Fast Reflexive Arithmetic Tactics the linear case and beyond

Frédéric Besson*

Irisa/Inria, Campus de Beaulieu, 35042 Rennes Cedex, France

Abstract. When goals fall in decidable logic fragments, users of proof-assistants expect automation. However, despite the availability of decision procedures, automation does not come for free. The reason is that decision procedures do not generate proof terms. In this paper, we show how to design efficient and lightweight reflexive tactics for a hierarchy of quantifier-free fragments of integer arithmetics. The tactics can cope with a wide class of linear and non-linear goals. For each logic fragment, off-the-shelf algorithms generate *certificates* of infeasibility that are then validated by straightforward *reflexive checkers* proved correct inside the proof-assistant. This approach has been prototyped using the Coq proof-assistant. Preliminary experiments are promising as the tactics run fast and produce small proof terms.

1 Introduction

In an ideal world, proof assistants would be theorem provers. They would be fed with theorems and would either generate a proof of them (if one exists) or reject them (if none exists). Unfortunately, in real life, theorems can be undecidable Yet, theorems often fall in decidable fragments. For those, users of proof assistants expect the proof process to be discharged to dedicated efficient decision procedures. However, using off-the-shelf provers is complicated by the fact that they cannot be trusted. A decision procedure, that when given as input a supposedly-so theorem, laconically answers back *yes* is useless. Proof assistants only accept proofs which they can check by their own means.

Approaches to obtain proofs from decision procedures usually require a substantial engineering efforts and a deep understanding of the internals of the decision procedure. A common approach consists in instrumenting the procedure so that it generates proof traces that are *replayed* in the proof-assistant. The Coq `omega` tactic by Pierre Crégut ([9] chapter 17) is representative of this trend. The tactics is a decision procedure for quantifier-free linear integer arithmetics. It generates Coq proof terms from traces obtained from an instrumented version of the Omega test [22]. Another approach, implemented by the Coq `ring` tactics [13], is to prove correct the decision procedure inside the proof-assistant and use computational reflection. In this case, both the computational complexity

* This work was partly funded by the IST-FET programme of the European Commission, under the IST-2005-015905 MOBIUS project.

of the decision procedure and the complexity of proving it correct are limiting factors.

In this paper, we adhere to the so-called *sceptical approach* advocated by Harrison and Théry [16]. The key insight is to separate proof-search from proof-checking. Proof search is delegated to fine-tuned external tools which produce certificates to be checked by the proof-assistant. In this paper, we present the design, in the Coq proof-assistant, of a tactics for a hierarchy of quantifier-free fragments of integer arithmetics. The originality of the approach is that proof witnesses, *i.e.,* certificates, are computed by black-box off-the-shelf provers. The soundness of a *reflexive* certificate checker is then proved correct inside the proof-assistant. For the logic fragments we consider, checkers are considerably simpler than provers. Hence, using a pair (untrusted prover, proved checker) is a very lightweight and efficient implementation technique to make decision procedures available to proof-assistants.

The contributions of this paper are both theoretical and practical. On the theoretical side, we put the shed on mathematical theorems that provide infeasibility certificates for linear and non-linear fragments of integer arithmetics. On the practical side, we show how to use these theorems to design powerful and space-and-time efficient checkers for these certificates. The implementation has been carried out for the Coq proof-assistant. Experiments show that our new reflexive tactic for linear arithmetics outperforms state-of-the-art Coq tactics.

The rest of this paper is organised as follows. Section 2 recalls the principles of reflection proofs. Section 3 presents the mathematical results on which our certificate checkers are based on. Section 4 describe our implementation of these checkers in the Coq proof-assistant. Section 5 compares to related work and concludes.

2 Principle of reflection proofs

Reflection proofs are a feature of proof-assistants embedding a programming language. (See Chapter 16 of the Coq'Art book [2] for a presentation of reflection proofs in Coq.) In essence, this technique is reducing a proof to a computation. Degenerated examples of this proof pattern are equality proofs of *ground*, *i.e.,* variable free, arithmetic expressions. Suppose that we are given the proof goal

$$4 + 8 + 15 + 16 + 23 + 42 = 108$$

Its proof is quite simple: evaluate $4 + 8 + 15 + 16 + 23 + 42$; check that the result is indeed 108. Reflection proofs become more challenging when goals involves variables. Consider, for instance, the following goal where x is universally quantified:

$$4 \times x + 8 \times x + 15 \times x + 16 \times x + 23 \times x + 42 \times x = 108 \times x$$

Because the expression contain variables, evaluation alone is unable to prove the equality. The above goal requires a more elaborate reflection scheme.

2.1 Prover-based reflection

Reflection proofs are set up by the following steps:

1. encode logical propositions into symbolic expressions F;
2. provide an evaluation function $[\![.]\!] : Env \rightarrow F \rightarrow Prop$ that given an environment binding variables and a symbolic expression returns a logical proposition;
3. implement a semantically sound and computable function $prover : F \rightarrow bool$ verifying $\forall ef, prover(ef) = true \Rightarrow \forall env, [\![ef]\!]_{env}$.

A reflexive proof proceeds in the following way. First, we construct an environment env binding variables. (Typically, variables in symbolic expressions are indexes and environments map indexes to variables.) Then, the goal formula f is replaced by $[\![ef]\!]_{env}$ such that, by computation, $[\![ef]\!]_{env}$ evaluates to f. As the prover is sound, if $prover(ef)$ returns $true$, we conclude that f holds.

Example 1 *Following the methodology described above, we show how goals of the form $c_1 \times x + \ldots c_n \times x = c \times x$ (where the c_is are integer constants and x is the only universally quantified variable) can be solved by reflection.*

- *Such formulae can be coded by pairs $([c_1; \ldots; c_n], c) \in F = \mathbb{Z}^* \times \mathbb{Z}$.*
- *The semantics function $[\![.]\!]$ is defined by $[\![l, c]\!]_x \stackrel{\triangle}{=} listExpr(x, l) = c \times x$ where $listExpr : \mathbb{Z} \times \mathbb{Z}^* \rightarrow \mathbb{Z}$ is defined by*

$$\begin{aligned} listExpr(x, []) &\stackrel{\triangle}{=} 0 \\ listExpr(x, [c]) &\stackrel{\triangle}{=} c \times x \\ listExpr(x, c :: l) &\stackrel{\triangle}{=} c \times x + listExpr(x, l) \end{aligned}$$

- *Given a pair (l, c), the prover computes the sum of the elements in l and checks equality with the constant c.*

$$prover(l, c) \stackrel{\triangle}{=} (fold\ +\ l\ 0) = c$$

To make decision procedure available to proof-assistants, the reflexive prover-based approach is very appealing. It is conceptually simple and can be applied to any textbook decision procedure. Moreover, besides soundness, it allows to reason about the completeness of the prover. As a result, the end-user of the proof-assistant gets maximum confidence. Upon success, the theorem holds; upon failure, the theorem is wrong.

2.2 Checker-based reflection

On the one hand, provers efficiency is due to efficient data-structures, clever algorithms and fine-tuned heuristics. On the other hand, manageable soundness proofs usually hinge upon simple algorithms. In any case, reflection proofs require runtime efficiency. Obviously, these facts are difficult to reconcile. To obtain a good trade-off between computational efficiency and proof simplicity, we advocate implementing and proving correct *certificate checkers* instead of genuine provers. Compared to provers, checkers take a certificate as an additional input and verify the following property:

$$\forall ef, (\exists cert, checker(cert, ef) = true) \Rightarrow \forall env, [\![ef]\!]_{env}$$

The benefits are twofold : checkers are simpler and faster. Complexity theory ascertains that checkers run faster than provers. In particular, contrary to all known provers, checkers for NP-complete decision problems have polynomial complexity. A reflexive proof of $[\![ef]\!]_{env}$ now amounts to providing a certificate *cert* such that $checker(cert, t)$ evaluates to *true*. As they are checked inside a proof-assistant, certificates can be generated by any untrusted optimised procedure.

Using certificates and reflexive checkers to design automated tactics is not a new idea. For instance, proof traces generated by instrumented decision procedures can be understood as certificates. In this case, reflexive checkers are trace validators which verify the logical soundness of the proof steps recorded in the trace. The Coq `romega` tactic [8] is representative of this trace-based approach: traces generated by an instrumented version of the Omega test [22] act as certificates that are validated by a reflexive checker.The drawback of this method is that instrumentation is an intrusive task and require to dig into the internals of the decision procedure. In the following, we present conjunctive fragments of integer arithmetics for which certificate generators are genuine off-the-shelf provers. The advantage is immediate: provers are now black-boxes. Moreover, the checkers are quite simple to implement and prove correct.

3 Certificates for integer arithmetics

In this part, we study a hierarchy of three quantifier-free fragments of integer arithmetics. We describe certificates, off-the-shelf provers and certificate checkers associated to them. We consider formulae that are conjunctions of inequalities and we are interested in proving the unsatisfiability of these inequalities. Formally, formulae of interest have the form:

$$\neg\left(\bigwedge_{i=1}^{k} e_i(x_1, \ldots, x_n) \geq 0\right)$$

where the e_is are fragment-specific integer expressions and the x_is are universally quantified variables.

For each fragment, we shall prove a theorem of the following general form:

$$(\exists cert, Cond(cert, e_1, \ldots, e_k)) \Rightarrow \forall(x_1, \ldots, x_n), \neg\left(\bigwedge_{i=1}^{k} e_i(x_1, \ldots, x_n)\right)$$

In essence, such a theorem establish that *cert* is a certificate of the infeasibility of the e_is. We then show that certificates can be generated by off-the-shelf algorithms and that *Cond* is decidable and can be efficiently implemented by a *checker* algorithm.

3.1 Potential constraints

To begin with, consider *potential constraints*. These are constraints of the form $x - y + c \geq 0$. Deciding the infeasibility of conjunctions of such constraints amounts to finding a cycle of negative weight in a graph such that a edge $x \xrightarrow{c} y$ corresponds to a constraint $x - y + c \geq 0$ [1, 21][1].

Theorem 1

$$\exists \pi \in \mathit{Path}, \bigwedge \begin{pmatrix} \mathit{isCycle}(\pi) \\ \mathit{weight}(\pi) < 0 \\ \pi \subseteq (\bigcup_{i=1}^{k} \{x_{i_1} \xrightarrow{c_i} x_{i_2}\}) \end{pmatrix} \Rightarrow \forall x_1, \ldots, x_n, \neg(\bigwedge_{i=1}^{k} x_{i_1} - x_{i_2} + c_i \geq 0)$$

Proof. Ad absurdum, we suppose that we have $\bigwedge_{i=1}^{k} x_{i_1} - x_{i_2} + c_i \geq 0$ for some $x_1, \ldots, x_n$. If we sum the constraints over a cycle π, variables cancel and the result is the total weight of the path $\sum_{x \xrightarrow{c} y \in \pi} x - y + c = \sum_{x \xrightarrow{c} y \in \pi} c$. Moreover, by hypothesis, we also have that $\left(\sum_{x \xrightarrow{c} y \in \pi} x - y + c\right) \geq 0$ (each element of the sum being positive). We conclude that the total weight of cycles is necessarily positive. It follows that the existence of a cycle of negative weight c yields a contradiction. □

As a result, a negative cycle is a certificate of infeasibility of a conjunction of potential constraints.

Bellmann-Ford shortest path algorithm is a certificate generator which runs in complexity $O(n \times k)$ where n is the number of nodes (or variables) and k is the number of edges (or constraints). However, this algorithm does not find the best certificate *i.e.,* the negative cycle of shortest length. Certificates, *i.e.,* graph cycles, can be coded by a list of binary indexes – each of them identifying one of the k constraints. The worst-case certificate is then a Hamiltonian circuit which is a permutation of the k constraint indexes. Its asymptotic size is therefore $k \times \mathit{log}(k)$:

$$\mathit{size}(i_1, \ldots, i_k) = \sum_{j=1}^{k} \mathit{log}(i_j) = \sum_{j=1}^{k} \mathit{log}(j) = \mathit{log}(\Pi_{j=1}^{k} j) = \mathit{log}(k!) \sim k \times \mathit{log}(k)$$

Verifying a certificate consists in checking that:

1. indexes are bound to genuine expressions;
2. verify that expressions form a cycle;
3. compute the total weight of the cycle and check its negativity

This can be implemented in time linear in the size of the certificate.

[1] As shown by Shostak [24], this graph-based approach generalises to constraints of the form $a \times x - b \times y + c \geq 0$.

3.2 Linear constraints

The linear fragment of arithmetics might be the most widely used. It consists of formulae built over the following expressions:

$$\mathit{Expr} ::= c_1 \times x_1 + \ldots + c_n \times x_n + c_{n+1}$$

A well-known result of linear programming is Farkas's Lemma which states a strong duality result.

Lemma 1 (Farkas's Lemma (Variant)) *Let $A : \mathbb{Q}^{m \times n}$ be a rational-valued matrix and $b : \mathbb{Q}^m$ be a rational-valued vector. Exactly one of the following statement holds:*

– $\exists (y \in \mathbb{Q}^n), y \geq \bar{0}, b^t \cdot y > 0, A^t \cdot y = \bar{0}$
– $\exists (x \in \mathbb{Q}^m), A \cdot x \geq b$

Over $\mathbb{Z}$, Farkas's Lemma is sufficient to provide infeasibility certificates for systems of inequalities.

Lemma 2 (Weakened Farkas's Lemma (over $\mathbb{Z}$)) *Let $A : \mathbb{Z}^{m \times n}$ be a integer-valued matrix and $b : \mathbb{Z}^m$ be a integer-valued vector.*

$$\exists (y \in \mathbb{Z}^n), y \geq \bar{0}, b^t \cdot y > 0, A^t \cdot y = \bar{0} \Rightarrow \forall (x \in \mathbb{Z}^n), \neg A \cdot x \geq b$$

Proof. Ad absurdum, we suppose that we have $A \cdot x \geq b$ for some vector x. Since y is a positive vector, we have that $(A \cdot x)^t \cdot y \geq b^t \cdot y$. However, $(A \cdot x)^t \cdot y = (x^t \cdot A^t) \cdot y = x^t \cdot (A^t \cdot y)$. Because $A^t \cdot y = 0$, we conclude that $0 \geq y^t \cdot b$ which contradicts the hypothesis stating that $y^t \cdot b$ is strictly positive. $\square$

Over $\mathbb{Z}$, Farkas's lemma is not complete. Incompleteness is a consequence of the discreetness of $\mathbb{Z}$: there are systems that have solutions over $\mathbb{Q}$ but not over $\mathbb{Z}$. A canonical example is the equation $2.x = 1$. The unique solution is the rational $1/2$ which obviously is not an integer. Yet, the loss of completeness is balanced by a gain in efficiency. Whereas deciding infeasibility of system of integer constraints is NP-complete; the same problem can be solved over the rationals in polynomial time.

Indeed, an infeasibility certificate is produced as the solution of the *linear program*

$$min\{y^t \cdot \bar{1} \mid y \geq \bar{0}, b^t \cdot y > 0, A^t \cdot y = \bar{0}\}$$

Note that linear programming also *optimises* the certificate. To get *small* certificates, we propose to minimise the sum of the elements of the solution vector.

Linear programs can be solved in polynomial time using interior point methods [17]. The Simplex method – despite its worst-case exponential complexity – is nonetheless a practical competitive choice.

Linear programs are efficiently solved over the rationals. Nonetheless, an integer certificate can be obtained from any rational certificate.

Proposition 1 (Integer certificate) *For any rational certificate of the form* $cert_{\mathbb{Q}} = [p_1/q_1; \ldots; p_k/q_k]$, *an integer certificate is*

$$cert_{\mathbb{Z}} = [p'_1; \ldots; p'_k]$$

where $p'_i = p_i \times lcm/q_i$ *and* lcm *is the least common multiple of the* $q_i s$.

Worst-case estimates of the size of the certificates are inherited from the theory of integer and linear programming (see for instance [23]).

Theorem 2 (from [23] Corollary 10.2a) *The bit size of the rational solution of a linear program is at most* $4d^2(d+1)(\sigma+1)$ *where*

- *d is the dimension of the problem;*
- *σ is the number of bits of the biggest coefficient of the linear program.*

Using Lemma 1 and Theorem 2, the next Corollary gives a coarse upper-bound of the bit size of integer certificates.

Corollary 1 (Bit size of integer certificates) *The bit size of integer certificates is bounded by* $4k^3(k+1)(\sigma+1)$

Proof. Let $cert_{\mathbb{Z}} = [p'_1; \ldots; p'_k]$ be the certificate obtained from a rational certificate $cert_{\mathbb{Q}} = [p_1/q_1; \ldots; p_k/q_k]$.

$$\begin{aligned} \mid cert_{\mathbb{Z}} \mid &= \textstyle\sum_{i=1}^{k} log(p'_i) \\ &= \textstyle\sum_{i=1}^{k} (log(p_i) - log(q_i)) + \sum_{i=1}^{k} log(lcm) \\ &= \textstyle\sum_{i=1}^{k} (log(p_i) - log(q_i)) + k \times log(lcm) \end{aligned}$$

At worse, the q_is are relatively prime and $lcm = \Pi_{i=1}^{n} q_i$.

$$\mid cert_{\mathbb{Z}} \mid \leq k \times \sum_{i=1}^{k} log(q_i) + \sum_{i=1}^{k} (log(p_i) - log(q_i))$$

As $\mid cert_{\mathbb{Q}} \mid = \sum_{i=1}^{k} log(p_i) + log(q_i)$, we have that $\mid cert_{\mathbb{Z}} \mid \leq k \times \mid cert_{\mathbb{Q}} \mid$. By Theorem 2, we conclude the proof and obtain the $4k^3(k+1)(\sigma+1)$ bound. □

Optimising certificates over the rationals is reasonable. Rational certificates are produced in polynomial time. Moreover, the worst-case size of the integer certificates is kept reasonable.

Checking a certificate *cert* amounts to

1. checking the positiveness of the integers in *cert*;
2. computing the matrix-vector product $A^t \cdot cert$ and verifying that the result is the null vector;
3. computing the scalar product $b^t \cdot cert$ and verifying its strict positivity

Overall, this leads to a quadratic-time $O(n \times k)$ checker in the number of arithmetic operations.

3.3 Polynomial constraints

For our last fragment, we consider unrestricted expressions built over variables, integer constants, addition and multiplication.

$$e \in \mathit{Expr} ::= x \mid c \mid e_1 + e_2 \mid e_1 \times e_2$$

As it reduces to solving diophantine equations, the logical fragment we consider is not decidable over the integers. However, it is a result by Tarski [26] that the first order logic $\langle \mathbb{R}, +, *, 0 \rangle$ is decidable. In the previous section, by lifting our problem over the rationals, we traded incompleteness for efficiency. Here, we trade incompleteness for decidability.

In 1974, Stengle generalises Hilbert's *nullstellenstaz* to systems of polynomial inequalities [25]. As a matter of fact, this provides a *positivstellensatz*, *i.e.,* a theorem of positivity, which states a necessary and sufficient condition for the existence of a solution to systems of polynomial inequalities. Over the integers, unlike Farkas's lemma, Stengle's *positivstellensatz* yields sound infeasibility certificates for conjunctions of polynomial inequalities.

Definition 1 (Cone) *Let $P \subseteq \mathbb{Z}[\bar{x}]$ be a finite set of polynomials. The cone of P ($\mathit{Cone}(P)$) is the smallest set such that*

1. $\forall p \in P, p \in \mathit{Cone}(P)$
2. $\forall p_1, p_2 \in \mathit{Cone}(P), p_1 + p_2 \in \mathit{Cone}(P)$
3. $\forall p_1, p_2 \in \mathit{Cone}(P), p_1 \times p_2 \in \mathit{Cone}(P)$
4. $\forall p \in \mathbb{Z}[\bar{x}], p^2 \in \mathit{Cone}(P)$

Theorem 3 states sufficient conditions for infeasibility certificates:

Theorem 3 (Weakened Positivstellensatz) *Let $P \subseteq \mathbb{Z}[x_1, \dots, x_n]$ be a finite set of polynomials.*

$$\exists \mathit{cert} \in \mathit{Cone}(P), \mathit{cert} \equiv -1 \Rightarrow \forall x_1, \dots, x_n, \neg \bigwedge_{p \in P} p(x_1, \dots, x_n) \geq 0$$

Proof. By adbsurdum, we suppose that we have $\bigwedge_{p \in P} p(x_1, \dots, x_n) \geq 0$ for some $x_1, \dots, x_n$. By routine induction over the definition of a *Cone*, we prove that any polynomial $p \in \mathit{Cone}(P)$ is such that $p(x_1, \dots, x_n)$ is positive. This contradicts the existence of the polynomial *cert* which uniformly evaluates to -1. □

Certificate generators explore the cone to pick a certificate. Stengle's result [25] shows that only a restricted (though infinite) part of the cone needs to be considered. A certificate *cert* can be decomposed into a finite sum of products of the following form:

$$\mathit{cert} \in \sum_{s \in 2^P} \left(q_s \times \prod_{p \in s} p \right)$$

where $q_s = p_1^2 + \ldots + p_i^2$ is a *sum of squares* polynomial.

As pointed out by Parrilo [20], a layered certificate search can be carried out by increasing the formal degree of the certificate. For a given degree, finding a certificate amounts to finding polynomials (of known degree) that are sums of squares. This is a problem that can be solved efficiently (in polynomial time) by recasting it as a *semidefinite program* [27]. The key insight is that a polynomial q is a *sum of square* if and only if it can be written as

$$q = \begin{pmatrix} m_1 \\ \dots \\ m_n \end{pmatrix}^t \cdot Q \cdot \begin{pmatrix} m_1 \\ \dots \\ m_n \end{pmatrix}$$

for some positive semidefinite matrix Q and some vector $(m_1, \dots, m_n)$ of linearly independent monomials.

An infeasibility certificates is a polynomial which belongs to the cone and is equivalent to -1. Using a suitable encoding, cone membership can be tested in linear time. Equivalence with -1 can be checked by putting the polynomial in Horner's normal form.

4 Implementation in the Coq proof-assistant

In this part, we present the design of the Coq reflexive tactics `micromega`[2]. This tactics solves linear and non-linear goals using the certificates and certificate generators described in Section 3. For the linear case, experiments show that `micromega` outperforms the existing Coq `(r)omega` tactics both in term of proof-term size and checking time.

4.1 Encoding of formulae

As already mentioned in section 2.1, to set up a reflection proof, logical sentences are encoded into syntactic terms. Arithmetic expressions are represented by the following inductive type:

```
Inductive Expr : Set :=
  | V      (v:Var)
  | C      (c:Z)
  | Mult   (e1:Expr) (e2:Expr)
  | Add    (e1:Expr) (e2:Expr)
  | UMinus (e:Expr).
```

The `eval_expr` function maps syntactic expressions to arithmetic expressions. It is defined by structural induction over the structure of `Expr`.

```
Fixpoint eval_expr (env:Env) (p:Expr) {struct p}: Z :=
 match p with
  | V v       ⇒ get_env env v
```

[2] `micromega` is available at http://www.irisa.fr/lande/fbesson.html.

```
  | C c       ⇒ c
  | Mult p q  ⇒ (eval_expr env p) * (eval_expr env q)
  | Add p q   ⇒ (eval_expr env p) + (eval_expr env q)
  | UMinus p  ⇒ - (eval_expr env p)
 end.
```

The environment binds variable identifiers to their integer value. For efficiency, variables identifiers are binary indexes and environments are binary trees. As a result, the function `eval_expr` runs in time linear in the size of the input expression.

Formulae are lists of expressions `Formulae := list Expr` and are equipped with an evaluation function `eval : Env→Formulae→Prop`

```
Fixpoint eval (env:Env)(f:Formulae){struct f}: Prop :=
 match f with
  | nil   ⇒ False
  | e::rf ⇒ ((eval_expr env e) ≥ 0)→(eval env rf)
 end.
```

The `eval` function generates formulae of the form $e_1(x_1,\ldots,x_n) \geq 0 \to \ldots \to e_k(x_1,\ldots,x_n) \geq 0 \to \mathit{False}$. By simple propositional reasoning, such a formula is equivalent to $\neg\left(\bigwedge_{i=1}^{k} e_i(x_1,\ldots,x_n) \geq 0\right)$ which is exactly the logical fragment studied in Section 3.

4.2 Proving the infeasibility criterion

At the core of our tactics are theorems which are reducing infeasibility of formulae to the existence of certificates. In the following, we present our formalisation of Stengle's Positivstellensatz in Coq. The cone of a set of polynomials is defined by an inductive predicate:

```
Inductive Cone (P: list Expr) : Expr → Prop :=
 |IsGen    :∀ p, In p P→Cone P p
 |IsSquare:∀ p, Cone P (Power p 2)
 |IsMult   :∀ p q, Cone P p→Cone P q→Cone P (Mult p q)
 |IsAdd    :∀ p q, Cone P p→Cone P q→Cone P (Add p q)
 |IsPos    :∀ c, c ≥ 0→Cone P (C c).
```

The fifth rule `IsPos` is redundant and absent from the formal definition of a cone (Definition 1). Indeed, any positive integer can be decomposed into a sum of square. It is added for convenience and to allow a simpler and faster decoding of certificates.

We are then able to state (and prove) our weakened *positivstellensatz.*

```
Theorem positivstellensatz : ∀ (f:Formulae),
  (∃ (e:Expr),
        Cone f e ∧
        (∀ env', eval_expr env' e = -1)) →
  ∀ env, eval env f.
```

4.3 Checking certificates

Given a certificate *cert*, we need an algorithm to check that

1. the certificate belongs to the cone `Cone(f,cert)`;
2. the certificate always evaluate to -1;

If certificates were terms of type `Expr`, proving cone membership would be a tricky task. This would be complicated and inefficient to reconstruct the cone decomposition of the expression. To avoid this pitfall, by construction, our certificates always belong to the cone. To do that, the data-structure of the certificates mimics the definition of the cone predicate:

```
Inductive Certificate : Set :=
  | Cert_IsGen    (n:nat)
  | Cert_IsSquare (e:Expr)
  | Cert_IsMult   (e1:Expr) (e2:Expr)
  | Cert_IsAdd    (e1:Expr) (e2:Expr)
  | Cert_IsZpos   (p:positive)
  | Cert_IsZ0.
```

Given the generators of a cone, *i.e.,* a list of expressions, a certificate is decoded into an expression:

```
Fixpoint decode
 (P: list Expr) (c: Certificate) {struct c} : Expr :=
 match c with
 |Cert_IsGen n     ⇒ nth n P (C 0)
 |Cert_IsSquare p ⇒ Mult p p
 |Cert_IsMult p q ⇒ Mult (decode P p)(decode P q)
 |Cert_Add p q     ⇒ Add (decode P p)(decode P q)
 |Cert_IsZpos p    ⇒ C (Zpos p)
 |Cert_IsZ0        ⇒ C Z0
 end.
```

This construction ensures that certificates are always mapped to expressions that belong to the cone as stated by the following lemma.

```
Lemma cert_in_cone : ∀ P cert, Cone P (decode l cert).
```

Because our certificate encoding ensures cone membership, it remains to test that a polynomial always evaluates to a negative constant. To do that, we reuse the algorithm developed by the Coq `ring` tactics which normalises polynomial expressions. In the end, our checker is implemented by the following algorithm:

```
Let checker (c:Certificate) (P: list Expr) : bool :=
    (polynomial_simplify (decode P c)) == -1
```

4.4 Certificate generation

In Section 3, we shed the light on three different arithmetic fragments, namely, potential constraints (Section 3.1), linear constraints (Section 3.2) and polynomial constraints (Section 3.3). Obviously, these logic fragments form a strict hierarchy: polynomial constraints subsume linear and potential constraints. It appears that this hierarchy is also apparent at the level of certificates: both negative-weighted cycles and Farkas's Lemma certificates can be interpreted as *Positivstellensatz* certificates.

For linear goals, our certificates are produced by a handcrafted linear solver. For non-linear goals, we are using the full-fledged semidefinite programming solver Csdp [5] through its HOL Light interface [15]. Anyhow, whatever their origin, certificates are translated into *Positivstellensatz* certificates.

4.5 Experiments

We have assessed the efficiency of `micromega` with respect to the existing Coq tactic `romega`[3]. As mentioned earlier, the `romega` is a reflexive tactics which solves linear goals by checking traces obtained from an instrumented version of the Omega test. Our benchmarks are the smallest SMALLINT problems of the Pseudo Boolean Evaluation 2005/2006 contest[4]. The number of variables is ranging from 220 to 2800 while the number of constraints is ranging from 42 to 160. The benchmarks are run on a 2.4 Ghz Intel Xeon desktop with 4GB of memory. The graph of Figure 1 presents the running time of the Coq type-checking of the certificates generated by `romega` ■ and `micromega` ▲. For this

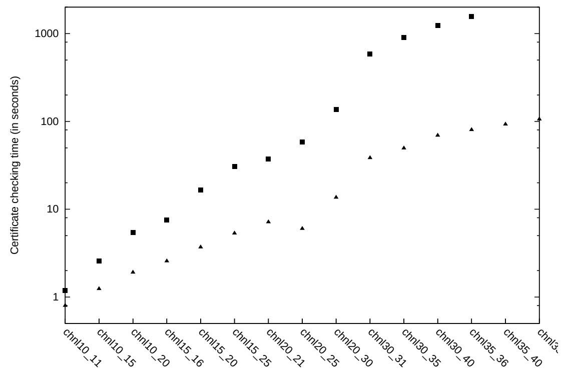

Fig. 1. Evolution of the type-checking time

experiment, the certificates produced by `micromega` are always faster to check.

[3] `romega` already outperforms the `omega` tactics
[4] http://www.cril.univ-artois.fr/PB06

Moreover, `micromega` scales far better than `romega`. It is also worth noting than `romega` fails to complete the last two benchmarks. For the last benchmark, the origin of the failure is not fully elucidated. For the penultimate one, a stack-overflow exception is thrown while type-checking the certificate.

Figure 2 plots the size of compiled proof-terms (.vo files) generated by `romega` ■ and `micromega` ▲ together with the textual size of the problems •. For small

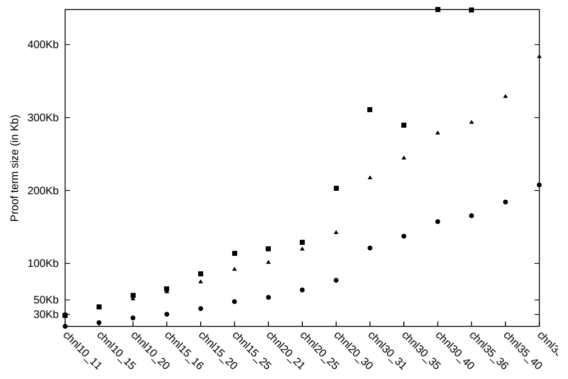

Fig. 2. Evolution of the size of proof-terms

instances, the two tactics generate proof-terms of similar size. For big instances, proof-terms of generated by `micromega` are smaller than those produced by `romega`. Moreover, their size is more closely correlated to the problem size.

On the negative side, both tactics are using a huge amount of memory. (For the biggest problems, the memory skyrockets up to 2.5 GB.) Further investigations is needed to fully understand this behaviour.

5 Related work and conclusion

In this paper, we have identified logical fragments for which certificate generators are off-the-shelf decision procedures (or algorithms) and reflexive certificate checkers are proved correct inside the proof-assistant. Using the same approach, Grégoire *et al.,* [14] check Pocklington certificates to get efficient reflexive Coq proofs that a number is prime. In both cases, the checkers benefit from the performance of the novel Coq virtual machine [12].

For Isabelle/HOL, recent works attest the efficiency of reflexive approaches. Chaieb and Nipkow have proved correct Cooper's decision procedure for Presburger arithmetics [7]. To obtain fast reflexive proofs, the HOL program is compiled into ML code and run inside the HOL kernel. Most related to ours is the work by Obua [19] which is using a reflexive checker to verify certificates generated by the Simplex. Our work extends this approach by considering more general certificates, namely *positivstellensatz* certificates.

To prove non-linear goals, Harrison mentions (chapter 9.2 of the HOL Light tutorial [15]) his use of semidefinite programming. The difference with our approach is that the HOL Light checker needs not to be proved correct but is a Caml program of type `cert` → `term` → `thm`. Dynamically, the HOL Light kernel ensures that theorems can only be constructed using sound logical inferences.

As decision procedures get more and more sophisticated and fine-tuned, the need for trustworthy checkers has surged. For instance, the state-of-the-art zChaff SAT solver is now generating proof traces [29]. When proof traces exist, experiments show that they can be efficiently rerun inside a proof-assistant. Using Isabelle/HOL, Weber [28] reruns zChaff traces to solve problems that Isabelle/HOL decision procedure could not cope with. Fontaine *et al.,* [11] are using a similar approach to solve quantifier-free formulae with uninterpreted symbols by rerunning proof traces generated by the Harvey SMT prover [10].

In Proof Carrying Code [18], a piece of code is downloaded packed with a checkable certificate – a proof accessing that it is not malicious. Certificate generation is done ahead-of-time while certificate checking is done at download time. Previous work has shown how to bootstrap a PCC infrastructure using a general-purpose proof-assistant like Coq [6, 3, 4]. In this context, the triples (certificate,checkers,prover) defined here could be used to efficiently check arithmetic verification conditions arising from the analysis of programs.

Acknowledgements Thanks are due to the anonymous referees for putting this work into perspective and pointing out relevant related work.

References

1. R. Bellman. On a routing problem. In *Quarterly of Applied Mathematics*, volume 16, pages 87–90, 1958.
2. Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions.* Springer, 2004.
3. F. Besson, T. Jensen, and D. Pichardie. A PCC Architecture based on Certified Abstract Interpretation. In *Proc. of 1st Int. Workshop on Emerging Applications of Abstract Interpretation*, ENTCS. Springer-Verlag, 2006.
4. F. Besson, T. Jensen, and D. Pichardie. Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. *Theoretical Computer Science*, 364:273–291, 2006.
5. B. Borchers. Csdp, 2.3 user's guide. *Optimization Methods and Software*, 11(2):597–611, 1999.
6. D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. *Theor. Comput. Sci.*, 342(1):56–78, 2005.
7. A. Chaieb and T. Nipkow. Verifying and reflecting quantifier elimination for presburger arithmetic. In *Proc. of 12th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3835 of *LNAI*, pages 367–380. Springer, 2005.
8. P. Crégut. Une procédure de décision réflexive pour un fragment de l'arithmétique de presburger. In *Journées Francophones des Langages Applicatifs*, 2004.
9. The Coq development team. The coq proof assistant - reference manual v 8.1.

10. D. Déharbe and S. Ranise. Light-weight theorem proving for debugging and verifying units of code. In *1st IEEE Int. Conf. on Software Engineering and Formal Methods*. IEEE Computer Society, 2003.
11. P. Fontaine, J-Y. Marion, S. Merz, L. Nieto, and A. Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In *12th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *LNCS*, pages 167–181. Springer-Verlag, 2006.
12. B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *In Proc. of the 7th Int. Conf. on Functional Programming*, pages 235–246. ACM Press, 2002.
13. B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in coq. In *Proc. of the 18th Int. Conf. on Theorem Proving in Higher Order Logics*, volume 3603 of *LNCS*, pages 98–113. Springer, 2005.
14. B. Grégoire, L. Théry, and B. Werner. A computational approach to pocklington certificates in type theory. In *Proc. of the 8th Int. Symp. on Functional and Logic Programming*, volume 3945 of *LNCS*, pages 97–113. Springer, 2006.
15. J. Harrison. HOL light tutorial (for version 2.20).
16. J. Harrison and L. Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.
17. N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proc. of the 16th ACM Symp. on Theory of Computing*, pages 302–311. ACM Press, 1984.
18. G. Necula. Proof-carrying code. In *Proc. of the 24th ACM Symp. on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.
19. S. Obua. Proving bounds for real linear programs in isabelle/hol. In *Proc. of the 18th Int. Conf. on Theorem Proving in Higher Order Logics*, volume 3603 of *LNCS*, pages 227–244. Springer, 2005.
20. P. A. Parrilo. Semidefinite programming relaxations for semialgebraic problems. *Math. Program.*, 96(2):293–320, 2003.
21. V. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, 1977.
22. W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proc. of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM Press, 1991.
23. A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1998.
24. R. Shostak. Deciding linear inequalities by computing loop residues. *J. ACM*, 28(4):769–779, 1981.
25. G. Stengle. A nullstellensatz and a positivstellensatz in semialgebraic geometry. *Mathematische Annalen*, 207(2):87–97, 1973.
26. A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, 2ed edition, 1951.
27. L. Vandenberghe and S. Boyd. Semidefinite programming. *SIAM Rev.*, 38(1):49–95, 1996.
28. T. Weber. Using a SAT solver as a fast decision procedure for propositional logic in an LCF-style theorem prover. In *Proc. of 18th Int. Conf. on the Theorem Proving in Higher Order Logics*, pages 180–189, August 2005.
29. L. Zhang and S. Malik. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe*, pages 10880–10885. IEEE Computer Society, 2003.