Internship Report – Extending a Verified SMT Solver for Mixed-Integer Linear Arithmetic

Alban Reynaud

This work has been done in an internship between may and July 2019 in the Department of Computer Science of the University of Innsbruck, under the supervision of René Thiemann, in the Computational Logic group.

Contents

1 Introduction

1.1 The Isabelle Proof Assistant

Isabelle [7] is a generic proof assistant. Isabelle/HOL (abbreviation of *Higher Order Logic*) is a specialization that is generally used.

One of the features of Isabelle is the use of mathematical symbols to make the proofs more readable. It also uses automatic provers. Moreover, there is the language *Isar* that can be used to conduct proofs in a structured way.

For example, here is a proof of the Cantor Theorem (the fact that there is no surjection from a set A to its powerset) [1], with Isabelle and Isar:

```
lemma "¬ surj (f :: 'a ⇒ 'a set)"
proof
  assume "surj f"
  hence "∃ a. {x. x ∉ f x} = f a" by blast
  thus False by auto
qed
```

The outline of the proof is the following:

- We assume that there exist a surjective function f from A to its powerset.
- We deduce that there exists an element a of A such that

$$f(a) = \{x \mid x \notin f(x)\}$$

- But from the following fact we can deduce that both $a \in f(a)$ and $a \notin f(a)$ which is a contradiction (represented by the proposition `False`).

To prove each of these facts, the tactics `blast` and `auto` are used, which internally rely upon automatic provers.

It is also possible to define objects in the functional programming way in Isabelle/HOL and to export code. This way, it is possible to have a certified implementation of an algorithm by writing it in Isabelle/HOL, proving its correctness and exporting its code.

1.2 Linear Arithmetic

Let $a_0, ..., a_{n-1}, b$ be real constants, $x_0, ..., x_{n-1}$ be variables and $\bowtie \in \{\leqslant, \geqslant, <, >\}$ be a comparison operator. With these parameters, we can define a linear constraint c:

$$\sum_{i<n} a_i x_i \bowtie b$$

An assignement v is a function that maps variables to real numbers. We say that the assignement v satisfies the constraint c (or $v \vDash c$) if

$$\sum_{i<n} a_i v(x_i) \bowtie b$$

These formulas define the theory of *linear arithmetic*.

[1] adapted from [6, Section 5.1]

If C is a finite set of linear constraints, we will use the notation $v \vDash C$ for $\forall c \in C.\ v \vDash c$, and say that the assignement c satisfies the set of constraints C. The problem of finding such an assignement is called a *linear problem.*

Note that we are only interested in the decision problem (*i.e* checking if a set of constraints admits a satisfying assignment), not the optimization problem (*i.e* finding an assignment satisfying the constraints that maximizes an objective function).

The other theory we will study is the theory of *mixed-integer linear arithmetic.* A mixed-integer linear problem (MILP) consists of a linear problem where some variables are required to be integral. We will only consider MILPs with only rational coefficients (or equivalently with integral coefficients). Furthermore, if all the variables are required to be integer, the problem is called an *integer linear problem* (ILP).

For example:

$$\left\{ \begin{array}{l} x + y \geqslant 3 \\ y - x < 2 \end{array} \right. \Bigg|\, x \in \mathbb{Z}$$

is a MILP.

More formally, if S in a set of linear constraints and I the set of variables that are required to be integral, if v is a valuation, I will use the notation

$$v \vDash_I S$$

for $v \vDash S$ and $\forall x_i \in I.\ v(x_i) \in \mathbb{Z}$. We will say that the assignement v is a solution to the MILP characterized by the set of constraints C and the set of integral variables I.

Remark: Linear problems with rationnal coefficients are polynomial, while ILPs and MILPs are NP-complete. The NP-hardness can be easily shown by reduction from 3-SAT to 0-1 ILPs. The inclusion in the class NP is more involved to prove, but a sketch of proof will be given later.

Intuitively, it suggests that it is harder to solve MILPs efficiently than linear problems. Nevertheless, we will use Simplex-based algorithms (which are not known to be polynomial in the worst case) to solve linear problems. More generally, we are more interested by algorithms efficient in practice than algorithms optimal in the worst-case.

1.3 SMT Solving

Let T be a quantifier-free theory. A T-solver is an algorithm that checks if a finite set of atoms over T can be satisfied. For linear or mixed-integer linear arithmetic, atoms are linear constraints and the work of a T-solver consists in deciding whether there exists an assignement that satisfies a set of constraints.

Let Φ be a boolean formula of atoms of the theory T. We may want to find an assignement v that satisfies Φ, or to know if no such assignement exists. This problem is called **Satisfiability Modulo Theory** (SMT).

For example, an SMT instance based on linear arithmetic would be:

$$(x + y > 0 \vee z \leqslant 1) \wedge \neg(x \geqslant 0)$$

An efficient procedure to solve SMT problems is DPLL(T) [5, Section 3.2], which works as a combination of a SAT-solver and a T-solver. Here is a quick description of the procedure:

- Replace every atom by a SAT variable to obtain a SAT formula Φ. Run a SAT-solver to find a valuation of each variable to 0 or 1 that satisfies this SAT-formula.
- From this affectation, derive a conjunction of atoms. Run a T-solver to find an assignement that satisfies this conjunction.
- If an assignment v is found, return v
- If no assignment is found, from this conjunction find a contradicting subset of atoms. From this subset, derive a new constraint, add it to the SAT-formula Φ, and go to the first step.

To work efficiently in combination with the SAT-solver, we may assume that the T-solver implements the following interface:

- `Assert`(α): Asserts the atom α. It is added to the set of T-atoms the should be satisfied.
- `Check`(): Runs a T-solver to find an assignment to the set of asserted atoms. If such an assignement is found, returns it. Otherwise, returns a subset of inconsistent asserted atoms.
- `Checkpoint`(): Returns a checkpoint c that contains all the necessary information to backtrack to the current state.
- `Backtrack`(c): Backtracks to the state represented by the checkpoint c.

We will call such an inteface an *incremental interface* as it allows us to add or remove constraints without having to do all the computation from the begining. A similar description can be found in [4] and [10, 1].

An example of an execution of DPLL(T) is given in appendix A.

Dutertre and de Moura have proposed a Simplex-based solver with an incremental interface to solve linear arithmetic problems [4]. A partial version of this algorithm has been formalized in Isabelle by Spasić and Marić [9] and extended to use the incremental interface by Ralph Bottesch, Max Haslbeck and René Thiemann [10, 1]. The ultimate goal of this internship is to extend the previous work to mixed-integer linear arithmetic.

2 Resolution of Mixed-Integer Linear Problems

In this section, I will describe an algorithm to solve MILPs and the mathematics required to prove its correctness.

2.1 The Branch-and-Bound Algorithm

Branch-and-bound is a simple algorithm to solve ILPs and MILPs. Though it can be stated as an optimization algorithm, we will focus on its formulation for decision problems.

Let S and I be respectively the set of linear constraints and the set of integral variables of our problem. The principle of the algorithm is the following:

- Search a valuation v such that $v \vDash S$. If no solution exists, return `Unsatisfiable`.
- If for all $x_i \in I$, $v(x_i) \in \mathbb{Z}$, then $v \vDash_I S$. Return v.
- If there exists a variable $x_i \in I$ such that $v(x_i) \notin \mathbb{Z}$, we can remark that for all solution of this MILP, either the constraint $x_i \leqslant \lfloor v(x_i) \rfloor$ or the constraint $x_i \geqslant \lceil v(x_i) \rceil$ is satisfied.

  We can then try to split our initial problem into two subproblems (this is the *branching step*):

  - Recursively call branch-and-bound on the constraints set
    $$S \cup \{x_i \leqslant \lfloor v(x_i) \rfloor\}$$
    and I. If a solution v' is found, return it.
  - Recursively call branch-and-bound on the constraints set
    $$S \cup \{x_i \geqslant \lceil v(x_i) \rceil\}$$
    and I. If a solution v' is found, return it.
- If after the branching step no solution is found, return `Unsatisfiable`.

The problem with this formulation is that the algorithm may loop. For example, let us examine the following ILP:

$$\left\{ \begin{array}{l} 3x - 3y \geqslant 1 \\ 3x - 3y \leqslant 2 \end{array} \right. \Bigg| \, x, y \in \mathbb{Z} \tag{1}$$

This problem has no solution, and its relaxation (that is the problem composed by the linear inequalities, without the integrality constaints) is unbounded, as shown in figure 2.1. Here is a possible execution of the branch-and-bound algorithm on this problem:

- Obtain the solution $(\frac{1}{3}, 0)$. The variable x is not integral. Try to solve the problem with the extra constaint $x \geqslant 1$.
- Obtain the solution $(1, \frac{1}{3})$. The variable y is not integral. Try to solve the new problem with the extra constraint $y \geqslant 1$.
- Obtain the solution $(1 + \frac{1}{3}, 1)$. The variable x is not integral. Try to solve the problem with the extra constaint $x \geqslant 2$.

- Obtain the solution $(2, 1 + \frac{1}{3})$. The variable y is not integral. Try to solve the new problem with the extra constraint $y \geqslant 2$.
- etc...

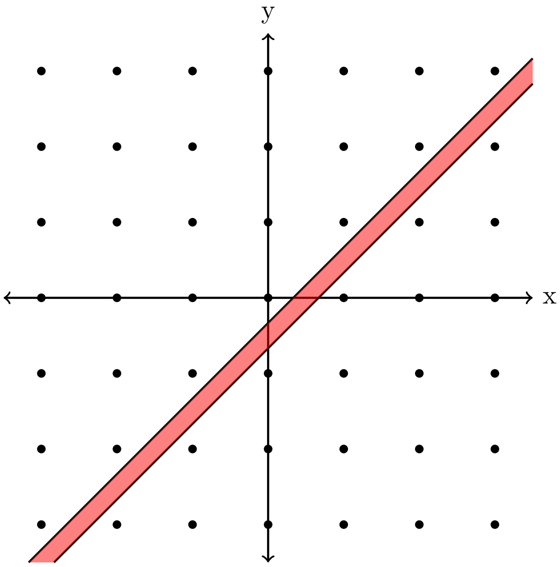

Figure 1: Graphical representation of the solutions of the problem 1. The solutions of the relaxation lie in the red area.

In this case, the problem is unbounded and makes the branch-and-bound algorithm loop. This is why we will try to add constraints of the form

$$-B \leqslant x_i \leqslant B$$

for all variable x_i to bound the problem, at the condition that the bounded problem admits a solution if and only if the original problem admits a solution. This is what we are going to see in the next section.

2.2 Obtaining Bounds for ILPs and MILPs

First of all, let us remark that if a linear problem contains only large inequalities, it could be put in the form:

$$\left\{\begin{array}{ccccccc} a_{11}x_1 & + & ... & + & a_{1n}x_n & \leqslant & b_1 \\ & & \ddots & & & & \\ a_{p1}x_1 & + & ... & + & a_{pn}x_n & \leqslant & b_p \end{array}\right.$$

or in the matrix form $Ax \leqslant b$ with

$$A = (a_{ij})_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant p}}$$

and $b = (b_j)_{1 \leqslant j \leqslant p}$. We identify a valuation of n variables to a vector of dimension n.

Now, we are going to give the outline of a proof that MILPs admit a solution if and only if they admit a solution of small size. In what follows, we will focus on ILPs, but the results can be easily extended to MILPs. Most of the definitions and results can be found in Schrijver's book [8, Sections 7 and 16].

Definition 1 (Finitely Generated Cone)**.** *A set of points C is a* finitely generated cone *if and only if $C = \text{cone }\{x_0, ..., x_{n-1}\}$ where*

$$\text{cone }\{x_0, ..., x_{n-1}\} = \left\{ \sum_{i=0}^{n-1} \lambda_i x_i \mid \lambda_0, ..., \lambda_{n-1} \geqslant 0 \right\}$$

for some $x_0, ..., x_{n-1}$.

Definition 2 (Polyhedral Cone)**.** *A set of points C is a* polyhedral cone *if and only if $C = \{x \mid Ax \leqslant 0\}$ for some matrix A.*

Theorem 1 (Farkas-Minkowsky-Weyl Theorem)**.** *A set C is a finitely generated cone if and only if it is a polyhedral cone.*

Definition 3 (Polyhedron)**.** *A set of points P is a* convex polyhedron *if and only if*

$$P = \{x \mid Ax \leqslant b\}$$

for some matrix A and vector b.

Note that a polyhedron could be interpreted as the set of solutions of a linear problem with only non-strict inequalities.

Definition 4 (Polytope)**.** *A set of points P is a* polytope *if and only if it is the convex hull of a finite set $C = \{x_0, ..., x_{m-1}\}$, i.e*

$$P = \left\{ \sum_{i=0}^{m-1} \lambda_i x_i \mid \lambda_0, ..., \lambda_{m-1} \geqslant 0 \wedge \sum_{i=0}^{m-1} \lambda_i = 1 \right\}$$

We will also use the following notation: $P = \text{hull } C$.

Corollary 1 (Decomposition Theorem for Polyhedra)**.** *A set of points P is a polyhedron if and only if there exists a polytope Q and a finitely generated cone C such that $P = Q + C$.*

To illustrate this theorem, let us take the following problem:

$$\left\{ \begin{array}{rcrcr} -6x & + & 2y & \leqslant & 1 \\ 3x & + & 2y & \geqslant & 11 \\ & & 2y & \geqslant & 1 \\ 2x & - & 4y & \leqslant & 5 \end{array} \right. \tag{2}$$

As we have seen, the set of the solutions of this problem is a polyhedron. On the figure 2.2, the polyhedron is represented by the red and blue areas. As we can see, it is the sum of a polytope (the red triangle) and a cone (cone $\{u, v\}$).

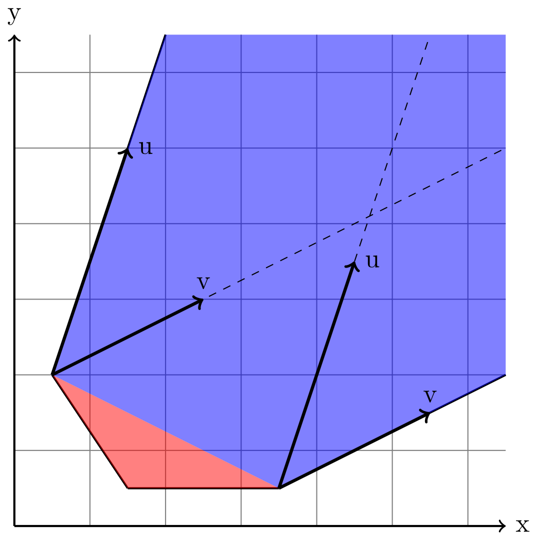

Figure 2: Decomposition of the polyhedron defined by the problem 2

Corollary 2. *P is a polytope if and only if P is a bounded polyhedron.*

Definition 5 (Integer Hull)**.** *Given a polyhedron P, the* integer hull *P_I is the convex hull of the integral vectors of P, i.e $P_I = \text{hull}(P \cap \mathbb{Z}^n\}$.*

Theorem 2 (Meyer (1974))**.** *For any rational polyhedron P, P_I is a polyhedron. Moreover, if $P = Q + C$ with Q a polytope and C a cone, then there exists a polytope Q' such that $P_I = Q' + C$.*

Proof. Let us decompose P into $P = Q + C$ where $C = \text{cone}\ \{y_0, ..., y_{s-1}\}$. All the y_i's are rational vectors. Without loss of generality we can assume that the y_i's are integral.

Let us define

$$B = \left\{ \sum_{i=0}^{s-1} \mu_i y_i \mid 0 \leqslant \mu_i \leqslant 1 \right\}$$

and show that

$$P = (Q + B)_I + C$$

The easy inclusion is the following:

$$(Q + B)_I + C \subseteq P_I + C = P_I + C_I \subseteq (P + C)_I = P_I$$

To prove the other inclusion, let us introduce $x \in P$. There exists $q \in Q$ and $\lambda_0, ..., \lambda_{s-1} \leqslant 0$ such that:

$$x = q + \sum_{i<s} \lambda_i y_i$$

But for every i, $\lfloor\lambda_i\rfloor y_i$ is an integral vector, so

$$x - \sum_{i<n}\lfloor\lambda_i\rfloor y_i = q + \sum_{i<n}(\lambda_i - \lfloor\lambda_i\rfloor)y_i$$

is integral, and as $\sum_{i<n}(\lambda_i - \lfloor\lambda_i\rfloor)y_i$ is contained in B:

$$x - \sum_{i<n}\lfloor\lambda_i\rfloor y_i \in (Q+B)_I$$

Finally: $x \in (Q+B)_I + C$

$(Q+B)$ is bounded, so $(Q+B)_I$ is the convex hull of a finite set, it is a polytope. If we take $Q' = (Q+B)_I$, we have the result. □

A graphical representation of this proof is given in figure 2.2. With again the polytope defined by the problem 2, the set boundary of the set $(Q+B)$ is represented in red, $(Q+B)_I$ is represented in green. As we can see, $(Q+B)_I + C$ (the green and yellow areas) is exactly the integer hull of the polyhedron.

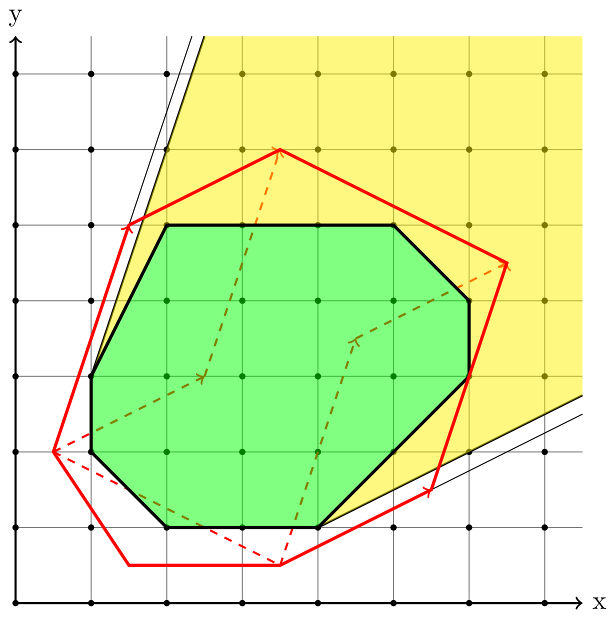

Figure 3: Decomposition of the integer hull of the polytope defined by the problem 2

Remark: Using the notations of the previous proof:

$$P \cap \mathbb{Z}^n \neq \emptyset \iff P_I \neq \emptyset \iff (Q+B)_I \neq \emptyset \iff (Q+B) \cap \mathbb{Z}^n \neq \emptyset$$

Moreover, we can assume that the vectors $\{y_0, ..., y_{s-1}\}$ are linearly independant [2], which implies that $s \leqslant n$.

Finally, if $Q = \text{hull } \{x_0, ..., x_{k-1}\}$, then for every $z \in Q + B$, $z \in [-B, B]^n$ where

$$B = \max_{i,j} |x_{ij}| + n \cdot \max_{i,j} |y_{ij}|$$

With this remark, what remains to do is to find a bound B that depends on the parameters of the problem. To do so, a solution is to generalize the previous theorems to incorporate bounds. We will use the notation $[\![a, b]\!]$ for the set of integers in the interval $[a, b]$.

Theorem 3 (Generalized Farkas-Minkowsky-Weyl Theorem)**.** *A set C is a finitely generated cone if and only if is is a polyhedral cone.*

Moreover, if $C = \{x \mid Ax \leqslant 0\}$ and $A \in [\![-m, m]\!]^{p\times n}$ with $m \geqslant 1$, there exists a finite set Y such that $Y \subseteq [\![-B, B]\!]^n$ and

$$C = \text{cone } Y$$

where

$$B = (n-1)! \cdot m^{n-1}$$

Corollary 3 (Generalized Decomposition Theorem)**.** *A set of points P is a polyhedron if and only if there exists a polytope Q and a finitely generated cone C such that $P = Q + C$.*

Moreover, if $P = \{x \mid Ax \leqslant b\}$ with $A \in [\![-m, m]\!]^{p\times n}$, $b \in [\![-m, m]\!]^p$, and $m \geqslant 1$, then there exists finite sets $Q \subseteq [-B, B]$ and $X \subseteq [\![-B, B]\!]$ such that

$$P = \text{hull } Q + \text{cone } X$$

with

$$B = n! \cdot m^n$$

We finally have the desired result:

Corollary 4 (Existence of small solutions for ILPs)**.** *If $P = \{x \mid Ax \leqslant b\}$ with $A \in [\![-m, m]\!]^{p\times n}$, $b \in [\![-m, m]\!]^p$ and $m \geqslant 1$, then*

$$P \cap \mathbb{Z}^n \neq \emptyset \iff P \cap [\![-B, B]\!]^n \neq \emptyset$$

where

$$B = (n+1)! \cdot m^n$$

With the same outline of proof, we can obtain a more general result that encompasses MILPs and strict inequalities.

[2] It is reasonnable using the fundamental theorem of inequalities that implies that every point of the cone C can be expressed as a non-negative linear combination of an independant set of $\{y_0, ..., y_{s-1}\}$. Using this, we can prove a similar result in the general case.

Theorem 4 (Existence of small solutions for MILPs)**.** *If I is a subset of $[\![1, n]\!]$ and*

$$P = \{x \mid Ax \leqslant b \wedge A'x < b'\}$$

with $A \in [\![-m, m]\!]^{p \times n}$, $b \in [\![-m, m]\!]^p$, $A' \in [\![-m, m]\!]^{p' \times n}$, $b' \in [\![-m, m]\!]^{p'}$ and $m \geqslant 1$, then

$$\{x \in P \mid \forall x_i \in I.\ x_i \in \mathbb{Z}\} \neq \emptyset \Longleftrightarrow \{x \in P \mid \forall x_i \in I.\ x_i \in [\![-B, B]\!]\} \neq \emptyset$$

where

$$B = (n+1)! \cdot m^n$$

Remark: If we take the previously defined bound B:

$$\log B \approx n \log n + n \log m$$

This way, we can use this bound B to show that the number of bits necessary to represent numbers in the interval $[\![-B, B]\!]$ is polynomial in the size of the input, and so that ILPs are NP.

3 Formalization in Isabelle

As we have presented a method to solve MILPs, we can focus on its formalization, which has been the main part of the work during this internship. It consisted first in the formalization of Schrijver results concerning linear inequalities (section 3.1), then in the implementation and the proof of correction of a branch-and-bound algorithm (section 3.2).

3.1 Formalization of Results concerning Linear Inequalities

This part concerns the formalization of the results from Schijver's book described in section 2.2. Also, other results including Farkas' lemma and the Carathéodory's theorem have been formalized.

This was done in collaboration with René Thiemann and Ralph Bottesch:

- René included bounds for the various operations that are performed when executing the decomposition theorem. Moreover, he verified Theorem 2 of Meyer.
- Ralph and René formalized the fundamental theorem of linear inequalities.
- I mainly focused on the demonstration of the Farkas-Minkowsky-Weyl and the decomposition theorem.

In total, 5551 lines of proofs were written. The work was published in the *Archive for Formal Proofs* (AFP), a collection of proof libraries in Isabelle [2].

The formalization follows the outline given in section 2.2. Several theorems, for example the Farkas-Minkowsky-Weyl theorem, have a general formulation:

```
lemma farkas_minkowsky_weyl_theorem:
  "(∃ X. X ⊆ carrier_vec n ∧ finite X ∧ P = cone X)
  ⟷ (∃ A nr. A ∈ carrier_mat nr n ∧ P = polyhedral_cone A)"
```

and particular formulations including bounds:

```
lemma farkas_minkowsky_weyl_theorem_2:
  assumes A: "A ∈ carrier_mat nr n"
  shows "∃ X. X ⊆ carrier_vec n ∧ finite X
           ∧ polyhedral_cone A = cone X
  ∧ (A ∈ ℤm ∩ Bounded_mat Bnd ⟶
     X ⊆ ℤv ∩ Bounded_vec (det_bound (n-1) (max 1 Bnd)))"
```

where the following Isabelle notations are used:

- `carrier_vec` n is the set of vectors $\mathbb{K}^n$ where $\mathbb{K}$ is a field.
- `carrier_mat` n_r n_c is the set of matrices $\mathbb{K}^{n_r \times n_c}$.
- $\mathbb{Z}_v$ is the set of integral vectors.
- $\mathbb{Z}_m$ is the set of integral matrices.
- `Bounded_vec` B is the set of vectors with coefficients in the interval $[-B, B]$.
- `Bounded_mat` B is the set of matrices with coefficients in the interval $[-B, B]$.
- `det_bound` n m is $n! \cdot m^n$.

A difficulty I was confronted to is the different definitions of linear combinations in Isabelle. Indeed, the standard definition is

$$\texttt{lincomb}\; c\; V = \sum_{v \in V} c(v) \cdot v$$

where V is a set of vectors and c is a map from vectors to scalars.

A problem that can occur is the distributivity when we multiply this sum by a matrix. Indeed, it is clear that

$$A \sum_i \lambda_i x_i = \sum_i \lambda_i \cdot A x_i.$$

But this equality cannot be simply stated with the operator `lincomb` Indeed, it is possible that there exists two vectors x_0 and x_1 such that $Ax_0 = Ax_1$. If we want to express $\sum_i \lambda_i \cdot Ax_i$ using `lincomb`, then the scalar coefficient associated to Ax_0 is neither $c(x_0)$ nor $c(x_1)$ but something like $\sum_{x.Ax=Ax_0} c(x)$, which is more complicated to work with. More generally, this problem occurs everytime we want to state the equality

$$f(\sum_i \lambda_i x_i) = \sum_i \lambda_i f(x_i)$$

where f is a linear map.

A solution to tackle this problem is to use linear combinations over lists, defined this way:

$$\texttt{lincomb\_list}\ c\ \texttt{Vs} = \sum_{i=0}^{l-1} c(i) \cdot v_i$$

where $\texttt{Vs} = [v_0, ..., v_{l-1}]$ is a list of vectors and c is a function mapping integers to scalars.

This way, the distributivity of a linear map to a linear combination can be expressed in a direct manner. To use the advantages of both definitions, we usually define our concepts in two manners: one using sets and the other using lists. For example, here is how non-negative linear combinations are defined, for sets and for lists:

```
definition "nonneg_lincomb c Vs b =
  (lincomb c Vs = b ∧ c ‘ Vs ⊆ {x. x ≥ 0})"

definition "nonneg_lincomb_list c Vs b =
  (lincomb_list c Vs = b ∧ (∀ i < length Vs. c i ≥ 0))"
```

where $\texttt{c ‘ Vs} = \{\texttt{c}(x) \mid x \in \texttt{Vs}\}$ is the image of the set Vs under c.
We can then use this definition to define convex combinations:

```
definition "convex_lincomb c Vs b =
  (nonneg_lincomb c Vs b ∧ sum c Vs = 1)"

definition "convex_lincomb_list c Vs b =
  (nonneg_lincomb_list c Vs b ∧ sum c {0..<length Vs} = 1)"
```

and then define convex hulls:

```
definition "convex_hull Vs =
  {x. ∃ Ws c. finite Ws ∧ Ws ⊆ Vs ∧ convex_lincomb c Ws x}"

definition "convex_hull_list Vs =
  {x. ∃ c. convex_lincomb_list c Vs x}"
```

Finally, we have to show that these two definitions are equivalent:

```
lemma finite_convex_hull_iff_convex_hull_list:
  assumes Vs: "Vs ⊆ carrier_vec n"
  and id': "Vs = set Vsl'"
shows "convex_hull Vs = convex_hull_list Vsl'"
```

This kind of equivalence is not always trivial to prove. For example, it took me around 300 lines of code to prove the previous result.

3.2 Formalization of a Branch-and-Bound Algorithm

The next part is the formalization of a branch-and-bound algorithm, as described in the section 2.1. The idea was to state and prove the correctness of a simple algorithm, without any optimization for the moment.

I first developped a function `branch_and_bound_core` that takes four arguments:

- A list of linear constraints `cs`.
- A list of variables `Is` required to be integral.
- A function `lb` mapping variables of `Is` to integral lower bounds.
- A function `ub` mapping variables of `Is` to integral upper bounds.

It returns either `Some` v is a solution v within the bounds is found, or `None` otherwise. The code of this function can be found in appendix B

The goal of the arguments `lb` and `ub` is to bound the problem. This way, I can prove that this function always terminates. To do so, a convenient way in Isabelle is to use a *measure*. It is function mapping the arguments of the function `f` that we want to prove its termination to natural numbers. If we prove that at each recursive call the measure strictly decreases, then there is no way that a infinite sequence of recursive calls can happen and the function always terminates. The measure I used is the following:

$$\max\left(0, \sum_{x_i \in \texttt{Is}} (\texttt{ub}(x_i) - \texttt{lb}(x_i))\right)$$

Once the termination proved, I proved correctness properties, the first is that if a valuation is found it satisfies the MILP:

```
lemma branch_and_bound_core_sat:
  "branch_and_bound_core cs Is lb ub = Some v $\Longrightarrow$
   v $\models_{mcs}$ (set cs, set Is)"
```

and the other is that if no valuation is found, then there is no valuation `v` that satisfies the MILP and such that the valuations of the integral variables are bounded by `lb` and `ub`:

```
lemma branch_and_bound_core_unsat:
  "branch_and_bound_core c Is lb ub = None $\Longrightarrow$
   $\forall$ i $\in$ set Is. of_int (lb i) $\leq$ v i $\wedge$ v i $\leq$ of_int (ub i) $\Longrightarrow$
   $\neg$ (v $\models_{mcs}$ (set c, set Is))"
```

where the Isabelle notation $v \models_{mcs} (S, I)$ is equivalent to notation $v \models_I S$.

The goal now is to use the theorem 4 to have a formalized complete solver for MILPs. The problem is that linear constraints are coded with a datastructure called *linear polynomials* that should be converted to the form of vectors and matrices to use the theorem.

To do so, linear constraints are converted to the type `Le_Constraint` that is more practical to use, as the description of different inequalities uses a sum-type with only two constructors instead of ten for the type `constraint`. Indeed, I first tried to express the conversion using directly the standard type, and when I switched to `Le_Constraint`'s, I considerably simplified the proofs (around 80 lines of proof were saved).

Afterwards, some work to convert rational constraints to integral constaints is needed. I use it to define a function `compute_bound` that maps a list of constraints to the desired bound, and prove that:

```
lemma compute_bound:
  assumes "v ⊨_mcs (set cs, I)"
  shows "∃ v. v ⊨_mcs (set cs, I) ∧
         (∀ i ∈ I. |v i| ≤ of_int (compute_bound cs))"
```

Finally, I am able to state a complete MILP-solver based on the branch-and-bound algorithm:

```
definition branch_and_bound :: "constraint list ⇒ var list ⇒ rat
valuation option"
where "branch_and_bound cs Is = (
  let Bnd = compute_bound cs in
  branch_and_bound_core cs Is (λ _. -Bnd) (λ _. Bnd))"
```

and to prove its correction:

```
lemma branch_and_bound_sat:
  "branch_and_bound cs Is = Some v ⟹ v ⊨_mcs (set cs, set Is)"

lemma branch_and_bound_unsat:
  assumes "branch_and_bound cs Is = None"
  shows "¬ v ⊨_mcs (set cs, set Is)"
```

In total, the formalization of this algorithm takes 1062 lines of code. You can find the state of my work at the end of this internship in this repository: `https://github.com/Cemoixerestre/MILP-Isabelle`. Notice that this is only a temporary repository, this work is not finished as you will see in the next section.

4 Future Work

Even if I have written and proved the correctness of a branch-and-bound algorithm, the work is not yet finished. Indeed, some tasks are needed before having a more effective formalized MILP-solver with an incremental interface. Here is the outline of what should be done after my internship to complete the work.

4.1 Using the Incremental Interface

We can remark that for the branch-and-bound algorithm, at the branching step, we only try to solve problems with a single constraint added to the original problem. This is why it should be interesting to use the incremental interface describe in section 1.3. Indeed, in the current implementation, the computation (including the preprocessing steps) is always made from scratch.

The problem is that in the current interface, the internal state is initialized with a (finite) list of constraints, and only constraints of this list can be asserted. To efficiently use the incrementality, this interface should be rewritten so that arbitrary constraints can be asserted.

4.2 Gomory Cuts

Suppose that a valuation v such that $v \vDash S$ has been found but that $v \nvDash_I S$. A *cut* is a linear constraint c such that v does not satisfy c but for every mixed-integer solution $v \vDash_I S$, v satisfies c. The goal of a cut is to prune non-integral solutions of the problems without affecting integral solutions.

For example, let us take the following ILP:

$$\left\{ \begin{array}{rcrcl} 2x & + & 10y & \geqslant & 5 \\ -2x & + & 10y & \leqslant & 5 \end{array} \right| x, y \in \mathbb{Z} \tag{3}$$

$(0, \frac{1}{2})$ is a non-integral solution. But we can remark that the following constraint

$$x \geqslant \frac{5}{2}$$

is satisfied by every integral solution of the problem, but not by $(0, \frac{1}{2})$. We will call this constraint the *cut*. A graphical representation can be found in figure 4.2.

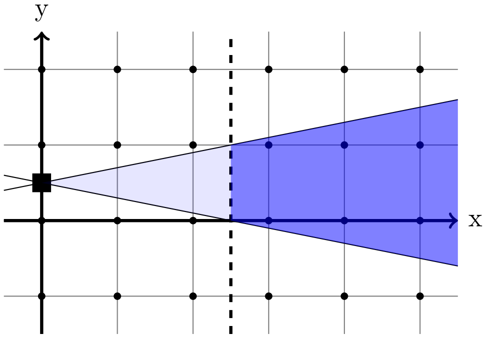

Figure 4: Graphical representation of the solutions of the problem 3. The solutions lie in the blue and light blue areas. More precisely, the solutions satisfying the cut lie in the blue area.

A general method to obtain cuts is described in [4, Section 4]. These cuts are called *Gomory cuts*. The combination of the branch-and-bound algorithm with cut generation is called *branch-and-cut* and a dramatic acceleration of the branch-and-bound algorithm has been observed [3].

It is planned to incorporate Gomory cuts in this branch-and-bound algorithm, and to prove its correction. Like in the previous section, it would be useful to extend the following interface so that arbitrary constraints (such as Gomory cuts) can be asserted.

4.3 Finalization

To finish the work, the branch-and-cut function should be extended to an incremental interface similar to the existing interface for linear problems so that it could be used efficiently by a SMT-solver.

Also, René have already started to export the code to do measure its execution time. It could be interesting to measure to difference obtained by each of the previously described optimizations. Finally, it is planned to publish this work in AFP.

Conclusion

During this internship, I contributed to formalize in Isabelle results about linear inequalities with René Thiemann and Ralph Bottesch. I used these results to prove the termination of a branch-and-bound algorithm to solve MILPs. I also proved the correctness of this algorithm.

This is the first step to extending a verified linear arithmetic solver to mixed-integer linear arithmetic, with an incremental interface so that it could be plugged in a SMT-solver.

References

[1] Ralph Bottesch, Max W. Haslbeck, and René Thiemann. Verifying an incremental theory solver for linear arithmetic in Isabelle/HOL. In *Proceedings of the 12th International Symposium on Frontiers of Combining Systems*, volume 11715 of *LNAI*, 2019. To appear.

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

[3] Gérard Cornuéjols. The ongoing story of gomory cuts. In *DOCUMENTA MATH.* Citeseer, 2012.

[4] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the 18th International Conference on Com-*

puter Aided Verification, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.

[5] Daniel Kroening and Ofer Strichman. *Decision procedures.* Springer, 2016.

[6] Tobias Nipkow and Gerwin Klein. *Concrete Semantics - With Isabelle/HOL.* Springer, 2014.

[7] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002.

[8] Alexander Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, 1998.

[9] Mirko Spasić and Filip Marić. Formalization of incremental simplex algorithm by stepwise refinement. In *International Symposium on Formal Methods*, volume 7436 of *LNCS*, pages 434–449. Springer, 2012.

[10] René Thiemann. Extending a verified simplex algorithm. In *LPAR-22 Workshop and Short Paper Proceedings. Kalpa Publications in Computing*, volume 9, pages 37–48, 2018.

A Example: an Execution of DPLL(T)

Let us solve the following SMT-instance (based on linear arithmetic):

$$\Phi \equiv (A \vee B \vee C) \wedge (\neg A \vee B) \wedge (\neg A \vee C \vee D) \wedge (\neg C \vee D)$$

with

$$\begin{array}{ccccccc} A & \equiv & x & + & y & \geqslant 3 \\ B & \equiv & x & & & \leqslant 1 \\ C & \equiv & & & y & \leqslant 1 \\ D & \equiv & y & - & x & < 2 \end{array}$$

using DPLL(T). First, let us interpret Φ as a SAT-formula and find an valuation that satisfies it. For example:

- Arbitrarily affect the variable A to 1. Assert the atom A. Get a checkpoint c_1.
- To solve the clause $(\neg A \vee B)$, we must affect the variable B to 1. Assert the atom B. Get a checkpoint c_2.
- Affect the variable C to 1. Assert the atom C. Get a checkpoint c_3.
- To solve the clause $(\neg C \vee D)$, we must affect the variable D to 1. Assert the atom D. Get a checkpoint c_4.

Now, we have found an valuation that satisfies Φ interpreted as a SAT-formula. But we need to check if this valuation is compatible with an assignment in the theory of linear arithmetic. It means that we need to find an assignment to the conjunction $A \wedge B \wedge C \wedge D$, which is equivalent to the system:

$$\left\{\begin{array}{ccccl} x & + & y & \geqslant 3 & (A) \\ x & & & \leqslant 1 & (B) \\ & & y & \leqslant 1 & (C) \\ y & - & x & < 2 & (D) \end{array}\right.$$

But this system has no solution. The procedure `Check`() may retun that the constraints A, B and C are mutually incompatible. As these three constraints cannot be satisfied simultaneously, an assignement that satisfies Φ must violate at least one of these constraints, so we can deduce that the clause $(\neg A \vee \neg B \vee \neg C)$ is true. Instead of solving Φ, we will try to solve the formula

$$\Phi' = \Phi \wedge (\neg A \vee \neg B \vee \neg C)$$

We need to backtrack, but we can notice that we could bactrack just before the choice to affect C to 1 was made. So let us backtrack to c_2, where only the variables A and B are affected.

- To solve the clause $(\neg A \vee \neg B \vee \neg C)$, we must affect the variable C to 0. Assert the atom $\neg C$. Get a checkpoint c'_3.
- To solve the clause $(\neg A \vee C \vee D)$, we must affect the variable D to 1. Assert the atom D. Get a checkpoint c'_4.

Again, we have a valuation that satisfies Φ' interpreted as a SAT-formula. We have to find an assignement that satisfies the conjunction $A \wedge B \wedge \neg C \wedge D$, which is equivalent to the system:

$$\left\{\begin{array}{ccccl} x & + & y & \geqslant 3 & (A) \\ x & & & \leqslant 1 & (B) \\ & & y & > 1 & (\neg C) \\ y & - & x & < 2 & (D) \end{array}\right.$$

`Check`() may return the assignement $(x = 1, y = 2)$. Finally, the formula Φ is satisfiable, and $(x = 1, y = 2) \vDash \Phi$.

B Code of the Core of the Branch-and-Bound Procedure

```
function branch_and_bound_core ::
  "constraint list ⇒ var list ⇒ (var ⇒ int) ⇒ (var ⇒ int)
     ⇒ rat valuation option" where
  "branch_and_bound_core cs Is lb ub =
(case simplex (cs @ bounds_to_constraints Is lb ub) of
   Unsat _ ⇒ None
 | Sat r ⇒ (let v = ⟨r⟩ in
     case find (λ x. v x ∉ ℤ) Is of
       None ⇒ Some v
     | Some x ⇒ (
         let lb' = (λ y. if y = x then ⌈v x⌉ else lb y) in
         let ub' = (λ y. if y = x then ⌊v x⌋ else ub y) in
         let sol = branch_and_bound_core cs Is lb ub' in
         if sol ≠ None then sol
         else branch_and_bound_core cs Is lb' ub)))"
```

To understand the code:

- `bounds_to_constraints Is lb ub` returns a list containing the constraints $x_i \geqslant$ `lb` x_i and $x_i \leqslant$ `ub` x_i for all element x_i of the list `Is`.
- " `@` " is the symbol used for list concatenation. In this case, `cs @ bounds_to_constraints Is lb ub` is the list composed by the constraints of `cs` and of `bounds_to_constraints Is lb ub`.
- The function `simplex` takes as arguments a list of constraints and returns `Unsat` if there is no rational solution to this combination of constraints, or `Sat v` such that ⟨v⟩ is a rational valuation satisfying these constraints.
- The function `find` takes as arguments a predicate `f` and a list `Is` and returns either `Some` x_i where x_i is an element of `Is` such that `f` x_i is true, or `None` if no such element exists.