Experiments on the feasibility of using a floating-point simplex in an SMT solver

Diego C B de Oliveira and David Monniaux

CNRS / Verimag
Grenoble, France
diego.caminha@imag.fr, david.monniaux@imag.fr

Abstract

SMT solvers use simplex-based decision procedures to solve decision problems whose formulas are quantifier-free and atoms are linear constraints over the rationals. State-of-art SMT solvers use rational (exact) simplex implementations, which have shown good performance for typical software, hardware or protocol verification problems over the years. Yet, most other scientific and technical fields use (inexact) floating-point computations, which are deemed far more efficient than exact ones. It is therefore tempting to use a floating-point simplex implementation inside an SMT solver, though special precautions must be taken to avoid unsoundness.

In this work, we describe experimental results, over common benchmarks (SMT-LIB) of the integration of a mature floating-point implementation of the simplex algorithm (GLPK) into an existing SMT solver (OpenSMT). We investigate whether commonly cited reasons for and against the use of floating-point truly apply to real cases from verification problems.

1 Introduction

Arithmetic is widely present in verification problems. The most common method for solving satisfiability problems modulo the theory of linear real (or, equivalently, rational) arithmetic is a combination of a DPLL SAT-solver and a decision procedure for conjunctions of linear (in)equalities, obtained by running phase I of the simplex algorithm [5].[1] Furthermore, problems over the theory of linear *integer* arithmetic are most often reduced to problems over the reals by judicious use of Gomory cuts, branch-and-bound, and other techniques from integer linear programming; thus the performance of the decision procedure for linear real arithmetic also conditions that of linear integer arithmetic.

The simplex implementation inside an SMT solver based on the DPLL($\mathcal{T}$) approach [8, 5] has the following tasks:

- Given a conjunction C of linear (in)equalities, say whether it is satisfiable or not and, if it is so, provide a solution.
- Optionally, given C, say whether it trivially implies certain other predicates (*theory propagation*).

Furthermore, the algorithm should be organized so that it is easy to add and remove constraints in C.

Most efficient implementations of the simplex algorithm operate over floating-point numbers. However, because of floating-point roundoff errors, such implementations may produce incorrect results in some cases; this will not do inside a verification tool such as an SMT solver.

[1] In operation research settings, this satisfiability phase is followed by another for optimization. This second phase is not commonly used inside SMT solvers.

SMT solvers thus use implementations of the simplex algorithm over the rational numbers; although such an implementation may be slower than one over floating-point numbers, it provides assurance that the results it obtains are sound and not subject to rounding errors.

Despite this weakness of floating-point implementations of the simplex algorithm, there have been several proposals to use them in SMT solvers to improve performance, of course with appropriate workarounds to ensure the soundness of results [7, 11, 2]. There exist indeed several ways to use a floating-point simplex implementation (whether one uses a primal or dual simplex, how to "correct" possibly unsound results...), and it was not so clear from experimental results whether one is better than another or even whether it is actually interesting to use a floating-point simplex.

As an example of a difficulty for evaluating simplex implementations, the main weakness of implementations using rational numbers is that the size of numerators and denominators may grow considerably for certain problems. While this depends of the problems being solved as well as the implementation of the simplex algorithm that may use techniques to reduce such tendency, such cases seldom seem to occur on some solvers handling real examples arising from software or hardware verification problems, as opposed to, say, random instances [11]. Furthermore, it is difficult to evaluate such systems outside of a full SMT solver, for the performance of a SMT solver not only depends on that of the theory solver (here, the simplex algorithm) but also on the size of the clauses output by the theory solver (smaller clauses are more efficient from the point of view of the SAT solver) and on other factors whose impact on overall performance is unclear.

In this article, we report on experiments of integration of a floating-point simplex solver (GLPK [10]) inside OpenSMT[2] [3].

2 Comparing the exact and floating-point simplex implementations

We shall distinguish two implementations of the simplex algorithm: the "floating-point simplex" and the "exact simplex", the former being implemented with floating-point numbers and the second with an exact rational representation, with arbitrary-precision numbers.

We investigate a combination of both implementations in order to provide exact results faster than a pure exact implementation. Our first step is to know how much faster the floating-point simplex can be compared to the exact simplex, and how often it is wrong.

2.1 Inside an SMT solver

The original implementation of OpenSMT calls an exact simplex as the theory solver for linear real arithmetic (LRA); it follows the same basic setup as Yices [5] or Z3. We modified OpenSMT so as to run both simplex implementations at the same time: its preexisting exact simplex and the floating-point simplex GLPK. As the floating-point simplex cannot handle strict inequalities directly, a strict inequality $\sum_i a_i x_i < b$ (the a_i and b are constants) is interpreted as $\sum_i a_i x_i \leq b - 10^{-9}$.

At this point, we are only interested in comparing how long both implementations take to verify whether a set of linear arithmetic constraints is satisfiable or not. All the extra work that

[2]The version used was the latest public available with the source code, OpenSMT 1.0.1, dated from October 2010.

is usually done by decision procedures (bookkeeping, communication with the SAT solver) is done by the original exact simplex and is discounted from this comparison.

The incremental nature of the decision procedure is preserved. The GLPK solver object is carried along the exact simplex state, and is not reinitialized as constraints are tightened or loosened. The GLPK solver tableau is created with all the linear forms $\sum_i a_i x_i$ present in the SMT formula; when constraints are tightened (i.e. $\sum_i a_i x_i \leq +\infty$, also known as "contraint not asserted", gets changed to $\sum_i a_i x_i \leq C$ where C is a finite constant, or when $\sum_i a_i x_i \leq C$ gets changed to $\sum_i a_i x_i \leq C'$ where $C' < C$) or loosened (the converse of the above), only the GLPK bounds (but not the tableau) are changed (of course, subsequent checks may induce pivoting operations and thus tableau updates).

2.2 The first experiment

The benchmarks used in this experiment are those from the division QF_LRA (quantifier-free linear real arithmetic) in the SMT-LIB [1]. These benchmarks are used by the SMT community to check their solvers and compare their respective speeds. They come from a variety of sources: there are random, crafted or industrial examples.

The distribution of the times taken to run the 634 benchmarks is shown in Figure 1. The time limit was set to 2 minutes. With a total of over 38 millions distinct set of arithmetic constraints tested, the total accumulated time of the exact simplex was 5 h 26 m 55 s, while the accumulated time of the floating-point simplex was 4 h 44 m 45 s. The times include benchmarks that timed out.

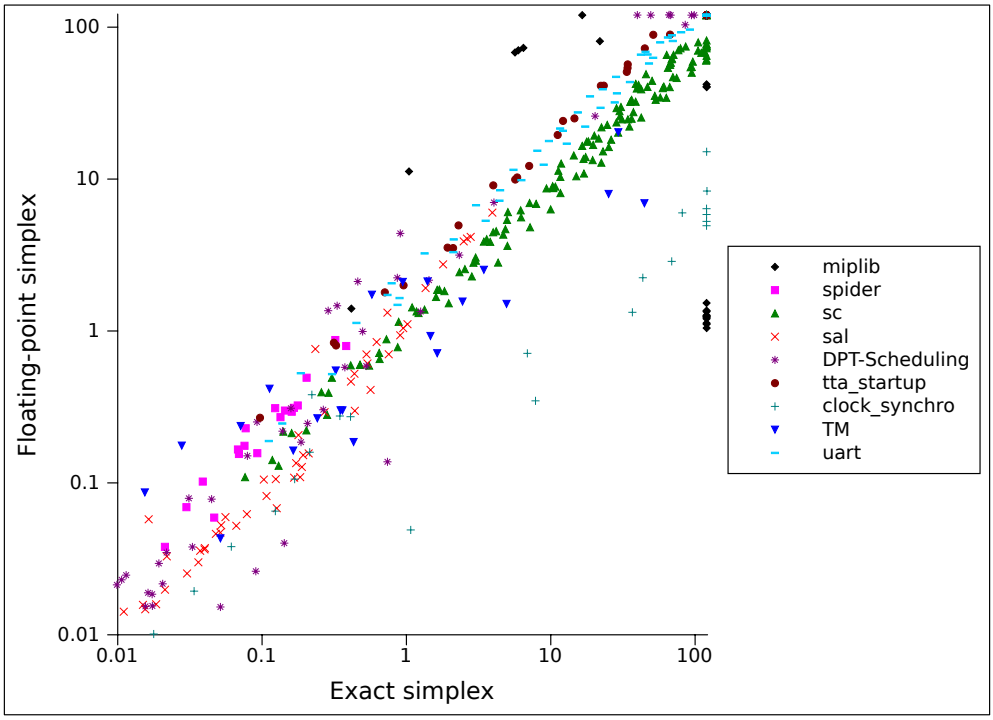

Figure 1: Exact simplex vs floating-point simplex in QF_LRA.

In the overall time comparison, the floating-point simplex is 14% faster than the exact simplex.

These results may even be discouraging: with the extra overhead of ensuring that the results from the floating-point simplex are sound, the final system is likely not to be faster than one using only exact computations. Yet, looking at the distribution, we can see that the floating-point simplex was much faster in a small number of cases, at least 10 times faster in approximately 3% of the benchmarks.

Apart from the timing distribution, the following points are important for understanding the feasibility of a floating-point simplex inside an SMT solver when solving practical problems:

Soundness. 11 (1.7%) of the 634 benchmarks tested had at least one incorrect check (meaning that the floating-point and exact simplex implementations yield different results). If we consider the total number of sets of arithmetic constraints checked, there were 852 (0.002%) incorrect results out of 38,566,969.

As expected, the floating-point simplex may give incorrect results. Even though they happen rarely, this should be taken in consideration when incorporating a floating-point simplex into an SMT solver; such situations must be detected and worked around. However, they are not statistically significant with respect to overall efficiency.

Overflow. OpenSMT and other SMT solvers like Z3 [4], implement arbitrary precision rational arithmetic in two layers: it first attempts computing the numerator and denominator inside machine words, and only if impossible, because of overflow, allocates extended precision numbers and branches into an extended precision library (OpenSMT[3] and early versions of Z3 use the GMP library [6][4]). GMP features very efficient procedures for computing on large integers, with both advanced high-level algorithms and highly optimized assembly code for basic operations. Yet, calling GMP for operations over small numbers incurs significant overhead — because of the cost of function calls to the library, and because the memory where GMP numbers are stored is allocated using `malloc()` and `free()`, thus incurring the cost of memory allocation and deallocation and breaking memory locality with respect to the processor cache.

It was recognized that in most cases from verification instances (as opposed to, say, random instances [11]), the solver never has to handle overflows and never calls the extended precision library. It is therefore common practice for tools such as decision procedures or static analyzers to adopt this layered approach, which, in the case of OpenSMT is implemented by overloaded C++ operators.

We shall now discuss our findings in this respect. 23 (3.6%) of the 643 benchmarks had at least one overflow. However, this proportion decreases even more when considering all the numbers manipulated (15,123,297,625), only 1,100,988 (0.007%) of them had overflow. All the benchmarks that had overflow are from the *clock_synchro* family.

One of the reasons that the floating-point simplex can be much faster than the exact simplex is when the latter starts operating on extended precision numbers due to overflow. Figure 1 shows that the problems from the family *clock_synchro* which had overflows are constantly solved faster by the floating-point simplex. Yet, as we can see that in the QF_LRA benchmarks, overflow is very rare and in this case, efficient rational number libraries can perform as well as floating-point computation.

[3] D. Monniaux implemented the C++ FastRationals layered arithmetic in OpenSMT. A similar library, ZArith, is available for OCaml from X. Leroy and A. Miné.

[4] L. de Moura, personal communication.

The problems of QF_LRA are sparse in general. Additionally, numbers presented in the formulas are usually small. Overflows happen much easier in dense problems when the frequent linear combination of the expression makes the coefficient of the variables to grow very fast [11].

Size and running times. It is also worth noticing the average running time per query of the simplex implementations: 0.44 ms for the floating-point simplex and 0.51 ms for the exact simplex. Both times are very low. That means that a decision procedure for linear arithmetic should be optimized to run several small/medium problems incrementally rather than a few large ones; this may explain earlier disappointing results when calling an industry-strength external linear programming package [7], which is optimized for solving very large instances from operation research.

The distribution of the running times seems randomly spread. Even though both implementations are variants on the simplex algorithm, they implement different heuristics (e.g. different pivoting strategies), thus explaining that the respective running times occasionally differ considerably.

With this first experiment, we conclude that the floating-point simplex is not always faster, but has the potential to solve a few extra problems. In the next section, we discuss a way of integrating the floating-point simplex into an SMT solver.

3 Integrating a floating-point simplex into an SMT solver

Our goal is to integrate a floating-point simplex into an SMT solver, keeping the exact simplex to maintain soundness. Finding a solution is generally more costly than just verifying it. The method we propose is to use the floating-point simplex to find a solution and use the exact simplex to check whether the solution is correct.

The set of solutions of the linear programming problem $AX \leq B$ ($X \in \mathbb{R}^n$, $B \in \mathbb{R}^m$, A a $m \times n$ matrix) is a convex polyhedron. Let us now recall the workings of phase II of the simplex algorithm, the optimization phase. It starts from an initial feasible vertex (provided by phase I), and moves from a vertex to neighboring vertex in successive improvements of the objective; it stops if further improvement is impossible, on the optimal solution. There exist different strategies for choosing among possible next vertices, thus different running times. Some strategies may also lead to infinite cycling, which is prevented by using Bland's rule; a typical efficient strategy is steepest descent first, and Bland's rule after maximal number of pivoting iterations. Figure 2 illustrates the result of several iterations over a arbitrary simplex problem in a geometrical perspective, where the current solution is moving through the vertices.

In the case of verification problems, we are not interested in optimization, only in phase I, but the algorithm works similarly. The details of different implementations can be found e.g. in [5, 13, 12].

Once we have a point that represents a solution in the floating-point simplex, we go directly to this point in the exact simplex and verify it [11]. This is much faster than using the exact simplex to search for this solution point, since the number of pivot operations in this case is at most the number of variables, while it is at most exponential when searching. The verification is done by simply executing the exact simplex algorithm which does a linear scan over the variables only when a variable had a bound violated during the check call.

Verification proceeds as follows:

- if the point is a valid solution, the method will detect it immediately and will stop;

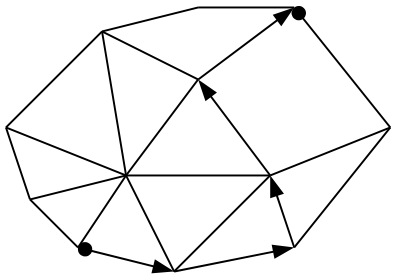

Figure 2: In the simplex algorithm, the point that represents the solution is moving through the neighbor vertices at each iteration.

- in the case the solution given by the floating-point simplex is incorrect, an uncommon case in practice, the exact simplex will find a correct solution starting from that point.

In other words, the verification procedure will use the solution found by the 'fast' floating-point simplex as the initial point of the 'slow' exact simplex, skipping a potentially expensive search every time the solution given is correct, as illustrated in Fig. 3.

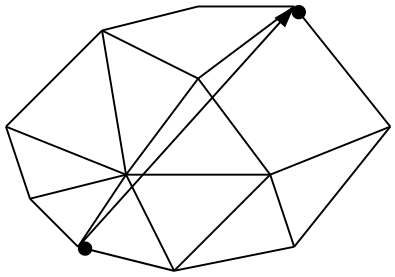

Figure 3: To verify a solution, we move directly to the point we are checking found by the floating-point simplex and 'continue' from there.

In order to direct the exact simplex to the same point reached by the floating-point simplex ("same" being defined as "the intersection of the same set of constraint planes", not "the same numerical value"), we reproduce inside it the same partition of variables into "basic" and "non-basic" and the same use of lower or upper bounds for variables as in the floating-point simplex [11], a purely combinatorial information. This avoids having to extract floating-point information and reinsert it into exact computations.

In any case, facts for the underlying SAT-solver are derived solely from the exact simplex: if the floating-point simplex answers a model, then this model is checked by exact computations, and if it answers that the system is unsatisfiable, then unsatisfiability is checked by the exact simplex, which also generates the conflicts (blocking clauses). In case of disagreement, the exact simplex has the last word.

4 The second experiment

In this section, we compare the combination implementation from Section 3 with the original exact implementation from OpenSMT: does our combination method actually improve the time the SMT solver takes to solve the problems? Figure 4 shows a comparison of their timings, for the 634 benchmarks with a time limit set to 4 minutes. Again, the benchmarks are the QF_LRA section of SMT-LIB.

The total accumulated time of the OpenSMT was 11h06m42s, while the accumulated time of the floating/exact OpenSMT was 13h26m35s. The times include benchmarks that timed out. *Unknown* results are considered time out. 37 problems were only solved by the original OpenSMT, while 8 other problems were only solved by the modified version. Of the problems that only the combination method could solve, 6 are from the *miplib* family and 2 of them are from the *tta_startup* family. The problems of the *miplib* family are a portion of the ones that in the first experiment were solved at least 10 times faster by the floating-point simplex.

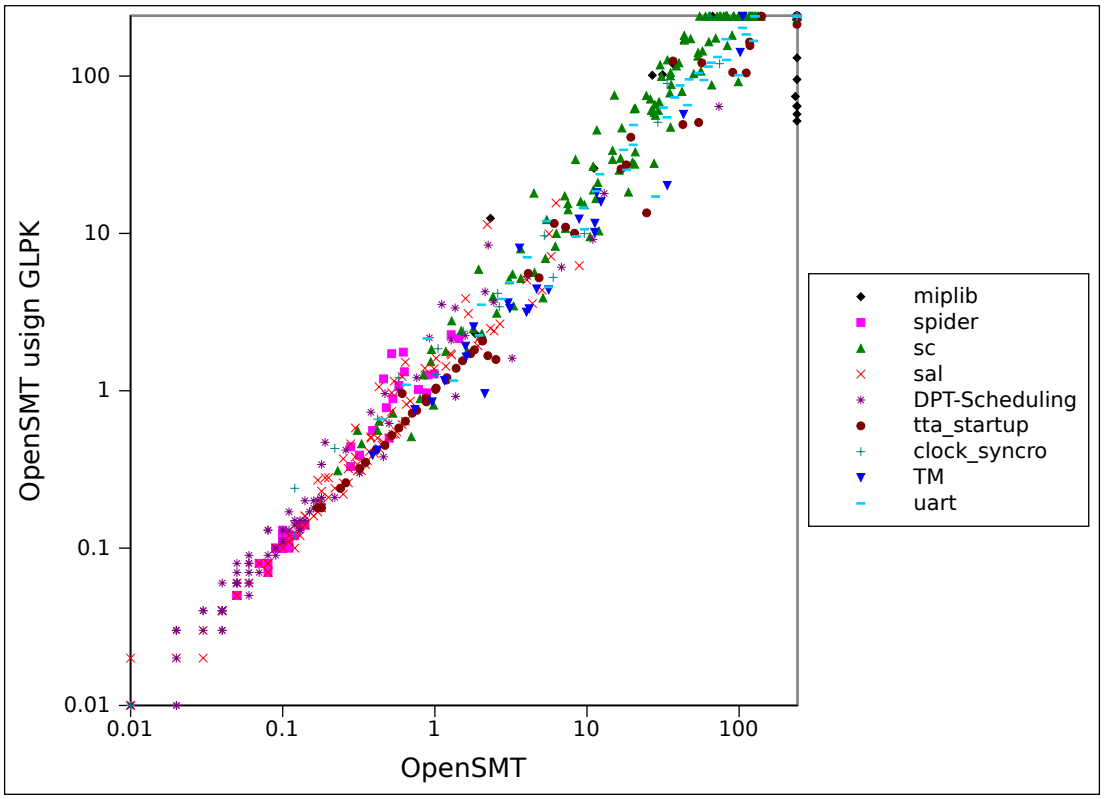

Figure 4: OpenSMT vs floating/exact OpenSMT in QF_LRA.

The result of this second experiment was rather unsurprising given the observations we obtained from the first one. The combined method was able to solve a few extra problems, but overall it was slower: its overhead is generally not compensated by the slightly higher speed of the floating-point simplex, though there exist some cases where it is markedly faster. We recapitulate the main points with a few extra observations here:

Soundness The number of incorrect checks done by GLPK is still very low in this experiment. GLPK and the exact simplex disagreed only on 0.0002% of the queries.

It is well-known that a naive floating-point implementation of the simplex algorithm, by direct implementation of the textbook description of variable selection and pivoting, will generally fail to solve larger problems because floating-point roundoff will cause it to enter "absurd" configurations, loop forever etc. GLPK, however, is not a naive implementation and generally avoids such pitfalls.

Overflow. One of the main reasons one could gain from the use of the floating-point simplex is the presence of overflow, which makes using floats (although unreliable) much faster than using an arbitrary precision library. However, we have seen in Section 2.2 that this does not happen often in QF_LRA and it continues very rare when the search is guided by GLPK. The presence of a few overflows in the family *clock_synchro* was not enough to make the combined method faster.

Size and running time. The average running time of the individual queries continues very low, being less than 1 ms. The high total running time comes from the very large number of linear programming feasibility queries to be solved per SMT problem, and not from the size of each individual query.

According to the GLPK FAQ [9], GLPK is "intended for solving large-scale linear programming [...] problems" and "is able to handle problems with up to 100,000 constraints". However, the average size of the problems in QF_LRA is 500 variables and 200 constraints. The advanced engineering implemented to solve large problems may not be the best to solve small ones. Additionally, the average number of checks is 60,000 per problem in these benchmarks, with consecutive checks usually changing very little. A good implementation of a simplex for SMT-solving is one able to solve fast thousands of relatively small, sparse and very similar set of constraints rather than just one very large.

Finally, there is not a clear reason of why the modified OpenSMT solves the few extra problems. The structure of these problems likely exhibits some characteristics that made them better solvable with the help of GLPK, yet the main reasons we could invoke for the viability of using a floating-point simplex, such as the presence of overflows and the large size of the problems, are not applicable in this case. Neither do these problems have overflows, nor are they very large in size (the average number of variables is 267 and the average number of constraints is 529).

5 Conclusion and possible future work

Despite the investigations of several authors, the merits of the integration of a floating-point simplex implementation into an SMT solver were unclear. We have therefore done a deep experimental comparison between the floating-point and exact simplex approaches on the QF_LRA benchmarks of SMT-LIB to shed some light on the issue.

On the one hand, the reason generally cited for possible better efficiency of the floating-point simplex — the use of expensive extended precision arithmetic in the exact simplex — applies to randomly generated benchmarks (on which pivoting quickly yields dense matrices with large numerators and denominators [11]), but not to benchmarks from SMT-LIB. On the other hand, the weakness generally cited about the floating-point simplex implementation — that floating-point roundoff errors could lead it to wrong answers that would be expensive to correct in order to obtain sound results — neither applies to SMT-LIB benchmarks.

Despite the generally slightly higher speed of the floating-point simplex compared to the exact simplex (our first experiment), the combination method using the floating-point simplex to guide the exact simplex to a solution (thus avoiding a potentially expensive search in exact representation) is generally slightly slower than the exact simplex. We have however identified some families of instances on which it is much faster, but it is unclear which characteristics of the problem produce this behavior and how they could be detected so as to choose which of the two implementations is more likely to be faster.

Our implementation currently reconstructs a starting point for the exact simplex by pivoting of the variables until their partition into basic and nonbasic matches that of the floating-point simplex. This boils down to moving from one base of a vector space to another by Gaussian elimination, and it is well-known that Gaussian elimination in exact rational arithmetic may generate, in its intermediate steps, dense matrices with large numerators and denominators even though the initial and final matrices are sparse and with small numerators and denominators.[5] For this reason, advanced methods for exactly solving systems of linear equations proceed by other means, such as performing Gaussian elimination modulo some prime numbers (if those numbers are not too large, all computations fit within machine words) and reconstructing the solution using the Chinese remainder theorem [14]. It seems possible to use such methods in lieu of pivoting to reconstruct the starting tableau of the exact simplex. It would perhaps be interesting to investigate such methods, though our second experiment indicates that the pivoting in the exact simplex seldom incurs overflows in practice and thus that it is unlikely that the gains from such advanced linear solving methods would be great (especially since they incur some overhead).

Theory propagation consists in deriving opportunistically facts that are implied by the current satisfiable configuration: for instance, if a line in the current simplex tableau implies immediately that $x \leq 20$ and there is a $x \leq 30$ atom A, then the solver can immediately assert A. This tends to be more efficient than waiting until the SAT solver branches on A and, for instance, asserts $\neg A$ only to discover, through more pivoting, that it makes the system unsatisfiable. Currently, theory propagation is performed only by the underlying simplex implementation, and thus only in the rare case where the floating-point simplex answers "unsatisfiable" but the exact simplex disagrees: the results of the floating-point simplex are not trusted in this respect. Two improvements are possible so as to perform more theory propagation. The simpler is to use inequalities "implied" by the floating-point simplex tableau (we use quotes so as to stress the unsound character of this implication) as mere hints in the SAT solver: instead of being asserted as truths, they are suggested as appropriate polarities for the associated atoms. A more ambitious endeavour is to run pivoting steps in the exact simplex (or use linear algebra techniques) in order to exactly reconstruct the tableau lines used for theory propagation, which can then yield sound facts.

References

[1] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB), 2010. `http://www.SMT-LIB.org`.

[2] Frédéric Besson. On using an inexact floating-point LP solver for deciding linear arithmetic in an SMT solver. In *8th International Workshop on Satisfiability Modulo Theories*, Edinburgh, Royaume-Uni, 2010.

[5] Take for instance a square invertible matrix with small integer coefficients and echelonize it, finally obtaining an identity matrix: intermediate steps may still generate rather large numerators and denominators.

[3] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT solver. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 150–153. Springer Berlin / Heidelberg, 2010.

[4] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[5] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

[6] Torbjrn Granlund et al. GNU multiple precision arithmetic library, 2000-2012. `http://gmplib.org/`.

[7] Germain Faure, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodrguez-Carbonell. SAT modulo the theory of linear arithmetic: exact, inexact and commercial solvers. In *Proceedings of the 11th international conference on Theory and applications of satisfiability testing*, SAT'08, page 7790, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In *Computer-aided verification (CAV)*, pages 175–188. Springer, 2004.

[9] Harley Mackenzie. Frequently asked questions about the gnu linear programming kit, 2004.

[10] Andrew Makhorin. GNU linear programming kit, 2000-2008. `http://www.gnu.org/software/glpk/`.

[11] David Monniaux. On using floating-point computations to help an exact linear arithmetic decision procedure. In *Computer-aided verification (CAV)*, number 5643 in LNCS, pages 570–583. Springer, 2009.

[12] Diego Caminha Barbosa De Oliveira. *Fragments de l'arithmétique dans une combinaison de procédures de décision*. PhD thesis, Université Nancy II, March 14 2011.

[13] Harald Rue and Natarajan Shankar. Solving linear arithmetic constraints. Technical Report SRI-CSL-04-01, SRI International, 2004.

[14] William Stein. *Modular forms, a computational approach*, volume 79 of *Graduate studies in mathematics*. American Mathematical Society, 2007.