Efficient Formal Verification of Bounds of Linear Programs

Alexey Solovyev and Thomas C. Hales*

Department of Mathematics, University of Pittsburgh,
Pittsburgh, PA 15260, USA

Abstract. One of the challenging problems in the formalization of mathematics is a formal verification of numerical computations. Many theorems rely on numerical results, the verification of which is necessary for producing complete formal proofs. The formal verification systems are not well suited for doing high-performance computing since even a small arithmetic step must be completely justified using elementary rules. We have developed a set of procedures in the HOL Light proof assistant for efficient verification of bounds of relatively large linear programs. The main motivation for the development of our tool was the work on the formal proof of the Kepler Conjecture. An important part of the proof consists of about 50000 linear programs each of which contains more than 1000 variables and constraints. Our tool is capable to verify one such a linear program in about 5 seconds. This is sufficiently fast for doing the needed formal computations.

1 Introduction

A trivial arithmetic step is not a problem in a traditional mathematical proof. The situation is different when one needs to obtain a formal proof where each step requires a complete verification using definitions, elementary logic operations, and axioms. Nevertheless, most proof assistant systems have special automated procedures that are able to prove simple arithmetic and logical statements. The main drawback of universal procedures is that they are not intended for high-performance computations. There exist theorems which require substantial computations. The formalization of these theorems can be quite challenging. One example is the four-color theorem, which was successfully formalized in Coq [1]. Another ambitious formalization project is the Flyspeck project [2]. The goal of this project is the formal proof of the Kepler conjecture [3]. This proof relies on extensive computer computations. An important part of the proof consists of more than 50000 linear programs (with about 1000 variables and constraints each). A bound of each linear program needs to be formally verified.

In our work, we present a tool for proving bounds of linear programs. The hard computational work is done using external software for solving linear programs. This software returns a special certificate which is used in the formal verification procedure. There are two main difficulties in this approach. One is the

* Research supported by NSF grant 0804189 and a grant from the Benter Foundation.

precision of the computer arithmetic. Usually, results of computer floating-point operations are not exact. Meanwhile, precise results are necessary for producing formal proofs. The second problem is the speed of the formal arithmetic. We successfully solved both problems and our tool is able to verify relatively large Flyspeck linear programs in about 5 seconds.

The Flyspeck project is carried out in the HOL Light proof assistant [5]. HOL Light is written in the Objective CAML programming language [10]. It has many convenient automated tools for proving arithmetic statements and can prove bounds of linear programs automatically. But even for small linear programs (10 variables and less) it can take a lot of time. Steven Obua in his thesis [8] developed a tool for verifying a part of Flyspeck linear programs. His work is done in the Isabelle proof assistant. We have made three significant advances over this earlier work. First, the combinatorial aspects of the linear programs have been simplified, as described in [4]. Second, we developed a tool for verifying bounds of general linear programs which verifies Flyspeck linear program much faster than Obua's program (in his work, the time of verification of a single linear program varies from 8.4 minutes up to 67 minutes). Third, our work is done in HOL Light so it is not required to translate results from one proof assistant into another. The code of our tool can be found in the Flyspeck repository [2] at trunk/formal_lp.

2 Verification of Bounds of Linear Programs

Our goal is to prove inequalities in the form $\mathbf{c}^T\mathbf{x} \le K$ such that $\mathbf{A}\mathbf{x} \le \mathbf{b}$ and $\mathbf{l} \le \mathbf{x} \le \mathbf{u}$, where $\mathbf{c}$, $\mathbf{b}$, $\mathbf{l}$, $\mathbf{u}$ are given n-dimensional vectors, $\mathbf{x}$ is an n-dimensional vector of variables, K is a constant, and $\mathbf{A}$ is an $m \times n$ matrix. To solve this problem, we consider the following linear program

$$\text{maximize } \mathbf{c}^T\mathbf{x} \text{ subject to } \bar{\mathbf{A}}\mathbf{x} \le \bar{\mathbf{b}},\ \bar{\mathbf{A}} = \begin{pmatrix} \mathbf{A} \\ -\mathbf{I}_n \\ \mathbf{I}_n \end{pmatrix},\ \bar{\mathbf{b}} = \begin{pmatrix} \mathbf{b} \\ -\mathbf{l} \\ \mathbf{u} \end{pmatrix}.$$

Suppose that $M = \max \mathbf{c}^T\mathbf{x}$ is the solution to this linear program. We require that $M \le K$. In fact, for our method we need a strict inequality $M < K$ because we employ numerical methods which do not give exact solutions.

We do not want to solve the linear program given above using formal methods. Instead, we use general software for solving linear programs which produces a special certificate that can be used to formally verify the original upper bound. Consider a dual linear program

$$\text{minimize } \mathbf{y}^T\bar{\mathbf{b}} \text{ subject to } \mathbf{y}^T\bar{\mathbf{A}} = \mathbf{c}^T,\ \mathbf{y} \ge 0.$$

The general theory of linear programming asserts that if the primal linear program has an optimal solution, then the dual program also has an optimal solution such that $\min \mathbf{y}^T\bar{\mathbf{b}} = \max \mathbf{c}^T\mathbf{x} = M$. Suppose that we can find an optimal solution to the dual program, i.e., assume that we know $\mathbf{y}$ such that $\mathbf{y}^T\bar{\mathbf{b}} = M \le K$

and $\mathbf{y}^T\bar{\mathbf{A}} = \mathbf{c}^T$. Then we can formally verify the original inequality by doing the following computations in a formal way:

$$\mathbf{c}^T\mathbf{x} = (\mathbf{y}^T\bar{\mathbf{A}})\mathbf{x} = \mathbf{y}^T(\bar{\mathbf{A}}\mathbf{x}) \le \mathbf{y}^T\bar{\mathbf{b}} = M \le K.$$

Our algorithm can be split into two parts. In the first part, we compute a solution $\mathbf{y}$ to the dual problem. In the second part, we formally prove the initial inequality using the computed dual solution and doing all arithmetic operations in a formal way.

2.1 Finding a Dual Solution

We impose additional constraints on the input data. We suppose that all coefficients and constants can be approximated by finite decimal numbers such that a solution of the approximated problem implies the original inequality. Consider a simple example. Suppose we need to prove the inequality $x - y \le \sqrt{3}$ subject to $0 \le x \le \pi$ and $\sqrt{2} \le y \le 2$. In general, an approximation that loosens the domain and tightens the range implies the original inequality. For example, consider an approximation of proving $x - y \le 1.732$, subject to $0 \le x \le 3.142$ and $1.414 \le y \le 2$. It is easy to see that if we can prove the approximated inequality, then the verification of the original inequality trivially follows. From now on, we assume that entries of $\bar{\mathbf{A}}$, $\bar{\mathbf{b}}$, $\mathbf{c}$, and the constant K are finite decimal numbers with at most p_1 decimal digits after the decimal point.

We need to find a vector $\mathbf{y}$ with the following properties:

$$\mathbf{y} \ge 0, \quad \mathbf{y}^T\bar{\mathbf{A}} = \mathbf{c}^T, \quad \mathbf{y}^T\bar{\mathbf{b}} \le K.$$

Moreover, we require that all elements of $\mathbf{y}$ are finite decimal numbers.

In our work, we use `GLPK` (GNU Linear Programming Kit) software for solving linear programs [7]. The input of this program is a model file which describes a linear program in the AMPL modeling language [9]. `GLPK` automatically finds solutions of the primal and dual linear programs. We are interested in the dual solution only. Suppose $\mathbf{r}$ is a numerical solution to the dual problem. Take its decimal approximation $\mathbf{y}_1^{(p)}$ with p decimal digits after the decimal point. We have the following properties of $\mathbf{y}_1^{(p)}$:

$$\mathbf{y}_1^{(p)} \ge 0, \quad M \le \bar{\mathbf{b}}^T\mathbf{y}_1^{(p)}, \quad \bar{\mathbf{A}}^T\mathbf{y}_1^{(p)} = \mathbf{c} + \mathbf{e}.$$

The vector $\mathbf{e}$ is the error term from numerical computation and decimal approximation.

We need to modify the numerical solution $\mathbf{y}_1^{(p)}$ to get $\mathbf{y}_2^{(p)}$ such that $\bar{\mathbf{A}}^T\mathbf{y}_2^{(p)} = \mathbf{c}$. Write $\mathbf{y}_1^{(p)} = (\mathbf{z}^T, \mathbf{v}^T, \mathbf{w}^T)^T$ where $\mathbf{z}$ is an m-dimensional vector, $\mathbf{v}$ and $\mathbf{w}$ are n-dimensional vectors. Define $\mathbf{y}_2^{(p)}$ as follows

$$\mathbf{y}_2^{(p)} = \begin{pmatrix} \mathbf{z} \\ \mathbf{v} + \mathbf{v}_e \\ \mathbf{w} + \mathbf{w}_e \end{pmatrix}, \quad \mathbf{v}_e = \frac{|\mathbf{e}| + \mathbf{e}}{2}, \quad \mathbf{w}_e = \frac{|\mathbf{e}| - \mathbf{e}}{2}.$$

In other words, if $e_i > 0$ (the i-th component of $\mathbf{e}$), then we add e_i to v_i, otherwise we add $-e_i$ to w_i. We obtain $\mathbf{y}_2^{(p)} \geq 0$. Moreover,

$$\bar{\mathbf{A}}^T \mathbf{y}_2^{(p)} = \mathbf{A}^T \mathbf{z} - (\mathbf{v} + \mathbf{v}_e) + (\mathbf{w} + \mathbf{w}_e) = \bar{\mathbf{A}}^T \mathbf{y}_1^{(p)} - \mathbf{e} = \mathbf{c}.$$

Note that elements of $\mathbf{y}_2^{(p)}$ are finite decimal numbers. Indeed, $\mathbf{y}_2^{(p)}$ is obtained by adding some components of the error vector $\mathbf{e} = \bar{\mathbf{A}}^T \mathbf{y}_1^{(p)} - \mathbf{c}$ to the vector $\mathbf{y}_1^{(p)}$, and all components of $\bar{\mathbf{A}}$, $\mathbf{c}$, and $\mathbf{y}_1^{(p)}$ are finite decimal numbers.

If $\bar{\mathbf{b}}^T \mathbf{y}_2^{(p)} \leq K$, then we are done. Otherwise, we need to find $\mathbf{y}_1^{(p+1)}$ using higher precision decimal approximation of $\mathbf{r}$ and consider $\mathbf{y}_2^{(p+1)}$. Assuming that the numerical solution $\mathbf{r}$ can be computed with arbitrary precision and that $M < K$, we eventually get $\bar{\mathbf{b}}^T \mathbf{y}_2^{(s)} \leq K$.

From the computational point of view, we are interested in finding an approximation of the dual solution such that its components have as few decimal digits as possible (formal arithmetic on small numbers works faster). We start from a small value of p_0 (we choose $p_0 = 3$ for Flyspeck linear programs) and construct $\mathbf{y}_2^{(p_0)}, \mathbf{y}_2^{(p_0+1)}, \ldots$ until we get $\bar{\mathbf{b}}^T \mathbf{y}_2^{(p_0+i)} \leq K$.

We implemented a program in C# which takes a model file with all inequalities and a dual solution obtained with `GLPK`. The program returns an approximate dual solution (with as low precision as possible) which then can be used in the formal verification step. The current implementation of our program does not work with arbitrary precision arithmetic, so it could fail on some linear programs. It should be not a problem for many practical cases because standard double precision floating-point arithmetic can exactly represent 15-digit decimal numbers (for instance, we need at most 6 decimal digits for proving Flyspeck linear programs).

2.2 Formal Verification

Our aim is to verify the inequality $\mathbf{c}^T \mathbf{x} \leq K$ using the computed dual solution approximation $\mathbf{y}^T = (\mathbf{z}^T, \mathbf{v}^T, \mathbf{w}^T)$ (we write $\mathbf{y}$ for the approximation $\mathbf{y}_2^{(s)}$, computation of which is described in the previous section):

$$\mathbf{c}^T \mathbf{x} = \mathbf{z}^T (\mathbf{A}\mathbf{x}) - \mathbf{v}^T \mathbf{x} + \mathbf{w}^T \mathbf{x} = \mathbf{y}^T \bar{\mathbf{A}} \mathbf{x} \leq \mathbf{y}^T \bar{\mathbf{b}} \leq K.$$

Here $\mathbf{x}$ is an n-dimensional vector of variables, $\mathbf{x} = (x_1, \ldots, x_n)$. We need to verify two results using formal arithmetic: $\mathbf{y}^T \bar{\mathbf{A}} \mathbf{x} = \mathbf{c}^T \mathbf{x}$ and $\mathbf{y}^T \bar{\mathbf{b}} \leq K$.

The computation of $\mathbf{y}^T \bar{\mathbf{b}}$ is a straightforward application of formal arithmetic operations. $\mathbf{y}^T \bar{\mathbf{A}} \mathbf{x}$ can be computed in a quite efficient way. Usually, the matrix $\bar{\mathbf{A}}$ is sparse, so it makes no sense to do a complete matrix multiplication in order to compute $\mathbf{y}^T \bar{\mathbf{A}} \mathbf{x}$. The i-th constraint inequality can be written in the form

$$\sum_{j \in I_i} a_{ij} x_j \leq b_i,$$

where I_i is the set of indices such that $a_{ij} \neq 0$ for $j \in I_i$, and $a_{ij} = 0$ for $j \notin I_i$. Also we have $2n$ inequalities for bounds of $\mathbf{x}$: $l_i \leq x_i \leq u_i$.

Define a special function in HOL Light for representing the left hand side of a constraint inequality

$$\vdash \mathrm{linf}[] = \&0 \land \mathrm{linf}[(a_1, x_1); t] = a_1 * x_1 + \mathrm{linf}[t]$$

For the sake of presentation, we write HOL Light expressions in a simplified notation. Here, $\vdash$ means that the definition is a HOL Light theorem; &0 denotes the real number zero. Our function linf has the following type

$$\mathrm{linf} : (real, real)list \to real.$$

It means that linf takes a list of pairs of real-valued elements and returns a real value. The function linf is defined recursively: we define its value on the empty list [], and we specify how linf can be computed on a k-element list using its value on a $(k-1)$-element list.

We suppose that all constraints and bounds of variables are theorems in HOL Light and each such theorem has the form

$$\vdash \alpha_1 x_1 + \ldots + \alpha_k x_k \leq \beta$$

We have a conversion which transforms $\alpha_1 x_1 + \ldots + \alpha_k x_k$ into the corresponding function $\mathrm{linf}[(\alpha_1, x_1); \ldots; (\alpha_k, x_k)]$. Also, variables x_i can have different names (like $var1$, $x4$, $y34$, etc.), and after the conversion into linf, all elements in the list will be sorted using some fixed ordering on the names of variables (usually, it is a lexicographic ordering). For efficiency, it is important to assume that the variables in the objective function $\mathbf{c}^T\mathbf{x}$ (i.e., variables for which $c_i \neq 0$) are the last one in the fixed ordering (we can always satisfy this assumption by renaming the variables).

First of all, we need to multiply each inequality by the corresponding value $0 \leq y_i$. It is a straightforward computation based on the following easy theorem

$$\vdash c * \mathrm{linf}[(a_1, x_1); \ldots; (a_k, x_k)] = \mathrm{linf}[(c * a_1, x_1); \ldots (c * a_k, x_k)]$$

Note that we can completely ignore inequalities for which $y_i = 0$ because they do not contribute to the sum which we want to compute.

The main step is computation of the sum of two linear functions. Suppose we have two linear functions $\mathrm{linf}[(a, x_i); t_1]$ and $\mathrm{linf}[(b, x_j); t_2]$ (t_1 and t_2 denote tails of the lists of pairs). Depending on the relation between x_i and x_j (i.e., we compare the names of variables), we need to consider three cases: $x_i \equiv x_j$ (the same variables), $x_i \prec x_j$ (in the fixed ordering), or $x_i \succ x_j$. In the first case, we apply the following theorem

$$\vdash \mathrm{linf}[(a, x); t_1] + \mathrm{linf}[(b, x); t_2] = (a + b) * x + (\mathrm{linf}[t_1] + \mathrm{linf}[t_2])$$

In the second case, we have the theorem

$$\vdash \mathrm{linf}[(a, x_i); t_1] + \mathrm{linf}[(b, x_j); t_2] = a * x_i + (\mathrm{linf}[t_1] + \mathrm{linf}[(b, x_j); t_2])$$

In the third case, the result is analogous to the second case. After applying one of these theorems, we recursively compute the expression in the parentheses and $a+b$ (if necessary). Then we can apply the following simple result and finish the computation of the sum

$$\vdash a * x + \mathrm{linf}[t] = \mathrm{linf}[(a, x); t]$$

Moreover, if the variables in both summands are ordered, then the variables in the result will be ordered.

After adding all inequalities for constraints, we get the inequality $\mathbf{z}^T\mathbf{A}\mathbf{x} \leq \mathbf{z}^T\mathbf{b}$ where the left hand side is computed in terms of linf. Now we need to find the sum of this inequality and inequalities for boundaries (multiplied by the corresponding coefficients). We do not transform boundary inequalities into the linf representation. Each boundary inequality has one of two forms

$$\vdash -x_i \leq -l_i \text{ or } \vdash x_i \leq u_i$$

Again, we need to multiply these inequality by the corresponding element of $\mathbf{y} = (\mathbf{z}^T, \mathbf{v}^T, \mathbf{w}^T)^T$ and get

$$\vdash -v_i * x_i \leq v_i * -l_i \text{ or } \vdash w_i * x_i \leq w_i * u_i$$

If $v_i = 0$ or $w_i = 0$, then we can ignore the corresponding inequality. If for some x_i we have both boundary inequalities (lower and upper bounds) with non-zero coefficients, then find the sum of two such inequalities. After that, we get one inequality of the form $r_i x_i \leq d_i$ for each variable x_i.

Before finding the sum of the inequality $\mathbf{z}^T\mathbf{A}\mathbf{x} \leq \mathbf{z}^T\mathbf{b}$ and the boundary inequalities, we sort boundary inequalities using the same ordering we used for sorting variables in linf. We assumed that the variables in the objective function $\mathbf{c}^T\mathbf{x}$ are the last ones in our ordering. Let $c_i = 0$ for all $i \leq n_0$. Suppose that we want to find the sum of $r_1 x_1 \leq d_1$ and $\mathrm{linf}[(a_1, x_1); t] \leq s$. We know that $c_1 = 0$, so the first term $a_1 x_1$ in the linear function and $r_1 x_1$ must cancel each other, hence we have $a_1 = -r_1$. The sum can be found using the following result

$$\vdash a * x + \mathrm{linf}[(-a, x); t] = \mathrm{linf}[t]$$

So we can efficiently compute the sum of all boundary inequalities for $i = 1, \ldots, n_0$. For the last $n - n_0$ variables, we have non-vanishing terms and the sum can be found in the standard way. Practically, the number $n - n_0$ is small compared to n, so most of computations are done in the efficient way.

At last, we get the inequality

$$\vdash \mathrm{linf}[(c_{n_0+1}, x_{n_0+1}); \ldots; (c_n, x_n)] \leq M',$$

where $M' = \mathbf{y}^T\mathbf{b} \leq K$. It is left to prove that $M' \leq K$. This can be done using standard HOL Light procedures because we need to perform this operation only once for each linear program.

3 Optimization

We implemented our algorithm using HOL Light elementary inference rules as much as possible. We avoided powerful but time consuming operations (such as tactics, rewrites, and derived rules) everywhere. Formal arithmetic operations play a very important role in our algorithm. The optimization of these operations is described below.

3.1 Integer Arithmetic

Formal arithmetic operations on integers are considerably faster than operations on rational (decimal) numbers. We want to perform all formal computations using integer numbers only. We have the following numerical values: entries of $\bar{\mathbf{A}}$, $\bar{\mathbf{b}}$, $\mathbf{c}$, $\mathbf{y}$, and the constant K. The input data $\bar{\mathbf{A}}$, $\bar{\mathbf{b}}$, $\mathbf{c}$, and K can be approximated by finite decimal numbers with at most p_1 decimal digits after the decimal point. The dual solution $\mathbf{y}$ is constructed in such a way that all its elements have at most p_2 decimal digits after the decimal point.

We modify the main step of the algorithm as follows. Compute

$$(10^{p_1+p_2}\mathbf{c}^T)\mathbf{x} = (10^{p_2}\mathbf{y}^T)\left(10^{p_1}\bar{\mathbf{A}}\right)\mathbf{x} \leq (10^{p_2}\mathbf{y}^T)(10^{p_1}\bar{\mathbf{b}}) \leq 10^{p_1+p_2}K.$$

It is clear, that the computations above can be done using integer numbers only. In the last step, we divide both sides by $10^{p_1+p_2}$ (using formal rational arithmetic only one time) and obtain the main inequality.

3.2 Faster Low Precision Arithmetic in HOL Light

We have obtained significant improvements in performance over the original implementations of basic arithmetic operations on natural numbers (operations on integers and rational numbers are derived from operations on natural numbers) by changing the internal representation of numerals. We are mostly interested in operations on relatively small numbers (5-20 decimal digits), so we do not consider improved arithmetic algorithms which usually work well on quite large numbers. Instead, we are using tables of pre-proved theorems which allow to get the computation results quickly for not very large numbers.

Numerals in HOL Light are constructed using three constants BIT0, BIT1, and 0 which are defined by

$$\mathrm{BIT0}(n) = n + n, \quad \mathrm{BIT1}(n) = n + n + 1.$$

Here, n is any natural number (it is expected that n is already in the correct form). With these constants, any numeral is represented by its binary expansion with the least significant bit first:

$$1 = \mathrm{BIT1}(0), \quad 2 = 1 + 1 = \mathrm{BIT0}(1) = \mathrm{BIT0}(\mathrm{BIT1}(0)),\ \text{etc.}$$

Define new constants for constructing natural numbers. Instead of base 2, represent a number using an arbitrary base $b \geq 2$. When the base is fixed, we define constants

$$D_i(n) = bn + i.$$

In HOL Light, we write $D0$, $D1$, $D2$, etc. For example, if $b = 10$, then we can write $123 = D3(D2(D1(0)))$.

We implemented arithmetic operations on numbers represented by our constants in a straightforward way. As an example, consider how the addition operation is implemented for our numerals. First of all, we prove several theorems which show how to add two digits. If $i + j = k < b$, then we have

$$\vdash D_i(m) + D_j(n) = D_k(m + n)$$

If $i + j = k \geq b$, then

$$\vdash D_i(m) + D_j(n) = D_{k-b}(\mathrm{SUC}(m + n))$$

Here $\mathrm{SUC}(n) = n + 1$ is another arithmetic operation which is implemented for our numerals. Also we have two terminal cases $\vdash n + 0 = n$ and $\vdash 0 + n = n$.

We store all these theorems in a hash table. The names of the constants are used as key values: the theorem with the left hand side $D_1(m) + D_2(n)$ has the key value "D1D2". With all these theorems, the addition of two numbers is easy. Consider an example. Suppose $b = 10$ and we want to compute $14 + 7 = D4(D1(0)) + D7(0)$. First, we look at the least significant digits $D4$ and $D7$ and find the corresponding theorem $D4(m) + D7(n) = D1(\mathrm{SUC}(m + n))$. In our case, instantiate the variables by $m = D1(0)$ and $n = 0$. We obtain $D4(D1(0)) + D7(0) = D1(\mathrm{SUC}(D1(0) + 0))$. Recursively compute $D1(0) + 0 = D1(0)$ (it is the terminal case). Then we apply the procedure for computing $\mathrm{SUC}(n)$ and obtain $\mathrm{SUC}(D1(0)) = D2(0)$. Hence, the final result is obtained: $D4(D1(0)) + D7(0) = D1(D2(0))$.

The multiplication of two numbers is more complicated but also straightforward. The original procedure for multiplying two natural numbers in HOL Light is based on the Karatsuba algorithm [6]. This algorithm asymptotically faster than a naive approach but we found that in our case (numbers with 10-30 digits) the Karatsuba algorithm does not give a significant advantage and can even slow computations down on small numbers.

The subtraction and division are implemented in the same way as in HOL Light. Initially, the result of an operation is obtained using “informal” computer arithmetic, and then simple theorems along with formal addition and multiplication operations are used to prove that the computed result is indeed the right one.

4 Performance Tests

The results of performance tests for the improved multiplication and addition operations are given in Tables 1 and 2. These results are obtained by performing formal operations on 1000 pairs of randomly generated integer numbers.

Table 1. Performance results for 1000 multiplication operations

Size of operands	Native HOL Light mult.	Base 16 mult.	Base 256 mult.
5 decimal digits	2.220 s	0.428 s	0.148 s
10 decimal digits	7.216 s	1.292 s	0.376 s
15 decimal digits	16.081 s	3.880 s	1.316 s
20 decimal digits	59.160 s	6.092 s	2.256 s
25 decimal digits	85.081 s	10.645 s	3.592 s

Table 2. Performance results for 1000 addition operations

Size of operands	Native HOL Light add.	Base 16 add.	Base 256 add.
5 decimal digits	0.188 s	0.064 s	0.052 s
10 decimal digits	0.324 s	0.100 s	0.064 s
15 decimal digits	0.492 s	0.112 s	0.096 s
20 decimal digits	0.648 s	0.176 s	0.108 s
25 decimal digits	0.808 s	0.208 s	0.132 s

Table 3 contains the performance results for our verification procedure of bounds of several Flyspeck linear programs. For each linear program, two test results are given. In one test, the native HOL Light formal arithmetic is used; in another test, the improved arithmetic with the fixed base 256 is utilized.

Table 3. Performance results for verification of linear program bounds

Linear program ID	# variables	# constraints	Native arith.	Base 256 arith.
18288526809	743	519	4.048 s	2.772 s
168941837467	750	591	5.096 s	3.196 s
25168582633	784	700	8.392 s	4.308 s
72274026085	824	773	7.656 s	5.120 s
28820130324	875	848	9.292 s	5.680 s
202732667936	912	875	9.045 s	5.816 s
156588677070	920	804	8.113 s	5.252 s
123040027899	1074	1002	11.549 s	6.664 s
110999880825	1114	1000	10.085 s	6.780 s

Note that most of the Flyspeck linear programs can be solved in less than 5 seconds. The linear programs in the table are selected to demonstrate a wide range of results.

References

1. Gonthier, G.: Formal Proof—The Four-Color Theorem. Notices of the AMS 55(11), 1382–1393 (2008)
2. Hales, T.C.: The Flyspeck Project. `http://code.google.com/p/flyspeck`
3. Hales, T.C.: A proof of the Kepler conjecture. Annals of Mathematics. Second Series 162(3), 1065–1185 (2005)
4. Hales, T.C.: Linear Programs for the Kepler Conjecture (Extended Abstract). Lecture Notes in Computer Science 6327, 149–151 (2010)
5. Harrison, J.: The HOL Light theorem prover. `http://www.cl.cam.ac.uk/~jrh13/hol-light/`
6. Karatsuba, A.A.: The Complexity of Computations. Proceedings of the Steklov Institute of Mathematics 211, 169–183 (1995)
7. Makhorin, A.O.: GNU Linear Programming Kit. `http://www.gnu.org/software/glpk/`
8. Obua, S.: Flyspeck II: The Basic Linear Programs (2008) Available at `http://code.google.com/p/flyspeck`
9. A Modeling Language for Mathematical Programming. `http://www.ampl.com/`
10. The Caml Language. `http://caml.inria.fr/`