Deciding Presburger arithmetic using reflection

Master's internship under T. Altenkirch supervision
University of Nottingham

G. Allais

August 30, 2010

Abstract

The need to prove or disprove a formula of Presburger arithmetic is quite frequent in certified software development (constraints generated automatically) or when working on higher arithmetic (number theory). The fact that this theory is decidable and that Agda is now mature enough to be able to implement such a solver pushed us to try to tackle this problem.

The numerous steps of Cooper's decision procedure makes it easy to introduce bugs in an unverified implementation. We will show how the use of reflection together with dependent types can allow us to develop a bug-free solver that is proved to be complete.

In the following developments, k, m, n will refer to integers, x, y, z to variables ranging over the integers, P, Q, R to quantifier free propositions and Φ, Ψ to any formula of Presburger arithmetic.

Part I
Deciding Presburger arithmetic

1 Preliminaries

1.1 Definitions

The main concern of this paper is to be able to deal automatically with formulas of an extension of Presburger arithmetic without free variable. The first-order theory on natural numbers with addition but without multiplication (Presburger arithmetic) is extended with the use of the canonical order on the integers and a couple notions that are only syntactic sugar on top of it:

- We allow the use of multiplication when the multiplicand is an integer: $k * e$ is only a shortcut for $\underbrace{sign(k).e + \cdots + sign(k).e}_{|k|\ times}$;
- We also allow the use of $k\ div\ e$ which is just a shortcut for $\exists x, e = k * x$.

The formulas Φ of Presburger arithmetic can be generated by the following grammar:

$$\begin{array}{lcl} e & ::= & k \mid x \mid k * e \mid e + e \\ \\ \Phi & ::= & \top \mid \bot \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \forall.\Phi \mid \exists.\Phi \mid \neg\Phi \mid \Phi \rightarrow \Phi \mid \\ & & e = e \mid e < e \mid e \leq e \mid e > e \mid e \geq e \mid k \; div \; e \end{array}$$

Elements of e will be called expressions while elements of Φ will be referred to as formulas.

1.2 What has already been done.

- 1929: Presburger introduces his arithmetic without multiplication

  It is proved to be coherent and decidable (unlike Peano arithmetic that does use multiplication and that is not decidable).

- 1972: Cooper's "theorem proving in arithmetic without multiplication"

  In this paper Cooper fully describes a decision procedure that is based on quantifier elimination. The results are refined a couple of years later, limiting the formula's size's blow-up.

- 1974: Fischer & Rabin's super-exponential complexity of PA [5]

  Even if Presburger arithmetic is known to be really hard to decide (super-exponential complexity), studies have shown that quite a lot of instances are solvable in a reasonable amount of time which is a good claim in favor of a solver that the user could invoke.

- 2001: CALIFE: ROmega (Universally quantified PA formulas)

  (Re-)implementation of the Omega tactic in Coq using reflection. It deals only with universally quantified formulas.

- 2005–08: Nipkow's quantifier elimination for PA (HOL)

  The second paper corrects a couple of crucial mistakes of the first one especially about the definition of the equivalent proposition when x tends to be very small.

1.3 Principle

This paper is mainly based on Tobias Nipkow's work [1, 2]. It uses Cooper's decision procedure which is based on a quantifier elimination mechanism [3]. The quantifier elimination is performed using an elimination set: a finite set from which one can generate a finite disjunction of quantifier free formulas that is equivalent to the original statement.

As the equality, the divisibility and the canonical order on the integers are obviously decidable, if we are able to perform quantifier elimination we can decide Presburger arithmetic. We, therefore, need to implement a procedure $elim$ that, given a quantifier free proposition P, outputs a proposition $elim(P)$ such that:

$$\exists.P \Leftrightarrow elim(P)$$

From this simple function and the fact that formulas without quantifier and without free variables are decidable, we can produce a fully constructive function $elim_\Phi$ that, given any formula of Presburger arithmetic, will output a quantifier free equivalent. We proceed by eliminating the innermost quantifier and assimilating $\forall.P$ to $\neg(\exists.\neg P)$ (which is true because P is quantifier free thus decidable).

This elimination process has two main parts: the first part is the normalisation of the input formulas (and the proof that this normalisation preserves provability) ; the second part is the proof of cooper's theorem which is roughly saying that we can perform quantifier elimination on the normalised formulas.

2 Normalisation

The purpose of the normalisation step is to output a simpler formula which is equivalent to the given one. The manipulation of the formulas during the proof of Cooper's theorem will be easier because we will have fewer cases to look at. This representation will be reached through various steps, each one of them preserving provability.

2.1 Negation normal form

Our first concern will be to push the negations inward so that we end up with negations only in front of equalities or divisibility statements. This is also the occasion to get rid off implication and the variety of statements involving $<$, $\geq$ and $>$ by using only $\leq$.

Definition The set of formulas under **negation normal form** is generated by the following grammar:

$$\begin{array}{rcl} P & ::= & \top \mid \bot \mid P \wedge P \mid P \vee P \mid \\ & & e = e \mid e \neq e \mid e \leq e \mid k \ div \ e \mid \neg(k \ div \ e) \end{array}$$

We can note that these formulas are obviously quantifier-free.

2.2 Linearisation

The second step is (now that the formulas' shape is standardized) to find a normal form for the expressions. As variables are referred to with de Bruijn indices and as there is no multiplication of variables, we can transform the given expression into a linear one where the variables are sorted by their index.

Definition A **linear expression** is an expression such that:

- each variable name is used at most once
- each coefficient is nonzero
- variables are sorted by their de Bruijn index

Definition The set of **linear** formulas is defined by the following grammar:

$$\begin{array}{rcl} P & ::= & \top \mid \bot \mid P \wedge P \mid P \vee P \mid \\ & & e = 0 \mid e \neq 0 \mid e \leq 0 \mid k \ div \ e \mid \neg(k \ div \ e) \end{array}$$

where the expressions e are linear and the ks are nonzero.

Lemma 2.1. *For every formula P in negative normal form, there exists a linear formula P' such that P is equivalent to P'.*

Proof. The definition of the formulas' linearity has two main parts: the first one is dealing with the formulas shape while the second one is focusing on the expressions' one. Proving that we can modify a formula's shape in order to fit the constraints is not really hard whereas the work on the construction of linear expressions is a bit more tedious.

- Expressions only on the left hand side

  Let C be an element of the set $\{=, \neq, \leq\}$. It is obvious that $e_1 \ C \ e_2$ is equivalent to $e_1 - e_2 \ C \ 0$. Once $e_1 - e_2$ is linearised, so is the formula.

- Nonzero divisors

  For all expressions of the shape $k \ div \ e$, if k is zero, then it is equivalent to $e = 0$. The linearisation of e ends the process.

- Linearisation of expressions

  The proof is done by structural induction on the expression e. We will focus only on the case where $e = e_1 + e_2$ as the other ones ($e = k * e_1$, $e = -e_1$) are pretty similar. We know by induction that there exists two linear expressions e_1^l and e_2^l such that $e_1 = e_1^l$ and $e_2 = e_2^l$. The only thing left to prove is that there exists a procedure that, given two linear expressions, outputs a linear expression that is the sum of the two inputs. Lets proceed by structural induction on (e_1^l, e_2^l).

  Here is a quick overview of the proof:

$\begin{pmatrix} e_1^l \\ e_2^l \end{pmatrix}$	$i_0 \overset{?}{\leq} j_0$	const.	+ind. hyp.
$\begin{pmatrix} n_1 \\ n_2 \end{pmatrix}$		$n_1 + n_2$	+ -
$\begin{pmatrix} c_{1,i_0} * x_{i_0} + r \\ n_2 \end{pmatrix}$		$c_{1,i_0} * x_{i_0}$	+ $r + n_2$
$\begin{pmatrix} n_1 \\ c_{2,j_0} * x_{j_0} + r \end{pmatrix}$		$c_{2,j_0} * x_{j_0}$	+ $r + n_1$
	$i_0 < j_0$	$c_{1,i_0} * x_{i_0}$	+ $r_1 + e_2^l$
$\begin{pmatrix} c_{1,i_0} * x_{i_0} + r_1 \\ c_{2,j_0} * x_{j_0} + r_2 \end{pmatrix}$	$i_0 = j_0$	$(c_{1,i_0} + c_{2,j_0}) * x_{i_0}$	+ $r_1 + r_2$
	$i_0 > j_0$	$c_{2,j_0} * x_{j_0}$	+ $e_1^l + r_2$

  – If both e_1^l and e_2^l are values, $e_1^l + e_2^l$ is the linear expression we are looking for.

- If $e_1^l = c_{1,i_0} * x_{i_0} + r_1$ and e_2^l is a value, let r^l be the linear expression obtained by induction hypothesis on r_1 and e_2^l. The linear expression: $c_{1,i_0} * x_{i_0} + r^l = c_{1,i_0} * x_{i_0} + (r_1 + e_2^l) = e_1^l + e_2^l$ will do the job.
- If e_1^l is a value and $e_2^l = c_{1,j_0} * x_{j_0} + r_2$, it is a similar problem (e_1 and e_2 have symmetric roles).
- If $e_1^l = c_{1,i_0} * x_{i_0} + r_1$ and $e_2^l = c_{2,j_0} * x_{j_0} + r_2$, we compare i_0 and j_0. There are three cases:

  1. If i_0 is smaller than j_0 then $c_{1,i_0} * x_{i_0} + r^l$ (where r^l is obtained by induction hypothesis on r_1 and $c_{2,j_0} * x_{j_0} + r_2$) is the linear expression we are looking for.
     The linearity comes from the fact that, given that i_0 is smaller than j_0, it is at the same time smaller than all the is in r_1 and all the js in $c_{2,j_0} * x_{j_0} + r_2$ and therefore smaller than all the indices used in the linearised equivalent of $r_1 + c_{2,j_0} * x_{j_0} + r_2$.
  2. If i_0 equals j_0 then we compute $r^l = r_1 + r_2$ by induction hypothesis and we have to distinguish two cases:
     Either $c_{1,i_0} + c_{2,j_0} = 0$**:** we now have r^l which is obviously linear and equal to $e_1^l + e_2^l$.
     Or $c_{1,i_0} + c_{2,j_0} \neq 0$**:** the expression $(c_{1,i_0} + c_{2,j_0}) * x_{i_0} + r^l$ is linear (r^l uses only indices that are greater than $i_0 = j_0$) and is equal to $c_{1,i_0} * x_{i_0} + c_{2,j_0} * x_{j_0} + r_1 + r_2 = e_1^l + e_2^l$.
  3. If i_0 is greater than j_0 then $c_{2,j_0} * x_{j_0} + r^l$ (where r^l is obtained by induction hypothesis on $c_{1,i_0} * x_{i_0} + r_1$ and r_2) is linear and equal to $e_1^l + e_2^l$. This case is similar to the one where i_0 is smaller than j_0 because e_1^l and e_2^l have symmetric roles.

□

This representation allows us to see very easily if the current expression mentions the variable bounded by the innermost quantifier (which is the one we want to eliminate):

$$e = c_{i_0} * v_{i_0} + c_{i_1} * v_{i_1} + \cdots + c_{i_k} * v_{i_k} + n$$

where $0 \leq i_0 < i_1 < \cdots < i_k$. We can note that a linear formula is obviously in negation normal form.

2.3 Unitarization

Definition We call "**unitarized**" an expression e such that:

- e is linear
- if x_0 appears in e, then its coefficient is either 1 or -1

Definition A **unitarized formula** is a linear formula such that every expression is a unitarized expression.

Definition The lcm_P of a linear formula P is the least common multiple of all the x_0's coefficients appearing in P. It is defined by structural induction on P:

P	lcm_P
$k * x_0 + r = 0$	$\|k\|$
$k * x_0 + r \neq 0$	$\|k\|$
$k * x_0 + r \leq 0$	$\|k\|$
$k\ div\ (k' * x_0 + r)$	$\|k'\|$
$\neg(k\ div\ (k' * x_0 + r))$	$\|k'\|$
$P_1 \wedge P_2$	$lcm(lcm_{P_1})(lcm_{P_2})$
$P_1 \vee P_2$	$lcm(lcm_{P_1})(lcm_{P_2})$
-	1

Definition The **unitarization** of a formula is the process that, given a linear formula P, outputs a unitarized formula P_u that is in some way equivalent (see following lemma).

Unitarization with respect to m: Let m be an integer such that all the x_0's coefficients appearing in P divides m (which obviously implies that if k is such a coefficient, then $\frac{m}{|k|}$ is an integer). The unitarization of P with respect to m is defined by structural induction on P:

- $k * x_0 + r = 0$ becomes $(sign\ k) \bullet x_0 + \frac{m}{|k|} * r = 0$
- $k * x_0 + r \neq 0$ becomes $(sign\ k) \bullet x_0 + \frac{m}{|k|} * r \neq 0$
- $k * x_0 + r \leq 0$ becomes $(sign\ k) \bullet x_0 + \frac{m}{|k|} * r \leq 0$
- $k\ div\ (k' * x_0 + r)$ becomes $(\frac{m}{|k'|} * k)\ div\ ((sign\ k') \bullet x_0 + \frac{m}{|k'|} * r)$
- $\neg(k\ div\ (k' * x_0 + r))$ becomes $\neg((\frac{m}{|k'|} * k)\ div\ ((sign\ k') \bullet x_0 + \frac{m}{|k'|} * r))$
- $P_1 \wedge P_2$ becomes $P_1' \wedge P_2'$ where P_1' and P_2' are the unitarized (with respect to m) versions of P_1 and P_2.
- $P_1 \vee P_2$ becomes $P_1' \vee P_2'$ where P_1' and P_2' are the unitarized (with respect to m) versions of P_1 and P_2.

Unitarization The unitarization we are talking about is the unitarization of P with respect to lcm_P. The unitarized version of P is named P_u.

Lemma 2.2. *Unitarization preserves provability*

$$\forall P, \forall x, P(x) \Leftrightarrow P_u(lcm_P * x)$$

Proof. Knowing that lcm_P is strictly positive, and that $\mathbb{Z}$'s integrity implies that $\forall e_1, e_2, k \in \mathbb{Z}, k \neq 0 \Rightarrow (e_1 = e_2 \Leftrightarrow k * e_1 = k * e_2)$, it is pretty straightforward to prove by structural induction on P that $\forall x, P(x) \Leftrightarrow P_u(lcm_P * x)$.

□

It is even possible to get rid of the lcm_P factor thanks to the following lemma:

Lemma 2.3.

$$\forall k, \forall P, (\exists x, P(k * x)) \Leftrightarrow (\exists x, P(x) \wedge k\ div\ x)$$

and its corollary:

Corollary 2.4. *Compatibility of unitarization and existential quantifiers*

$$\forall P, \exists x, P(x) \Leftrightarrow \exists x, P_u(x) \wedge lcm_P \ div \ x$$

2.4 Conclusions

The normalisation of the inputs reduces drastically the amount of formulas that we have to look at while preserving provability.

3 Elimination procedure

In the following section, we will only deal with unitarized formulas.

3.1 A few remarks

The idea of the elimination procedure is to note a couple of basic things:

1. there exists a subset of formulas that are periodic (almost x_0-free formulas)
2. when x_0 tends to be very small, P is equivalent to an almost x_0-free formula $P_{-\infty}$
3. there exists a set B of x_0-free expressions such that: if there is no element $b \in B$ such that $P(b)$ then there exists a $k > 0$ such that for all x, $P(x)$ implies $P(x - k)$

3.2 Almost x_0-free formulas

Definition A formula P is said to be **almost x_0-free** when it is a unitarized formula in which the $k \ div \ e$ statements are the only statements where x_0 may appear.

Definition $lcm_{div}(P)$ is the least common multiple of all the ks such that $k \ div \ e$ appears in P (where e contains x_0).

Lemma 3.1. *Almost x_0-free formulas' periodicity*
For all almost x_0-free formula P, P is $lcm_{div}(P)$-periodic:

$$\forall x, P(x) \Leftrightarrow P(x \pm lcm_{div}(P))$$

Proof. The proof is done by structural induction on P and is straightforward. As P is an almost x_0 free formula, the only interesting base case that we have to look at is the case where we have $k \ div \ (\pm x_0 + r)$. As we know (by definition) that $k \ div \ lcm_{div}(P)$, we can conclude that $k \ div \ (\pm(x_0 \pm lcm_{div}(P)) + r)$.
□

3.3 When $x_0 \to -\infty$...

Lemma 3.2. *For all unitary formula P, there exists an integer z and an almost x_0-free proposition $P_{-\infty}$ such that:*

$$\forall x, x \leq z \Rightarrow (P(x) \Leftrightarrow P_{-\infty}(x))$$

Proof.

$$\begin{array}{rcl} x_0{+}r \leq 0 & \Leftrightarrow & \top \ when \ x_0 \leq -r \\ -x_0{+}r \leq 0 & \Leftrightarrow & \bot \ when \ x_0 \leq r-1 \\ k*x_0{+}r = 0 & \Leftrightarrow & \bot \ when \ x_0 \leq -k*r-1 \\ k*x_0{+}r \neq 0 & \Leftrightarrow & \top \ when \ x_0 \leq -k*r-1 \end{array}$$

The formula that we obtain only mentions x_0 in $k \ div \ e$ statements and is unitarized; it is therefore and almost x_0-free formula that is equivalent to P for a certain z small enough.

□

This leads us to the first theorem about the existence of an elimination set for P (it deals with the case where x is very small):

Lemma 3.3. *Elimination of small xs*

For all unitary formula P, there exists an integer z and an almost x_0-free formula $P_{-\infty}$ such that:

$$(\exists x, x \leq z \wedge P(x)) \Leftrightarrow (\exists d \in [|0; lcm_{div}(P_{-\infty}) - 1|], P_{-\infty}(d))$$

Proof. The proof is the combination of the previous lemma which gives us the integer z and the proposition $P_{-\infty}$ and the other one saying that almost x_0-free formulas are periodic.

$$\begin{array}{lcll} \forall x, x \leq z \Rightarrow \quad P(x) & \underset{Lem.\ 3.2}{\iff} & P_{-\infty}(x) & \\ & \underset{Lem.\ 3.1}{\iff} & P_{-\infty}(x & (\text{mod } lcm_{div}(P_{-\infty}))) \end{array}$$

□

Corollary 3.4. *The existence of a very small x such that $P(x)$ is equivalent to a finite disjunction that is **totally** x_0**-free**:*

$$(\exists x, x \leq z \wedge P(x)) \Leftrightarrow \bigvee_{d \in [|0; lcm_{div}(P_{-\infty})-1|]} P_{-\infty}(d)$$

3.4 The B-set

The B set of a formula is (roughly) a set of values such that if $P(x)$ is provable $P(x - lcm_{div}(P))$ might not be. The definition might appear a bit magical but the constraints fall from the needs in the proof of the following lemma.

P	$B(P)$
$-x_0 + r \leq 0$	$\{r - 1\}$
$x_0 + r = 0$	$\{-r - 1\}$
$-x_0 + r = 0$	$\{r - 1\}$
$k * x_0 + r \neq 0$	$\{-k * r\}$
$P_1 \wedge P_2$	$B(P_1) \cup B(P_2)$
$P_1 \vee P_2$	$B(P_1) \cup B(P_2)$
-	$\{\}$

Remark When there is no ambiguity, we talk about the B set instead of the $B(P)$ set.

Lemma 3.5. *For all unitarized formula P,*

$$\forall x, \neg\left(\exists b \in B, \exists j \in [|0; lcm_{div}(P)|], x = b + j\right) \Rightarrow P(x) \Rightarrow P(x - lcm_{div}(P))$$

Proof. The proof is done by structural induction on P. If x_0 does not appear in P, it's trivial. Let's consider the formulas where x_0 appears.

- if $P(x)$ is $-x + r \leq 0$ then $P(x - lcm_{div}(P))$ is $(-x + r) + lcm_{div}(P) \leq 0$ and

  Either $(-x + r) + lcm_{div}(P) \leq 0$ which is what we want to prove

  Or $(-x + r) + lcm_{div}(P) > 0$ which means that there exists $j \in [|0; lcm_{div}(P)|]$ and a $b \in \{r - 1\}$ such that $x = r - 1 + lcm_{div}(P)$ which contradicts one of the hypothesis.

- if $P(x)$ is $x+r = 0$, $-x+r = 0$ or $k*x_0+r \neq 0$ then, by a similar reasoning, it is either trivial or a contradiction to the hypothesis that there are no j and b such that $b \in B$, $j \in [|0; lcm_{div}(P)|]$ and $x = b + j$.

- if $P(x)$ is $P_1(x) \wedge P_2(x)$ or $P_1(x) \vee P_2(x)$, as $\neg\left(\exists b \in B(P), \exists j \in [|0; lcm_{div}(P)|], x = b + j\right)$ implies both $\neg\left(\exists b \in B(P_1), \exists j \in [|0; lcm_{div}(P)|], P(b + j)\right)$, a simple use of the hypothesis induction will do the job.

□

Corollary 3.6. *Pseudo-periodicity of PA formulas*

$$\forall P, \neg\left(\exists b \in B, \exists j \in [|0; lcm_{div}(P)|], P(b + j)\right) \Rightarrow \forall x, P(x) \Rightarrow P(x - lcm_{div}(P))$$

Proof. $(\exists b \in B, \exists j \in [|0; lcm_{div}(P)|], x = b + j)$ is decidable. □

3.5 Cooper's theorem

By combining these two principles, we can prove the elimination principle known as Cooper's theorem.

Theorem 3.7. *Cooper's theorem*
For all unitarized formula P of Presburger arithmetic that is quantifier free,

$$\exists x, P(x) \Leftrightarrow \left(\begin{array}{l} \exists b \in B, \exists j \in [|0; lcm_{div}(P)|], P(b + j) \\ \vee \exists j \in [|0; lcm_{div}(P_{-\infty}) - 1|], P_{-\infty}(j) \end{array}\right.$$

Proof. Let's prove the reciprocal implication first.

$$\boxed{\exists x, P(x) \Leftarrow \left\{ \begin{array}{l} \exists b \in B, \exists j \in [|0; lcm_{div}(P)|], P(b+j) \\ \vee \exists j \in [|0; lcm_{div}(P_{-\infty}) - 1|], P_{-\infty}(j) \end{array} \right.}$$

If $\exists b \in B, \exists j \in [|0; lcm_{div}(P)|], P(b+j)$ then it is obvious that $\exists x, P(x)$.

If $\exists j \in [|0; lcm_{div}(P_{-\infty}) - 1|], P_{-\infty}(j)$ then, as $lcm_{div}(P_{-\infty}) > 0$, there exists a k such that $j - k * lcm_{div}(P_{-\infty})$ is small enough to have $P(j - k * lcm_{div}(P_{-\infty})) \Leftrightarrow P_{-\infty}(j - k * lcm_{div}(P_{-\infty}))$. From $P_{-\infty}(j - k * lcm_{div}(P_{-\infty})) \Leftrightarrow P_{-\infty}(j)$ we deduce that $P(j - k * lcm_{div}(P_{-\infty})) \Leftrightarrow P_{-\infty}(j)$ which ends the proof.

$$\boxed{\exists x, P(x) \Rightarrow \left\{ \begin{array}{l} \exists b \in B, \exists j \in [|0; lcm_{div}(P)|], P(b+j) \\ \vee \exists j \in [|0; lcm_{div}(P_{-\infty}) - 1|], P_{-\infty}(j) \end{array} \right.}$$

As $\exists b \in B, \exists j \in [|0; lcm_{div}(P)|], P(b+j)$ is decidable, we have two options:

1. if $\exists b \in B, \exists j \in [|0; lcm_{div}(P)|], P(b+j)$ then we can obviously conclude.
2. if $\neg(\exists b \in B, \exists j \in [|0; lcm_{div}(P_{-\infty})|], P(b+j))$ then, as there exists x such that $P(x)$, we can use the pseudo-periodicity of PA formulas to find an x' that is small enough in order to have $P(x') \Leftrightarrow P_{-\infty}(x')$. From the periodicity of $P_{-\infty}$ we can conclude that there exists a d in $[|0; lcm_{div}(P_{-\infty}) - 1|]$ such that $P_{-\infty}(j)$ which ends the proof of Cooper's theorem.

□

Part II
Implementation using reflection

All the source files are available on my darcs repository: `https://patch-tag.com/r/gallais/agda/` (directory `src/presburger`).

4 The major role of reflection

We decided to use reflection in order to develop a certified solver. The power of dependent types allowed us to manipulate expressions and formulas with great precision and to be able to guarantee that our decision procedure is totally bug-free. Unlike what one could think, the use of dependent types did not force us to mix too much the computational content of the procedure with the proof that it does what it is supposed to do: most of the time we have on the one hand an algorithm that is working on the datatypes representing a subset of the formulas or expressions and on the other hand an inductive prove of the correctness of this algorithm.

The way our implementation is organised is completely reflecting this aspect of the solver: almost every `XXXX` module comes together with an `XXXX-prop` module. The first one contains the algorithms and all the computational content while the other one is just proving that everything is working great.

5 Data structures

The data structures are quite similar to the mathematical definitions: the formulas of Presburger arithmetic are defined using a datatype for the expressions and another one for the formulas. The semantics attached to these datatypes is straightforward: the variables are given integer values, the equality is the propositional equality, a conjunction is a sigma type, a disjunction is a sum type, etc.

5.1 Multiple datatypes versus Properties

The normalisation procedure is transforming formulas in their very own structure. There are two ways to deal with these modifications: one can either create a specific datatype matching the definition for each step or one can define properties on the formulas that will restrict the use of certain constructors.

5.2 Advantages & drawbacks of specialized datatypes

Advantages The use of different data-structures to represent the formulas in different forms is really close to the mathematical definitions that were given earlier. The main advantage is that you do not have to carry proofs along with the datatypes: the correction of your algorithm is guaranteed by the type checker.

Drawbacks The use of specialized datatypes to represent formulas forces to define as many semantics as there are datatypes which is quite tedious and unsatisfactory. Expressing and proving lemmas about the same formula (e.g. a unitarized formula is also linear) is too complicated: in our example, we have to use a unitarized formula, a function from unitarized formulas to linear ones, the semantics of unitarized formulas and the one of linear ones and prove that the interpretations of the two formulas (the unitarized one and its version lifted in the world of linear formulas) are equivalent $(\forall \rho, [|P|]_{unit}\rho \Leftrightarrow [|unit\_to\_lin(P)|]_{lin}\rho)$.

5.3 Advantages & drawbacks of properties

Advantages The use of properties on the formulas allows to have only one datatype for the formulas and to have to define only one semantic. It is really easy to express and prove lemmas about a formula (e.g. a unitarized formula is also linear : $\forall P, unit(P) \Rightarrow lin(P)$).

Drawbacks All the lemmas contains preconditions (properties of the formulas involved in the theorem) that would be expressed by the formulas' type itself if we had multiple datatypes. All the functions have to deal with proofs about the output they are creating (for example a function from linear formulas to unitarized ones will construct a proof that the output formula is unitarized).

5.4 Choice

The implementation we are presenting uses the second approach: we have only two main datatypes[1] – one for the expressions and one for the formulas – and lots of properties on them[2].

The use of sigma types to manipulate formulas that have a certain property lightens a lot the statements: Nnf represents the formulas in negation normal form, Lin the ones that are linear and Unf the unitarized ones.

6 Difficulties

6.1 Lack of libraries on $\mathbb{Z}$

Unlike $\mathbb{N}$, there are only a few results on $\mathbb{Z}$ in Agda's standard library [4]. I had to formalise the basic notions (divisibility, GCD, LCM, etc.)[3] and prove their common properties.

I also had to prove that $\mathbb{Z}$ is commutative ring and that the order on the integer is compatible with the common operators.

6.2 Structural recursion

Agda's termination checker is only verifying that the recursive calls of the defined functions are always structurally smaller. Thanks to AIM XI, it even became possible to be more efficient in detecting that a recursive call nested in a mutually recursive function or after a `with` clause (which is more or less the same because of the way Agda handles `with` clauses) was structurally decreasing which saved us some tedious work[4].

6.3 Resource consumption

Agda's typechecker is currently not using opaque definitions as a basis and is not capable of reusing everything that remained the same in order to typecheck a small change. At the end of this internship, typechecking a rather important theorem (e.g. one of the lemmas that we had to prove in order to prove Cooper's theorem) could take half a minute on a quite decent computer.

This was a real brake on the development of the solver even if it could have been worse: the separation between computational content and proofs allowed to save some resources and the performances were not as bad as the ones encountered by some of the team's PhD students who where trying to formalise category theory (a file could take up to 15 minutes to be typechecked).

[1] See `Representation.agda`

[2] See `Properties.agda`

[3] All these formalisations are based on their $\mathbb{N}$ counterpart that are in the standard library.

[4] The techniques that are used to avoid this problem of termination checking are almost always the manual expansion of the `with` clause or the recursive call to explicit the fact that it is structurally decreasing

7 Future work

7.1 Gluing everything together

At this point, Cooper's theorem is proved and relies only on a couple of assumptions on least common multiples on the integers. The elimination procedure for is almost finished (a few decidability results are lacking).

7.2 Interface improvement

Since AIM XI, it has become possible to access to a representation of the current goal. The development version of Agda [6] comes now with two constructs (`quoteGoal_in_` and `quote`) and a datatype (`Term`) which allow functions to scrutinize the goal.

It would be nice to have a function that would allow the use of the automatic goal quoting in order to improve even more the usability of this decision procedure. For a working example of such a function, see the solver for propositional logic on the darcs repository (directory `rls1`) .

7.3 Decision procedure optimisations

7.3.1 A smaller elimination set?

In the elimination procedure, the elimination set is defined in order to use the proposition that is equivalent to P when x tends to be very small. It is also possible to take advantage of the proposition that is equivalent to P when x tends to be very big; in this case the elimination set is different and might be smaller / bigger.

A common optimisation is to compute both sets and to use the smallest one in order to avoid the formula's size blow-up (every $\exists d \in [|m; m + n|], \ldots$ statement is actually expanded as a disjunction with $n + 1$ subterms in the procedure).

7.3.2 More subtle datatypes?

The B sets are handled with lists: in practice they are multisets rather than sets. Defining a module allowing the user to define proper finite subsets of a set in which equality is decidable would sometimes limit the blow-up in the formula's size (but it would cost much more in term of resources).

References

[1] Amine Chaieb and Tobias Nipkow. "Proof Synthesis and Reflection for Linear Arithmetic". In: *J. Autom. Reasoning* 41.1 (2008), pp. 33–59.

[2] Amine Chaieb and Tobias Nipkow. "Verifying and Reflecting Quantifier Elimination for Presburger Arithmetic". In: *LPAR*. 2005, pp. 367–380.

[3] D. C. Cooper. "Theorem proving in arithmetic without multiplication". In: *Machine Intelligence 7* (1972), pp. 91–99.

[4] Nils Anders Danielsson. *Agda's standard library*. URL: `http://www.cs.nott.ac.uk/~nad/repos/lib/`.

[5] Michael Jo Fischer et al. “Super-Exponential Complexity of Presburger Arithmetic”. In: 1974, pp. 27–41.

[6] Agda development team. *Development version - darcs repository.* URL: `http://code.haskell.org/Agda`.