

Proving Bounds for Real Linear Programs in Isabelle/HOL

Steven Obua*

Technische Universität München
D-85748 Garching, Boltzmannstr. 3, Germany
e-mail: obua@in.tum.de, url: <http://www4.in.tum.de/~obua>

Abstract. Linear programming is a basic mathematical technique for optimizing a linear function on a domain that is constrained by linear inequalities. We restrict ourselves to linear programs on bounded domains that involve only real variables. In the context of theorem proving, this restriction makes it possible for any given linear program to obtain certificates from external linear programming tools that help to prove arbitrarily precise bounds for the given linear program. To this end, an explicit formalization of matrices in Isabelle/HOL is presented, and how the concept of lattice-ordered rings allows for a smooth integration of matrices with the axiomatic type classes of Isabelle.

As our work is a contribution to the Flyspeck project, we argue that with the above techniques it is now possible to prove bounds for the linear programs arising in the proof of the Kepler conjecture sufficiently fast.

1 Introduction and Motivation

The *Flyspeck project* [3] has as its goal the complete formalization of Hales' proof [2] of the Kepler conjecture. The formalization has to be carried out within a mechanical theorem prover. For our work described in this paper, we have chosen the generic proof assistant Isabelle, tailored to Higher-Order Logic (HOL) [4]. In the following, we will refer to this environment as *Isabelle/HOL*.

An important step in Hales' proof is the maximization of about 10^5 *real linear programs*. The size of these linear programs (LPs) varies; the largest among them consist of about 2000 inequalities in about 200 variables. The considered LPs have the important property that there exist a priori bounds on the range of the variables. The situation is further simplified by our attitude towards the linear programs: we only want to know whether the objective function of a given LP is bounded from above by a given constant K .

Under these assumptions, Hales describes [1] a method for obtaining an arbitrarily precise upper bound for the maximum value of the objective function of an LP. We will show that this method still works nicely in the context of mechanical theorem provers. The burden of calculating the upper bound is delegated

* Supported by the Ph.D. program "Logik in der Informatik" of the "Deutsche Forschungsgemeinschaft."

to an LP solver that needs not to be trusted. Instead, the LP solver delivers a small certificate to Isabelle/HOL that can be checked cheaply. Furthermore, there is no need to delve into the details of the actual method of optimizing an LP, which is usually the Simplex method. These details just do not matter for the theorem prover.

In this paper we describe all relevant issues and notions that arise in implementing the method in Isabelle/HOL. Although our point of view is necessarily influenced by the capabilities and restrictions of Isabelle/HOL, we think that the results are of independent interest, and we try to present them that way.

We first describe the basic idea of the method. Then we define the notion of *finite matrices* and explain why these are our representation of choice for linear programs. Finite matrices can be fitted into the system of numeric axiomatic type classes in Isabelle/HOL via the algebraic concept of *lattice-ordered rings*, and we take a short look at the changes of the hierarchy of type classes in Isabelle/HOL that were necessary for this. Checking the certificate from the external LP solver is basically a calculation involving finite matrices, and the matrices we have to deal with coming from our Flyspeck background are sparse, therefore we present a sparse matrix representation of finite matrices and formalize operations like sparse matrix multiplication.

2 The Basic Idea

There are quite a lot of different ways to state a linear programming problem [5, sect. 7.4], which are all general in the sense that *every* linear programming problem can be stated that way. Here is one such way: a linear program consists of a matrix $A \in \mathbb{R}^{m \times n}$, a row vector $c \in \mathbb{R}^{1 \times n}$ and a column vector $b \in \mathbb{R}^{m \times 1}$. The goal is to maximize the objective function

$$x \mapsto cx, \quad x \text{ feasible}, \quad (1)$$

where x is called *feasible* iff $x \in \mathbb{R}^{n \times 1}$ and $Ax \leq b$ holds. Note that we are dealing with matrix inequality here: $X \leq Y$ for two matrices X and Y iff every matrix element of X is less than or equal to the corresponding element of Y .

Usually, the above stated goal really encompasses several goals / questions:

1. Find out if there exists any feasible x at all (otherwise the LP is called *infeasible*).
2. Find out if there is a feasible x_{\max} such that $cx_{\max} \geq cx$ for any feasible x , and calculate this x_{\max} .
3. Calculate $M = \sup \{cx \mid x \text{ is feasible}\}$.

Note that $M = -\infty$ iff the answer to the first question is no. And $M = \infty$ iff the answer to the first question is yes and the answer to the second question is no. If $M < \infty$ then the LP is called *bounded*. Linear programming software is good at answering all those questions and at exhibiting (approximately) such an x_{\max} if it exists. Our goal is more modest in some ways, but more demanding in others:

assuming a priori bounds for the feasible region, that is assuming $l \leq x \leq u$ for all feasible x with a priori known bounds l and u , actually prove within Isabelle/HOL that $M \leq K$, where we can choose K arbitrarily close to M . In particular, we do not want to calculate x_{\max} , but just want to approximate M as precise as we wish for. Furthermore, we can assume $M \neq \infty$ because of

$$M \leq \sum_{i=1}^n |c_{1i}| \max\{|l_{i1}|, |u_{i1}|\} < \infty . \quad (2)$$

It might seem that our goal can be accomplished trivially by setting K to the above sum. But of course this is not the case, as K is probably not a particularly good approximation for M , and there is nothing in the above inequality telling us how to get a better approximation in case we need one.

2.1 Reducing the case $M = -\infty$ to the case $-\infty < M < \infty$

The case of an infeasible LP can be reduced to the case of a feasible LP [1]. We will give a more detailed description here than the one found in [1].

Remember that we are only considering LPs for which we know l and u s.t.

$$Ax \leq b \implies l \leq x \leq u . \quad (3)$$

In this subsection we additionally require A to fulfill the inequality

$$Ax \leq 0 \implies x = 0 . \quad (4)$$

This can easily be arranged by replacing A and b by \tilde{A} and \tilde{b} where

$$\tilde{A} = \begin{pmatrix} A \\ I_n \\ -I_n \end{pmatrix} \quad \text{and} \quad \tilde{b} = \begin{pmatrix} b \\ u \\ -l \end{pmatrix} . \quad (5)$$

$I_n \in \mathbb{R}^{n \times n}$ denotes the identity matrix.

Now let us assume that for the given LP both (3) and (4) hold. We can construct for any $K \in \mathbb{R}$ a modified LP with objective function

$$x' = \begin{pmatrix} x \\ t \end{pmatrix} \mapsto cx + Kt, \quad x' \text{ feasible}, \quad (6)$$

where $x' \in \mathbb{R}^{n+1}$ is called feasible with respect to the modified LP iff

$$Ax + tb \leq b \quad \text{and} \quad 0 \leq t \leq 1 . \quad (7)$$

Lemma 1.

$$\begin{pmatrix} x \\ 1 \end{pmatrix} \text{ is feasible} \iff x = 0 , \quad (8)$$

$$0 \leq t < 1 \implies \left(\begin{pmatrix} x \\ t \end{pmatrix} \text{ is feasible} \iff x/(1-t) \text{ is feasible} \right) . \quad (9)$$

On the left hand side of above equivalences we talk about feasibility with respect to the modified LP, on the right hand side about feasibility with respect to the original LP.

Proof. To show (8) in the direction from left to right one needs the fact that A fulfills (4). The rest is obvious by just expanding the respective definition of feasibility. \square

Lemma 2. *Defining $M' := \sup \{c x + K t \mid x' = \begin{pmatrix} x \\ t \end{pmatrix}, x' \text{ feasible}\}$ yields*

$$-\infty < \max \{M, K\} = M' < \infty . \quad (10)$$

As a special case follows

$$M = -\infty \implies M' = K . \quad (11)$$

Proof. Because of (8) we have $M' \geq K$, in particular $M' > -\infty$. Considering $t = 0$ in (9) gives us $M' \geq M$. From (9) and (3) in the case $t \neq 1$ and (8) in the case $t = 1$ we obtain bounds for x' :

$$x' = \begin{pmatrix} x \\ t \end{pmatrix} \text{ is feasible} \implies l^- \leq (1-t)l \leq x \leq (1-t)u \leq u^+ .$$

Here l^- denotes the *negative part* of l which results from l by replacing every positive matrix element by 0. Similarly, the *positive part* u^+ results from u by replacing every negative element by 0. We conclude $M' < \infty$.

So far we have shown $-\infty < \max \{M, K\} \leq M' < \infty$. To complete the proof, we need to show $\max \{M, K\} \geq M'$. We will proceed by case distinction.

Assume $M \geq K$. We show that for any feasible $x' = \begin{pmatrix} x \\ t \end{pmatrix}$, $M \geq c x + K t$, and therefore $M \geq M'$. This is obvious in the case $t = 1$, the feasibility of x' accompanied by the equivalence (8) forces x to be zero. In the case $t \neq 1$, (9) implies that $x/(1-t)$ is feasible with respect to the original LP. But this is just what we claim:

$$M \geq c(x/(1-t)) \implies M \geq c x + t M \geq c x + t K .$$

Now assume $M < K$. Assume further $M' > K$. Because of $-\infty < M' < \infty$ there is a feasible $x' = \begin{pmatrix} x \\ t \end{pmatrix}$ s.t. $M' = c x + K t$. For $t = 1$ we would have again $x = 0$ and therefore the contradiction $K < M' = K$. Finally $0 \leq t < 1$ also leads to a contradiction:

$$M' = c x + K t \leq c x + M' t \implies M' \leq c(x/(1-t)) \leq M < K \leq M' .$$

Therefore the only possibility is $M' \leq K$. \square

From now on we will assume that we are dealing with feasible, bounded LPs, that is with LPs for which we know $-\infty < M < \infty$.

2.2 The case $-\infty < M < \infty$

This case is the heart of the method. Again we construct a modified LP. The original LP is called the *primal* LP, the modified LP is called the *dual* LP. The objective function of the dual LP

$$y \mapsto yb, \quad y \text{ feasible,}$$

is to be minimized. Here $y \in \mathbb{R}^{1 \times m}$ is called feasible iff $yA = c$ and $y \geq 0$ holds.

Lemma 3. *Any feasible y induces an upper bound on M :*

$$yb \geq M . \quad (12)$$

Proof. For any feasible x we have

$$yb \geq y(Ax) = (yA)x = cx . \quad (13)$$

□

But is there such a feasible y so that we can utilize (12)? And if there is, can we accomplish $yb = M$ by carefully choosing y ? The well-known answer to both questions is yes:

Lemma 4. *Define $M' := \inf \{yb \mid yA = c \text{ and } y \geq 0\}$. Then*

$$-\infty < M = M' < \infty . \quad (14)$$

Furthermore, choose a feasible y such that $M' = yb$. Then

$$\text{card} \{i \in \mathbb{N} \mid 1 \leq i \leq m \text{ and } y_{1i} > 0\} \leq n . \quad (15)$$

Proof. Corollary 7.1g and 7.1l in [5]. □

Now the basic idea of our method can be described as follows. First, form the dual LP. Then use an external LP solver to solve the dual LP for an optimal y . This optimal y serves as a *certificate*. In our application, where typically $m \approx 2000$ and $n \approx 200$, y will be *sparse*, as inequality (15) tells us. Finally, use (12) to verify our desired upper bound $M \leq K = yb$.

This basic idea is complicated by the fact that we are dealing with *real* data and *numerical* algorithms. The external LP solver does not return an y such that $yA = c$ and $y \geq 0$, but rather an y such that $yA \approx c$ and $y \gtrsim 0$. In order to obtain a provably upper bound on M , one has to take (3) into account. Furthermore the input data A , b and c need not to be given as exact numerical data either, for example an element of A could equal π .

The rest of the paper will discuss the implementation in Isabelle/HOL of the method outlined here and will also deal with the mentioned complications.

3 Finite Matrices

Somebody who wants to implement the method outlined in the previous section faces up to the problem of how to represent linear programs. This problem is prominent outside of the realm of mechanical theorem proving, too: designers of linear programming packages typically provide various ways of input of data to the LP algorithms these packages provide, one can normally choose at least between dense and sparse representations of the data. The issue is to provide a certain convenience of dealing with the data without compromising the efficiency of the LP algorithms by too much overhead.

Our situation is different: we need to reason within our mechanical theorem proving environment Isabelle/HOL why our computations lead to a correct result, therefore we need a good representation of LPs for reasoning about them. Of course we also need to compute efficiently. But we should avoid mixing up those two issues if we can. The reasoning in the previous section has used matrices and the properties of matrix operations like associativity of matrix multiplication extensively. Therefore representing LPs within Isabelle/HOL as matrices is a good idea.

So how exactly does one represent matrices in higher-order logic? Obviously, matrices should be a type, but how does one deal with the dimension of a matrix? HOL does not have *dependent types*, so it seems impossible to have a parametrized family of types where the dimension of the matrix would be the parameter. But it is: one possibility that is pursued by John Harrison in the 2005 version of his Hol-light system is to represent the needed parameter by type variables! He uses this representation in order to formalize multivariate calculus. But in our case this idea cannot be used without causing serious problems later when we turn our attention to sparse matrices.

Another possibility is to represent the dimension of a matrix by a predicate that carves the set of all matrices of this dimension out of a certain bigger, already existing type. This is a common technique to overcome the absence of dependent types in HOL [10]. This approach could work like this:¹

```

type  $\alpha$  M = nat  $\times$  nat  $\times$  (nat  $\Rightarrow$  nat  $\Rightarrow$   $\alpha$ )

constdef
  Mequiv :: ( $\alpha$  M  $*$   $\alpha$  M) set
  Mequiv  $\equiv$   $\{((m, n, f), (m, n, g)) \mid \forall j i. (j < m \wedge i < n) \longrightarrow f j i = g j i\}$ 
typedef  $\alpha$  matrix = UNIV // Mequiv
constdef
  is-matrix :: nat  $\Rightarrow$  nat  $\Rightarrow$   $\alpha$  matrix  $\Rightarrow$  bool
  is-matrix m n A  $\equiv$   $\exists f. (m, n, f) \in \text{Rep-matrix } A$ .

```

¹ Here and in the following we deviate slightly from actual Isabelle/HOL syntax for various reasons, the most important being formatting; the actual Isabelle/HOL user will have no difficulty translating the given theory snippets to proper Isabelle/HOL syntax.

In (16) α *matrix* is the bigger type, and *is-matrix* m n acts as the predicate that carves out all matrices consisting of m rows and n columns. Here matrices are modelled as equivalence classes [7] of triples (m, n, f) where m denotes the number of rows, n the number of columns and f a function from indices to matrix elements. The set of these equivalence classes is denoted by *UNIV // Mequiv*. With this formalization of matrices an error element comes for free: there is exactly one matrix *Error* such that

$$\text{is-matrix } 0 \ 0 \ \text{Error} \quad (17)$$

holds. When adding matrices A and B which fulfill

$$\exists m \ n . (\text{is-matrix } m \ n \ A) \wedge (\neg \text{is-matrix } m \ n \ B) \quad (18)$$

and when multiplying matrices A and B for which

$$\exists m \ n \ u \ v . (\text{is-matrix } m \ n \ A) \wedge (\text{is-matrix } u \ v \ B) \wedge (n \neq u) \quad (19)$$

holds, the matrix *Error* is returned to signal that the operands do not belong to the natural domain of addition and multiplication, respectively.

Still, this approach is not entirely satisfying: in Isabelle/HOL there exists a large number of theorems that are valid for types that form a group or a ring. The fact that a type can be viewed as such an algebraic structure is formulated via the concept of *axiomatic type classes* [6]. But α *matrix* in (16) with the suggested error signaling definition of addition does not even form a group, because there is no matrix *Zero* with

$$\forall A . A + \text{Zero} = A \ , \quad (20)$$

but rather a whole family *Zero* m n such that

$$\forall A . \text{is-matrix } m \ n \ A \longrightarrow A + (\text{Zero } m \ n) = A \ . \quad (21)$$

Therefore we advocate a different approach that exploits the fact that the matrix elements commonly used in mathematics [11] themselves carry an algebraic structure, that of a ring, which always contains a zero. We define α *matrix* to be the type formed by all infinite matrices that have only finitely many non-zero elements of type α :

$$\begin{aligned} \text{type } \alpha \text{ infmatrix} &= \text{nat} \Rightarrow \text{nat} \Rightarrow \alpha \\ \text{typedef } \alpha \text{ matrix} &= \{f :: (\alpha :: \text{zero})\text{infmatrix} \mid \text{finite}\{(j, i) \mid f \ j \ i \neq 0\}\} \ . \end{aligned} \quad (22)$$

Hence we choose the name *finite matrices* for objects of type α *matrix*. Note the restriction $\alpha :: \text{zero}$ in (22). This means that the elements of a matrix cannot have just any type but only a type that is an instance of the axiomatic type class *zero* and has thus an element denoted by 0. Of course this is not a real restriction on the type; any type can be declared to be an instance of the axiomatic type class *zero*.

3.1 Dimension of a Finite Matrix

The dimension of a finite matrix deviates from the notion of dimension that one is used to. Because we did *not* encode the number of rows and columns explicitly in the representation of a finite matrix as we did in (16), we have to recover the dimension of a finite matrix by extensionality:

constdefs

$$\begin{aligned}
\text{nrows} &:: \alpha \text{ matrix} \Rightarrow \text{nat} \\
\text{nrows } A &\equiv \text{LEAST } m. \forall j i. m \leq j \longrightarrow (\text{Rep-matrix } A j i = 0) \\
\text{ncols} &:: \alpha \text{ matrix} \Rightarrow \text{nat} \\
\text{ncols } A &\equiv \text{LEAST } n. \forall j i. n \leq i \longrightarrow (\text{Rep-matrix } A j i = 0) \\
\text{is-matrix} &:: \text{nat} \Rightarrow \text{nat} \Rightarrow \alpha \text{ matrix} \Rightarrow \text{bool} \\
\text{is-matrix } m n A &\equiv \text{nrows } A \leq m \wedge \text{ncols } A \leq n .
\end{aligned} \tag{23}$$

The expression $\text{LEAST } x. P x$ equals the least x such that $P x$ holds. The definition of the type $\alpha \text{ matrix}$ has introduced two automatically defined functions Rep-matrix and Abs-matrix

consts

$$\begin{aligned}
\text{Rep-matrix} &:: \alpha \text{ matrix} \Rightarrow \alpha \text{ infmatrix} \\
\text{Abs-matrix} &:: \alpha \text{ infmatrix} \Rightarrow \alpha \text{ matrix}
\end{aligned} \tag{24}$$

that convert between finite matrices and infinite matrices. They enjoy the following crucial properties:

$$(A = B) = (\forall j i. \text{Rep-matrix } A j i = \text{Rep-matrix } B j i) , \tag{25}$$

$$\exists_1 f. A = \text{Abs-matrix } f , \tag{26}$$

$$\text{Abs-matrix } (\text{Rep-matrix } A) = A , \tag{27}$$

$$\text{finite } \{(j, i) \mid \text{Rep-matrix } A j i \neq 0\} , \tag{28}$$

$$\text{finite } \{(j, i) \mid f j i \neq 0\} \Longrightarrow \text{Rep-matrix } (\text{Abs-matrix } f) = f . \tag{29}$$

Thus $\text{Rep-matrix } A j i$ denotes the matrix element of A in row j and column i . Note that the first row is row 0, likewise for columns.

Let us return to the definitions in (23). The definition of is-matrix implies that a matrix has not exactly one dimension, but infinitely many! Therefore there is no need for signaling an error due to incompatibility of dimensions: for any two matrices A and B one shows

$$\exists m. \text{is-matrix } m m A \wedge \text{is-matrix } m m B . \tag{30}$$

The intuition behind (30) is that every matrix can be viewed as a square matrix of dimension m as long as m is large enough: one just needs to fill up the missing rows and columns with zeros.

The need for an *Error* matrix has vanished, but one can still use (17) to uniquely define a matrix. This time, we denote that matrix by 0:

$$\forall A. (A = 0) = (\text{is-matrix } 0 0 A) . \tag{31}$$

Another possibility of defining 0 is given by the following theorem:

$$\forall A. (A = 0) = (\forall m n. \text{is-matrix } m n A) . \tag{32}$$

We will see that 0 is actually the proper name for this matrix.

3.2 Lifting Unary Operators

In this subsection we look at how to define an unary operator U on matrices,

$$U :: \alpha \text{ matrix} \Rightarrow \beta \text{ matrix} , \quad (33)$$

by lifting an unary operator u on matrix elements,

$$u :: \alpha \Rightarrow \beta . \quad (34)$$

The first step is to lift u to infinite matrices:

$$\begin{aligned} \text{constdef} \\ \text{apply-infmatrix} :: (\alpha \Rightarrow \beta) \Rightarrow (\alpha \text{ infmatrix} \Rightarrow \beta \text{ infmatrix}) \\ \text{apply-infmatrix } u \equiv \lambda f j i. u (f j i) , \end{aligned} \quad (35)$$

which results in the lifting property

$$(\text{apply-infmatrix } u f) j i = u (f j i) . \quad (36)$$

Its proof is apparent from the definition of *apply-infmatrix*.

Now the unary lifting operator *apply-matrix* can be defined by first lifting u to infinite matrices, and then to finite matrices:

$$\begin{aligned} \text{constdef} \\ \text{apply-matrix} :: (\alpha \Rightarrow \beta) \Rightarrow (\alpha \text{ matrix} \Rightarrow \beta \text{ matrix}) \\ \text{apply-matrix } u \equiv \lambda A. \text{Abs-matrix} (\text{apply-infmatrix } u (\text{Rep-matrix } A)) . \end{aligned} \quad (37)$$

What is the lifting property for *apply-matrix*? A first guess yields

$$\text{Rep-matrix} (\text{apply-matrix } u A) j i = u (\text{Rep-matrix } A j i) . \quad (38)$$

But this is false (in the sense that we cannot prove it in HOL)! To see why, consider $\alpha = \beta = \text{int}$ and $u = \lambda x. 1$. Then we have

$$\text{apply-infmatrix } u (\text{Rep-matrix } A) = \begin{pmatrix} 1 & 1 & \dots \\ 1 & 1 & \dots \\ \vdots & \vdots & \ddots \end{pmatrix} \neq \text{Rep-matrix } B \quad (39)$$

for all matrices A and any matrix B . But there is a simple condition on u that turns out to be sufficient and necessary to prove (38):

$$u 0 = 0 \implies \text{Rep-matrix} (\text{apply-matrix } u A) j i = u (\text{Rep-matrix } A j i) . \quad (40)$$

This is easily provable using (37), (28), (29) and (36).

3.3 Lifting Binary Operators

Just as we have defined a unary lifting operator *apply-matrix*, we can define similarly a binary lifting operator *combine-matrix*:

$$\begin{aligned} \text{constdef} \\ \text{combine-infmatrix} :: \\ (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow (\alpha \text{ infmatrix} \Rightarrow \beta \text{ infmatrix} \Rightarrow \gamma \text{ infmatrix}) \\ \text{combine-infmatrix } v \equiv \lambda f g j i. v (f j i) (g j i) , \end{aligned} \quad (41)$$

constdef

$$\begin{aligned} \text{combine-matrix} &:: (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow (\alpha \text{ matrix} \Rightarrow \beta \text{ matrix} \Rightarrow \gamma \text{ matrix}) \\ \text{combine-matrix } v &\equiv \\ &\lambda A B. \text{Abs-matrix} (\text{combine-infmatrix } v (\text{Rep-matrix } A) (\text{Rep-matrix } B)) , \end{aligned} \quad (42)$$

The lifting property for *combine-matrix* reads

$$v \ 0 \ 0 = 0 \implies \text{Rep-matrix} (\text{combine-matrix } v \ A \ B) \ j \ i = v (\text{Rep-matrix } A \ j \ i) (\text{Rep-matrix } B \ j \ i) . \quad (43)$$

Lifting binary operators passes on commutativity and associativity. Defining

constdefs

$$\begin{aligned} \text{commutative} &:: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \text{bool} \\ \text{commutative } v &\equiv \forall x y. v \ x \ y = v \ y \ x \\ \text{associative} &:: (\alpha \Rightarrow \alpha \Rightarrow \alpha) \Rightarrow \text{bool} \\ \text{associative } v &\equiv \forall x y z. v \ (v \ x \ y) \ z = v \ x \ (v \ y \ z) \end{aligned} \quad (44)$$

we can formulate this propagation concisely:

$$\begin{aligned} \text{commutative } v \implies \text{commutative} (\text{combine-matrix } v) , \\ \llbracket v \ 0 \ 0 = 0; \text{associative } v \rrbracket \implies \text{associative} (\text{combine-matrix } v) . \end{aligned} \quad (45)$$

You might be surprised that the propagation of commutativity does not require $v \ 0 \ 0 = 0$, which is due to the idiosyncrasies of the definite description operator that is hidden in *Abs-matrix*.

3.4 Matrix Multiplication

We need one last lifting operation, the most interesting one: given two binary operators addition and multiplication on the matrix elements, define the matrix product induced by those two operators. As a basic tool we first define by primitive recursion a fold operator that acts on sequences:

$$\begin{aligned} \text{const } \text{foldseq} &:: (\alpha \Rightarrow \alpha \Rightarrow \alpha) \Rightarrow (\text{nat} \Rightarrow \alpha) \Rightarrow \text{nat} \Rightarrow \alpha \\ \text{primrec} & \\ \text{foldseq } f \ s \ 0 &= s \ 0 \\ \text{foldseq } f \ s \ (\text{Suc } n) &= f \ (s \ 0) \ (\text{foldseq } f \ (\lambda k. s \ (\text{Suc } k)) \ n) \end{aligned} \quad (46)$$

For illustration purposes, assume $s = (s_1, s_2, s_3, s_4, \dots, s_n, 0, 0, 0, \dots)$. Then

$$\begin{aligned} \text{foldseq } f \ s \ 0 &= s_1 , \\ \text{foldseq } f \ s \ 1 &= f \ s_1 \ s_2 , \\ \text{foldseq } f \ s \ 2 &= f \ s_1 \ (f \ s_2 \ s_3) , \\ \text{foldseq } f \ s \ 3 &= f \ s_1 \ (f \ s_2 \ (f \ s_3 \ s_4)) , \\ \text{foldseq } f \ s \ n &= f \ s_1 \ (f \ s_2 \ (\dots (f \ s_n \ 0) \dots)) , \\ \text{foldseq } f \ s \ (n+1) &= f \ s_1 \ (f \ s_2 \ (\dots (f \ s_n \ (f \ 0 \ 0)) \dots)) \text{ and so on.} \end{aligned} \quad (47)$$

Note that if $f \ 0 \ 0 = 0$ the above sequence converges:

$$f \ 0 \ 0 = 0 \implies \forall m. n \leq m \longrightarrow \text{foldseq } f \ s \ m = \text{foldseq } f \ s \ n . \quad (48)$$

Now we are prepared to deal with matrix multiplication:

$$\begin{aligned}
& \mathbf{constdef} \\
& \text{mult-matrix-}n :: \text{nat} \Rightarrow (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow (\gamma \Rightarrow \gamma \Rightarrow \gamma) \Rightarrow \\
& \quad \alpha \text{ matrix} \Rightarrow \beta \text{ matrix} \Rightarrow \gamma \text{ matrix} \\
& \text{mult-matrix-}n \ n \ \text{mult add } A \ B \equiv \text{Abs-matrix } (\lambda j \ i. \\
& \quad \text{foldseq add } (\lambda k. \text{mult } (\text{Rep-matrix } A \ j \ k) (\text{Rep-matrix } B \ k \ i)) \ n)
\end{aligned} \tag{49}$$

The idea of *mult-matrix- n mult add A B* is to consider only the first n columns of A and the first n rows of B when calculating the matrix product. Of course the matrix product should be independent of n . We achieve this by setting

$$\text{mult-matrix mult add} \equiv \lim_{n \rightarrow \infty} \text{mult-matrix-}n \ n \ \text{mult add} \ , \tag{50}$$

which is due to (48) well-defined if $\forall x. \text{mult } x \ 0 = \text{mult } 0 \ x = \text{add } 0 \ 0 = 0$ holds:

$$\begin{aligned}
& \mathbf{constdef} \\
& \text{mult-matrix} :: (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow (\gamma \Rightarrow \gamma \Rightarrow \gamma) \Rightarrow \\
& \quad \alpha \text{ matrix} \Rightarrow \beta \text{ matrix} \Rightarrow \gamma \text{ matrix} \\
& \text{mult-matrix mult add } A \ B \equiv \\
& \quad \text{mult-matrix-}n \ (\max(\text{ncols } A) \ (\text{nrows } B)) \ \text{mult add } A \ B \ .
\end{aligned} \tag{51}$$

Again, we have a lifting property:

$$\begin{aligned}
& \llbracket \forall x. \text{mult } x \ 0 = 0 \wedge \text{mult } 0 \ x = 0; \text{add } 0 \ 0 = 0 \rrbracket \implies \\
& \quad \text{Rep-matrix } (\text{mult-matrix mult add } A \ B) \ j \ i = \text{foldseq add} \\
& \quad (\lambda k. \text{mult } (\text{Rep-matrix } A \ j \ k) (\text{Rep-matrix } B \ k \ i)) \ (\max(\text{ncols } A) \ (\text{nrows } B)) \ .
\end{aligned} \tag{52}$$

Finally, let us examine what properties of element addition and element multiplication induce distributivity and associativity of *mult-matrix*.

Distributivity We distinguish between left and right distributivity:²

$$\begin{aligned}
& \mathbf{constdefs} \\
& \text{r-distributive} :: (\alpha \Rightarrow \beta \Rightarrow \beta) \Rightarrow (\beta \Rightarrow \beta \Rightarrow \beta) \Rightarrow \text{bool} \\
& \text{r-distributive mult add} \equiv \forall a \ u \ v. \text{mult } a \ (\text{add } u \ v) = \text{add } (\text{mult } a \ u) \ (\text{mult } a \ v) \\
& \text{l-distributive} :: (\alpha \Rightarrow \beta \Rightarrow \alpha) \Rightarrow (\alpha \Rightarrow \alpha \Rightarrow \alpha) \Rightarrow \text{bool} \\
& \text{l-distributive mult add} \equiv \forall a \ u \ v. \text{mult } (\text{add } u \ v) \ a = \text{add } (\text{mult } u \ a) \ (\text{mult } v \ a)
\end{aligned} \tag{53}$$

Distributivity of *mult* over *add* lifts to distributivity of *mult-matrix mult add* over *combine-matrix add* if *add* is associative and commutative and both *add* and *mult* behave as expected with respect to 0:

$$\begin{aligned}
& \llbracket \text{l-distributive mult add}; \text{associative add}; \text{commutative add}; \\
& \quad \forall x. \text{mult } x \ 0 = 0 \wedge \text{mult } 0 \ x = 0; \text{add } 0 \ 0 = 0 \rrbracket \\
& \implies \text{l-distributive } (\text{mult-matrix mult add}) \ (\text{combine-matrix add}) \ , \\
& \llbracket \text{r-distributive mult add}; \text{associative add}; \text{commutative add}; \\
& \quad \forall x. \text{mult } x \ 0 = 0 \wedge \text{mult } 0 \ x = 0; \text{add } 0 \ 0 = 0 \rrbracket \\
& \implies \text{r-distributive } (\text{mult-matrix mult add}) \ (\text{combine-matrix add}) \ .
\end{aligned} \tag{54}$$

² Our convention is that left distributivity means that the factor is distributed over the left sum, *not* that the left factor is the one that gets distributed.

Associativity We state the law of associativity for *mult-matrix* in a very general form:

$$\begin{aligned}
& \llbracket \forall a. \text{mult}_1 a 0 = 0; \forall a. \text{mult}_1 0 a = 0; \forall a. \text{mult}_2 a 0 = 0; \forall a. \text{mult}_2 0 a = 0; \\
& \quad \text{add}_1 0 0 = 0; \text{add}_2 0 0 = 0; \\
& \quad \forall a b c d. \text{add}_2 (\text{add}_1 a b) (\text{add}_1 c d) = \text{add}_1 (\text{add}_2 a c) (\text{add}_2 b d); \\
& \quad \forall a b c. \text{mult}_2 (\text{mult}_1 a b) c = \text{mult}_1 a (\text{mult}_2 b c); \\
& \quad \text{associative add}_1; \text{associative add}_2; \\
& \quad \text{l-distributive mult}_2 \text{ add}_1; \text{r-distributive mult}_1 \text{ add}_2 \rrbracket \\
\implies & \text{mult-matrix mult}_2 \text{ add}_2 (\text{mult-matrix mult}_1 \text{ add}_1 A B) C = \\
& \text{mult-matrix mult}_1 \text{ add}_1 A (\text{mult-matrix mult}_2 \text{ add}_2 B C) .
\end{aligned} \tag{55}$$

For $\text{mult} = \text{mult}_1 = \text{mult}_2$ and $\text{add} = \text{add}_1 = \text{add}_2$ this simplifies to

$$\begin{aligned}
& \llbracket \forall a. \text{mult} a 0 = 0; \forall a. \text{mult} 0 a = 0; \text{add} 0 0 = 0; \\
& \quad \text{associative add}; \text{commutative add}; \text{associative mult}; \\
& \quad \text{l-distributive mult add}; \text{r-distributive mult add} \rrbracket \\
\implies & \text{associative (mult-matrix mult add)} .
\end{aligned} \tag{56}$$

3.5 Lattice-Ordered Rings

Paulson describes in [6] how numerical theories like the theory of integers or the theory of reals can be organized in Isabelle/HOL using axiomatic type classes. For example both integers and reals form a ring, therefore Paulson recommends to prove theorems that are implied purely by ring properties only once, and then to prove that both types *int* and the type *real* are an instance of the axiomatic type class *ring*.

Birkhoff points out [12, chapt. 17] that for a fixed n the ring of all $n \times n$ square matrices forms a lattice-ordered ring in a natural way. The same is true for our finite matrices! Therefore it suggests itself to establish an axiomatic type class *lordered-ring* that captures the property of a type to form a lattice-ordered ring. Of course *lordered-ring* should be integrated with the other type classes like *ring* and *ordered-ring* of Isabelle/HOL to maximize theorem reuse. Two major changes along with minor modifications were necessary to the original hierarchy of type classes as described in [6]:

1. The original type class *ring* demanded both the existence of a multiplicative unit element and the commutativity of multiplication. But our finite matrices do not have such a multiplicative unit element, nor is multiplication of finite matrices a commutative operator. Nevertheless, finite matrices still form a ring in common mathematical terminology. Therefore the original type class *ring* was renamed to become *comm-ring-1* and new type classes *ring*, *ring-1* and *comm-ring* were introduced, suitable for rings that do not necessarily possess a 1 and/or are not commutative.
2. All ordered algebraic structures contained in the original hierarchy were linearly ordered. The natural (elementwise) order for finite matrices is a proper partial order, actually a lattice order. Therefore we enriched the hierarchy with type classes that model partially ordered algebraic systems like partially ordered groups and rings, or lattice-ordered groups and rings. For this we follow largely [13], [12].

We do not want to delve into the details of the modified hierarchy, but refer the curious reader to the 2005 release of Isabelle where all these changes have been incorporated. Instead, let us directly turn to lattice-ordered rings. A type α is an instance of the axiomatic type class *lordered-ring* iff

ring α is a ring with addition $+$, subtraction $-$, additive inverse $-$, multiplication $*$, zero 0 ,

lattice α is a lattice with partial order \leq and operators *join* and *meet*,

monotonicity addition and multiplication are monotone:

$$a \leq b \longrightarrow c + a \leq c + b \quad , \quad (57)$$

$$a \leq b \wedge 0 \leq c \longrightarrow a * c \leq b * c \wedge c * a \leq c * b \quad . \quad (58)$$

Both *int* and *real* are instances of *lordered-ring*:

instance *int* :: *lordered-ring*
instance *real* :: *lordered-ring* . (59)

Our goal is to prove

instance *matrix* :: (*lordered-ring*) *lordered-ring* . (60)

The above meta theorem has the following meaning (which is not legal Isabelle syntax):

(**instance** α :: *lordered-ring*) \implies (**instance** α *matrix* :: *lordered-ring*) . (61)

Of course, in order to prove (60), one first has to define 0 , $+$, $*$ etc. for objects of type *matrix*. The zero matrix is easy to define:

instance *matrix* :: (*zero*) *zero*
def (overloaded) (62)
 $0 \equiv \text{Abs-matrix}(\lambda j i. 0)$.

It is simple to show that this is actually the 0 we refer to in (31) and (32).

Addition $+$, multiplication $*$, subtraction $-$, unary minus $-$, can all be defined using the lifting machinery we have developed:

instance *matrix* :: (*plus*) *plus*
instance *matrix* :: (*minus*) *minus*
instance *matrix* :: ({*plus*, *times*}) *times*
defs (overloaded) (63)
 $A + B \equiv \text{combine-matrix}(\lambda a b. a + b) A B$
 $A - B \equiv \text{combine-matrix}(\lambda a b. a - b) A B$
 $-A \equiv \text{apply-matrix}(\lambda a. - a) A$
 $A * B \equiv \text{mult-matrix}(\lambda a b. a * b) (\lambda a b. a + b) A B$.

Finally, we need to be able to compare matrices:

instance *matrix* :: ({*ord*, *zero*}) *ord*
defs (overloaded) (64)
 $A \leq B \equiv \forall j i. \text{Rep-matrix } A j i \leq \text{Rep-matrix } B j i$

After having introduced the necessary syntax, we need to show that α matrix really constitutes a lattice-ordered ring, provided α constitutes one, in order to obtain (60). But almost the entire work has already been done: for example, in order to prove associativity of matrix multiplication,

$$\forall (A :: (\alpha :: \text{ordered-ring}) \text{ matrix}). A * (B * C) = (A * B) * C , \quad (65)$$

which is the hardest of all proof obligations, just apply (56)! The remaining proof obligations are not difficult to prove, either, one just has to make use of matrix extensionality (25) and the lifting properties (40), (43) and (52). It is useful, though, first to dispose of the assumptions in these lifting properties, so for example instead of using (43) directly one should prove and use

$$\begin{aligned} \text{Rep-matrix}(A + B) j i &= (\text{Rep-matrix } A j i) + (\text{Rep-matrix } B j i) \\ \text{Rep-matrix}(A - B) j i &= (\text{Rep-matrix } A j i) - (\text{Rep-matrix } B j i) . \end{aligned} \quad (66)$$

A proof obligation that differs from the others because it is not a universal property that needs to be shown, but an existential one, turns up when one has to show that *join* and *meet* do exist:

$$\begin{aligned} \exists j. \forall a b x. a \leq j a b \wedge b \leq j a b \wedge (a \leq x \wedge b \leq x \longrightarrow j a b \leq x) \\ \exists m. \forall a b x. m a b \leq a \wedge m a b \leq b \wedge (x \leq a \wedge x \leq b \longrightarrow x \leq m a b) \end{aligned} \quad (67)$$

But these are not difficult to exhibit! Just choose

$$\text{join} \equiv \text{combine-matrix join}, \text{ meet} \equiv \text{combine-matrix meet} . \quad (68)$$

3.6 Positive Part and Negative Part

In lattice-ordered rings (actually in groups, also), both the positive part and the negative part can be defined:

$$\begin{aligned} \text{constdefs} \\ \text{pprt} :: \alpha \Rightarrow (\alpha :: \text{ordered-ring}) \\ \text{pprt } x &\equiv \text{join } x 0 \\ \text{nppt} :: \alpha \Rightarrow (\alpha :: \text{ordered-ring}) \\ \text{nppt } x &\equiv \text{meet } x 0 \end{aligned} \quad (69)$$

We will write x^+ instead of $\text{pprt } x$, and x^- instead of $\text{nppt } x$. We have:

$$0 \leq x^+, \quad x^- \leq 0, \quad x = x^+ + x^-, \quad x \leq y \implies x^- \leq y^- \wedge x^+ \leq y^+ . \quad (70)$$

Positive part and negative part come in handy for calculating bounds for a product when bounds for each of the factors of the product are known:

$$\begin{aligned} \llbracket a_1 \leq a; a \leq a_2; b_1 \leq b; b \leq b_2 \rrbracket \\ \implies a * b \leq a_2^+ * b_2^+ + a_1^+ * b_2^- + a_2^- * b_1^+ + a_1^- * b_1^- \end{aligned} \quad (71)$$

In order to prove (71), decompose the factors into their parts and use distributivity. Then take advantage of the monotonicity of positive and negative part:

$$\begin{aligned} a * b &= (a^+ + a^-) * (b^+ + b^-) \\ &= a^+ * b^+ + a^+ * b^- + a^- * b^+ + a^- * b^- \\ &\leq a_2^+ * b_2^+ + a_1^+ * b_2^- + a_2^- * b_1^+ + a_1^- * b_1^- . \end{aligned}$$

4 The Main Theorem: Proving Bounds by Duality

Now we have everything in place to represent LPs by finite matrices. In sect. 2, we presented the basic idea of how to prove an arbitrarily precise upper bound for the objective function (1) of a given LP. There the LP was represented by matrices whose elements are real numbers:

$$c \in \mathbb{R}^{1 \times n}, \quad A \in \mathbb{R}^{m \times n}, \quad b \in \mathbb{R}^{m \times 1}, \quad l, u \in \mathbb{R}^{n \times 1}.$$

Dropping the dimensions we arrive at a representation of a real linear program by finite matrices:

$$c, A, b, l, u :: \text{real matrix} .$$

From now on we are always talking in terms of finite matrices.

We need a further modification of our representation of LPs: our method is based on numerical algorithms like the Simplex method, therefore we need to represent the data numerically. We allow for this possibility by looking at *intervals* of linear programs instead of only considering a single LP. Such an interval is given by finite matrices $c_1, c_2, A_1, A_2, b, l, u$. We can now state the main theorem as it has been proven in Isabelle/HOL:

$$\begin{aligned} & \llbracket A * x \leq b; A_1 \leq A; A \leq A_2; c_1 \leq c; c \leq c_2; l \leq x; x \leq u; 0 \leq y \rrbracket \\ \implies & c * x \leq y * b + (\text{let } s_1 = c_1 - y * A_2; s_2 = c_2 - y * A_1 \\ & \text{in } s_2^+ * u^+ + s_1^+ * u^- + s_2^- * l^+ + s_1^- * l^-) . \end{aligned} \quad (72)$$

The proof is by standard algebraic manipulations: using $A * x \leq b$ and $y \geq 0$,

$$c * x \leq y * b + (c - y * A) * x$$

follows at once. Then one just has to apply (71) to the product $(c - y * A) * x$. Note that this proof not only works for matrices, but for any lattice-ordered ring. Therefore the main theorem is valid also for lattice-ordered rings!

This is how our method works: First, we calculate the approximate optimal solution y of the dual LP. We know our primal LP only approximately, so we can pass only approximate data to the external LP solver. We could pass for example c_1, A_1, b, l, u . The LP solver will return the certificate y , which is only approximately non-negative. Therefore we replace all negative elements of y by 0. We then plug the known numerical data $y, c_1, c_2, A_1, A_2, b, l$ and u into (72) and simplify the resulting theorem. The simplification will rewrite $0 \leq y$ to *True* and the large expression on the right hand side of the inequality to a matrix numeral K with $ncols K \leq 1$ and $nrows K \leq 1$. The result of our method is therefore the theorem

$$\begin{aligned} & \llbracket \mathbf{A} * \mathbf{x} \leq b; A_1 \leq \mathbf{A}; \mathbf{A} \leq A_2; c_1 \leq \mathbf{c}; \mathbf{c} \leq c_2; l \leq \mathbf{x}; \mathbf{x} \leq u \rrbracket \\ \implies & \mathbf{c} * \mathbf{x} \leq K . \end{aligned} \quad (73)$$

In the above theorem, free variables are set in **bold face**. All other identifiers denote matrix numerals.

5 Sparse Matrices and Floats

After reading the previous section, you probably wonder what a matrix numeral might look like. We have chosen to represent matrix numerals in such a way that sparse matrices are encoded efficiently:

types

$$\begin{aligned} \alpha \text{ svec} &= (\text{nat} * \alpha) \text{ list} \\ \alpha \text{ smat} &= (\alpha \text{ svec}) \text{ svec} \end{aligned} \quad (74)$$

constdefs

$$\begin{aligned} \text{sparse-row-vector} &:: \alpha \text{ svec} \Rightarrow \alpha \text{ matrix} \\ \text{sparse-row-vector } l &\equiv \text{foldl}(\lambda m (i, e). m + (\text{singleton-matrix } 0 \ i \ e)) \ 0 \ l \\ \text{sparse-row-matrix} &:: \alpha \text{ smat} \Rightarrow \alpha \text{ matrix} \\ \text{sparse-row-matrix } L &\equiv \\ &\text{foldl}(\lambda m (j, l). m + (\text{move-matrix } (\text{sparse-row-vector } l) \ j \ 0)) \ 0 \ L \end{aligned} \quad (75)$$

Here *singleton-matrix* $j \ i \ e$ denotes the matrix whose elements are all zero except the element in row j and column i , which equals e . Furthermore *move-matrix* $A \ j \ i$ denotes the matrix that one gets if one moves the matrix A by j rows down and i columns right, and fills up the first j rows and i columns with zero elements.

Real numbers are represented as binary, arbitrary precision floating point numbers:

constdef

$$\begin{aligned} \text{float} &:: (\text{int} * \text{int}) \Rightarrow \text{real} \\ \text{float } (m, e) &\equiv (\text{real } m) * 2^e \end{aligned} \quad (76)$$

Finally, here is an example of a matrix numeral with floats as its elements:

$$[[(1, [(1, \text{float } (7, 0)), (3, \text{float } (-3, 2))]), (2, [(0, \text{float } (1, -3)), (1, \text{float } (-3, -4))])]]$$

$$\xrightarrow{\text{sparse-row-matrix}} \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 7 & 0 & -12 \\ \frac{1}{8} & -\frac{3}{16} & 0 & 0 \end{pmatrix}$$

We have formalized addition, subtraction, multiplication, comparison, positive part and negative part directly on sparse vectors and matrices by recursion on lists. The multiplication algorithm for sparse matrices is inspired by the one given in [8].

These operations on sparse vectors/matrices can be proven correct with respect to their finite matrices counterpart via the *sparse-row-matrix* morphism, assuming certain sortedness constraints. This is actually not too hard: all students of an introductory Isabelle/HOL class taught at Technische Universität München have been able to complete these proofs within four weeks as their final assignment with varying help from their tutors. Using these correctness results, one can then easily prove a sparse version of (72).

6 Conclusion

We have presented a novel way to prove arbitrarily precise bounds within higher-order logic for real linear programs that have a priori bounds. Our approach has three main virtues:

1. It is fast. The actual work is done by an external LP solver, the theorem prover has only to check a small certificate. Using a rewriting oracle that is based on the ideas found in [9], this check can be performed so quickly that it is projected that the linear programs arising in the proof of the Kepler conjecture can be bounded in about 10 days on a 3Ghz Pentium 4. The original computer programs from the 1998 proof of the Kepler conjecture that do not generate proofs at all needed back then about 7 days.
2. It decouples reasoning from computing issues. The new notion of finite matrices has been introduced, and it turned out that finite matrices and lattice-ordered rings are a natural choice to describe and reason about our method. At the same time, well-known data structures like sparse matrices can still be used for efficient computing.
3. It is independent from the actual method of solving LPs. This method could be Simplex, but does not have to be.

References

1. Thomas C. Hales. Some algorithms arising in the proof of the Kepler conjecture, sect. 3.1.1., arXiv:math.MG/0205209
2. Thomas C. Hales. A Proof of the Kepler Conjecture, *Annals of Mathematics*, to appear.
3. The Flyspeck Project Fact Sheet.
<http://www.math.pitt.edu/~thales/flyspeck/index.html>
4. Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, Springer 2002
5. Alexander Schrijver. *Theory of Linear and Integer Programming*, Wiley & Sons 1986
6. Lawrence C. Paulson. Organizing Numerical Theories Using Axiomatic Type Classes. *Journal of Automated Reasoning*, in press.
7. Lawrence C. Paulson. Defining Functions on Equivalence Classes. *ACM Transactions on Computational Logic*, in press.
8. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Transactions on Mathematical Software*, Vol 4. No 3, pp. 250-269, 1978.
9. Bruno Barras. Programming and Computing in HOL. *TPHOLs 2000*, LNCS 1869, pp. 17-37, Springer 2000.
10. Bart Jacobs, Tom Melham. Translating Dependent Type Theory into Higher Order Logic. *Typed Lambda Calculi and Applications: Proceedings of the International Conference*, LNCS 664, pp. 209-229, Springer 1993.
11. Serge Lang. *Algebra*, Addison-Wesley 1974.
12. Garrett Birkhoff. *Lattice Theory*, AMS 1967.
13. László Fuchs. *Partially ordered algebraic systems*, Addison-Wesley 1963.