{from_ind_geq } (\dim c) \ 0_v^{\dim b}) \quad (9)$$

Solving this list of constraints with the `simplex` procedure yields a mapping SOL of $\mathbb{N}$ to $\mathbb{Q}$. Using a simple split function `split_nm (dim c) (dim b) SOL` we obtain the pair of vectors (x, y) which satisfy the Constraints 9 and in turn 8.

3.3 Abstract Linear Programming

Having described a way of expressing the constraints in our setting, we now take a look at linear programming from an abstract point of view. That is, we will define necessary definitions and derive results that we use to formally prove the correctness of our algorithm.

First, we define the abstract notions of satisfying assignments for the primal and dual problems.

definition `sat_primal` $A \ b = \{x. A \cdot_v x \leq_{pw} b\}$
definition `sat_dual` $A \ c = \{y. y \cdot^v A =_{pw} c \wedge y \geq_{pw} 0\}$

In addition we define the notion of optimality.

definition `optimal_LP` $f \ S \ c = \{x \in S. (\forall y \in S. f \ (y \bullet c) \ (x \bullet c))\}$

Here f is a function of type $\alpha \Rightarrow \alpha \Rightarrow \mathbb{B}$ which usually defines an order, S is a set of polynomials to optimise over, and c is the objective function represented as a vector. Combining these we get the maximisation problem `max_LP`

`optimal_LP` $(\leq) \ (\text{sat_primal } A \ b) \ c$

and its dual minimisation problem `min_LP`

`optimal_LP` $(\geq) \ (\text{sat_dual } A \ c) \ b.$

Next we want to prove the weak duality theorem for `max_LP` and `min_LP`. To this end we first create an abstract environment using Isabelle's *locale* mechanism [2] which also allows us to state assumptions:

locale `abstract_LP` =
fixes $A \ b \ c$
assumes $A \in \mathbb{F}^{m \times n}$ **and** $b \in \mathbb{F}^m$ **and** $c \in \mathbb{F}^n$

Note that since we are only conducting abstract reasoning without a concrete algorithm yet, we do not restrict ourselves to $\mathbb{Q}$ or $\mathbb{R}$. Furthermore, the underlying type-class of $\mathbb{F}$ is a linearly ordered commutative semiring. Within this environment we can prove Lemma 6 and Theorem 7. The proof of the former can be found in the formalisation under the name `weak_duality_aux`.

► **Lemma 6.** *If x and y are solutions to the primal and dual problem respectively, then*

$$c \bullet x \leq b \bullet y.$$

► **Theorem 7** (Weak Duality Theorem). *If $x \in \max_LP$ and $y \in \min_LP$ then we can show*

$$x \bullet c \leq y \bullet b.$$

Proof. From the definition of $\max_LP$ we know that $A \cdot_v x \leq_{pw} b$. And since we also know that $y \geq_{pw} 0$ (from $\min_LP$), we can show $y \cdot_v A \cdot_v x \leq y \bullet b$. Furthermore, from the definition of $\min_LP$ we have $y \cdot_v A =_{pw} c$ and hence $c \bullet x = y \cdot_v A \cdot_v x$. Putting these together we get, $c \bullet x = y \cdot_v A \cdot_v x \leq y \bullet b$. Note that the assumptions in the *locale* guarantee that the dimensions of the vectors/matrices are correct and allow for the dot product to be commutative, thus proving $x \bullet c \leq y \bullet b$. ◀

3.4 Final Algorithm

With the necessary definitions and results we are now able to describe the algorithm and prove its soundness. As explained above we will be using the general simplex algorithm `simplex` which has previously been formalised and proven correct within Isabelle. With `simplex` as a subroutine we write the following function.

```
fun create_optimal_solution where
  create_optimal_solution A b c =
    let cs =
      [xc_geq_yb c b] @
      mat_leq_eqc A b c @
      from_ind_geq (dim c) 0_v^dim b
    in
      case simplex cs of
        | Unsat S ⇒ Unsat S
        | Sat S ⇒ Sat (split_nm (dim c) (dim b) S)
```

We first create a list of constraints *cs* which describes the System of Constraints 8 (cf. Listing 9). Subsequently, we use the general simplex algorithm `simplex` to obtain an arbitrary variable assignment that satisfies the constraints *cs*. If a satisfying assignment exists, we split the resulting assignment and create a pair of vectors (x, y) where x and y satisfy the System of Constraints 8. It is important to note that this algorithm assumes the inputs A , b , and c to be of the right dimensions (i.e. the dimensions assumed in the locale `abstract_LP`). Furthermore, since this is now a concrete algorithm the elements of A , b , and c are assumed to be rational numbers again.

Finally, in order we get a general algorithm without having to rely on the locale assumptions, we simply add a check for the dimensions of the input and if this fails we return `None` and otherwise the result of `create_optimal_solution`.

```
fun maximize where
  maximize A b c =
    if dim b = row A ∧ dim c = row A then
      Some (create_optimal_solution A b c)
    else
      None
```

With this algorithm we prove the following theorem without any locale assumptions, that is without any assumptions outside the one specifically stated.

theorem soundness

assumes `maximize A b c = Some (Sat (x, y))`

shows $x \in \text{max_lp } A \ b \ c$

Proof. Since the result is `Some (Sat (x, y))` we know the dimensions of the input were correct. From the constraints created in `create_optimal_solution` we know that for x , the following constraints hold:

$$A \cdot_v x \leq_{pw} b$$

Therefore, by the definition of `sat_primal` we have $x \in \text{sat_primal } A \ b$. Conversely, for y we have:

$$y \cdot^v A =_{pw} c \text{ and } y \geq_{pw} 0$$

Now, by definition of `sat_dual` we have $y \in \text{sat_dual } A \ c$. From Theorem 7 we derive that no matter if x or y optimise their respective objective function, as long as $x \in \text{sat_primal } A \ b$ and $y \in \text{sat_dual } A \ c$ the inequality $x \bullet c \leq y \bullet b$ must hold. Finally, due to `xc_geq_yb c b` we also have that x and y must satisfy $x \bullet c \geq y \bullet b$. Combining these we get $x \bullet c = y \bullet b$. By Lemma 6 all $v \in \text{sat_primal } A \ b$ must obey $v \bullet c \leq y \bullet b$, leading to $v \bullet c \leq x \bullet c$. Hence, x is optimal, that is $x \in \text{max_lp } A \ b \ c$. ◀

4 Code Generation and Examples

Using Isabelle's code generation mechanism [7] we can generate code for the `maximize` function. Isabelle by default allows for the generation of code in Haskell, SML, OCaml, and Scala. Using this generated code we get the function `maximize` in the objective language. Using Haskell, as an example¹, we can implement a simple parser to create a program that takes a matrix and two vectors as input and calculates the solution to this linear program using `maximize`:

```
solveLP :: (String, String, String) -> Maybe ([Nat] + (Vec Rat))
solveLP (a, b, c) = maximize matA vecB vecC where
  matrix = parseMatrix a
  mRows = (nat_of_int (maximum (map length matrix)))
  matA = mat_of_cols_list mRows matrix
  vecB = parseListToVec b
  vecC = parseListToVec c
```

The compiled Haskell program can be used to solve Example 1 with the Constraints 5 as input. The result is the vector $[2 \ 1]$. Hence $7 * 2 + 1 * 1 = 15$ is the maximum value. An example closer to our intended application is Example 8.

► **Example 8** (Solving Rock Paper Scissors). The commonly known game of Rock-Paper-Scissors can be modelled with the following payoff matrix:

¹ Any of the aforementioned languages could be used.

	Rock	Paper	Scissors
Rock	0	-1	1
Paper	1	0	-1
Scissors	-1	1	0

This payoff matrix is to be interpreted as follows: If player one (rows) plays paper and player two (columns) plays rock then the payoff for player one has a payoff of 1 meanwhile player two has the payoff of -1 (i.e. player one wins and player two loses). Since the sum of the payoffs always equals 0, this is a zero sum game. Encoding the strategies rock, paper, and scissors, in x_1 , x_2 , and x_3 we can encode this game into the following linear program:

$$\begin{aligned}
& \text{maximise } u \\
& \text{subject to: } u \leq -x_2 + x_3 \\
& \quad u \leq x_1 - x_3 \\
& \quad u \leq -x_1 + x_2 \\
& \quad x_1 + x_2 + x_3 = 1 \\
& \quad 0 \leq x_1, 0 \leq x_2, 0 \leq x_3
\end{aligned} \tag{10}$$

The first three constraints encode the payoff matrix where u is the payoff (i.e. utility). Hence, $u \leq -x_2 + x_3$ implies that the payoff u cannot be higher than the sum of the utilities for playing rock combined (0, -1 , 1). Similarly, we encode the other strategies. The last two lines of constraints ensure that the resulting assignment is a probability distribution over $\{x_1, x_2, x_3\}$. After transforming these to matrix form we can use the extracted program to calculate the following vector:

$$[0, \frac{1}{3}, \frac{1}{3}, \frac{1}{3}]$$

Meaning the expected payoff of playing the optimal strategy is 0, and the optimal strategy is the mixed strategy of playing x_1 , x_2 , and x_3 with equal probability ($\frac{1}{3}$). Hence, playing rock, paper, or scissors each with a probability of $\frac{1}{3}$ is the optimal strategy of this game.

For small game theory examples such as Example 8 the generated algorithm performs well and produces a result instantly. However, we did not tamper with the underlying formalisation of the simplex algorithm [14] which famously has exponential worst case complexity but behaves well most of the time. Hence our algorithm exerts the same asymptotic complexity as the underlying general simplex algorithm. However, since we introduce different kinds of constraints the number of constraints roughly doubles. Neither our reduction nor the underlying algorithm was formalised with efficiency or competitiveness in constraint solving in mind. Hence, we did not conduct any experiments comparing the generated code with existing off the shelf constraint solvers.

5 Conclusion and Future Work

We presented the formalisation of an algorithm for solving linear programs. This work is based on previous formalisation's of the general simplex algorithm as well as a matrix library. The previous formalisation only considers the satisfaction of linear constraints and does not allow for optimising an objective function. We improved upon this by incorporating optimisation. Linear programming (i.e. linear optimisation) has many applications in many different fields. In our case the motivation is its use in game theory where linear programming

can be used to solve two player zero-sum games. In this paper we present a formalisation of an algorithm that solves linear programs as well as some results derived from it. Furthermore the algorithm is described in such a way that Isabelle’s code generation mechanism can be used to generate executable code providing a verified solver for linear programs. The formalisation is part of the Archive of Formal Proofs [12].

Although the algorithm is formally proven to be sound within the proof assistant, a completeness proof is sketched in this paper but does not exist in a formalised manner, yet. We leave this for future work. Furthermore, we plan on using this formalisation to produce a verified solver for zero-sum two player games as part of a game theory [11] formalisation effort.

References

- 1 Xavier Allamigeon and Ricardo D. Katz. A formalization of convex polyhedra based on the simplex method. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving*, pages 28–45, Cham, 2017. Springer International Publishing.
- 2 Clemens Ballarín. Locales: A module system for mathematical theories. *Journal of Automated Reasoning*, 52(2):123–153, Feb 2014. doi:10.1007/s10817-013-9284-7.
- 3 Sylvain Boulmé and Alexandre Maréchal. A Coq tactic for equality learning in linear arithmetic. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving*, pages 108–125, Cham, 2018. Springer International Publishing.
- 4 Ambros M. Gleixner, Daniel Steffy, and Kati Wolter. Improving the accuracy of linear programming solvers with iterative refinement. Technical Report 12-19, ZIB, Takustr. 7, 14195 Berlin, 2012.
- 5 GNU Linear Programming Kit (GLPK). <https://www.gnu.org/software/glpk/glpk.html>, 2019. Online; accessed 2 Mai 2019.
- 6 Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O’Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 163–179, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- 7 Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming*, pages 103–117, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- 8 Filip Marić, Mirko Spasić, and René Thiemann. An incremental simplex algorithm with unsatisfiable core generation. *Archive of Formal Proofs*, August 2018. <http://isa-afp.org/entries/Simplex.html>, Formal proof development.
- 9 Michael Maschler, Eilon Solan, and Shmuel Zamir. *Game Theory*. Cambridge University Press, 2013. doi:10.1017/CB09780511794216.
- 10 Steven Obua and Tobias Nipkow. Flyspeck II: the basic linear programs. *Annals of Mathematics and Artificial Intelligence*, 56(3):245–272, Aug 2009. doi:10.1007/s10472-009-9168-z.
- 11 Julian Parsert and Cezary Kaliszyk. Towards formal foundations for game theory. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018*, volume 10895 of *LNCIS*, pages 495–503. Springer, 2018. doi:10.1007/978-3-319-94821-8_29.
- 12 Julian Parsert and Cezary Kaliszyk. Linear programming. *Archive of Formal Proofs*, August 2019. https://isa-afp.org/entries/Linear_Programming.html, Formal proof development.
- 13 Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.
- 14 Mirko Spasić and Filip Marić. Formalization of incremental simplex algorithm by stepwise refinement. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, pages 434–449, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

- 15 René Thiemann and Akihisa Yamada. Formalizing Jordan normal forms in Isabelle/HOL. In Jeremy Avigad and Adam Chlipala, editors, *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 88–99. ACM, 2016. URL: <http://dl.acm.org/citation.cfm?id=2854065>, doi:10.1145/2854065.2854073.
- 16 René Thiemann and Akihisa Yamada. Matrices, Jordan normal forms, and spectral radius theory. *Archive of Formal Proofs*, August 2015. http://isa-afp.org/entries/Jordan_Normal_Form.html, Formal proof development.
- 17 W.L. Winston and J.B. Goldberg. *Operations Research: Applications and Algorithms*. Thomson Brooks/Cole, 2004. URL: <https://books.google.at/books?id=tg5DAQAAIAAJ>.