Décidabilité de l'arithmétique de Presburger

Année scolaire 2024-2025 – 28 mars 2025

L'objectif de ce projet est de prouver la décidabilité de l'arithmétique de Presburger. Le projet est découpé en trois parties : elles ne sont pas indépendantes mais peuvent être traitées en parallèle, certains résultats pouvant éventuellement être admis pour avancer sur d'autres développements.

L'arithmétique de Presburger correspond à l'arithmétique de Peano du premier ordre (c'est-à-dire avec quantification sur les entiers naturels), privée de la multiplication. Par application itérée de l'addition, il est possible de multiplier un nombre par une constante. On pourra donc écrire des énoncés comme $\forall n, \exists m, n = 2 \cdot m \vee n = 2 \cdot m + 1$, qu'on peut réécrire sous la forme $\forall n, \exists m, n = m + m \vee n = m + m + 1$. Cependant, on ne pourra pas écrire d'énoncé comme $\forall n, \forall m, n > 0 \wedge m > 1 \Rightarrow n \times m > n$ car $n \times m$ n'est pas exprimable. Presburger a montré que ce fragment est décidable : il existe un algorithme qui, étant donnée une proposition, décide si oui ou non cette proposition est satisfiable. Quelques années plus tard, Gödel montrera que l'arithmétique du premier ordre avec la multiplication est indécidable.

Mojżesz Presburger. 1929. Über die Vollst ändigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du congrès de Mathématiciens des Pays Slaves*. 92–101

La première partie propose une procédure de décision pour un noyau de l'arithmétique de Presburger. Ce noyau est une syntaxe réduite pour les propositions arithmétiques qui se prête bien à la formalisation. Dans cette partie, nous proposons un codage en automates finis selon la méthode introduite par Büchi.

J. Richard Büchi. 1960. Weak Second-Order Arithmetic and Finite Automata. *Mathematical Logic Quarterly 6, 1-6 (1960), 66–92*. DOI :`http://dx.doi.org/10.1002/malq.19600060105`

La deuxième partie introduit la syntaxe réduite et montre que la procédure définie dans la première partie décide les propositions écrites dans cette syntaxe.

La troisième partie introduit une syntaxe plus riche pour écrire les propositions arithmétiques et montre comment réécrire ces propositions dans la syntaxe noyau traitée dans la deuxième partie.

1 Procédure de décision par automates

1.1 Représentation des ensembles de solutions par des automates finis

La méthode de Büchi repose sur une transformation des propositions de l'arithmétique de Presburger en automates finis.

Un automate fini (déterministe, complet) $\mathcal{A}$ est la donnée d'un 5-uplet $\mathcal{A} = (A, S, i, T, F)$, où A est un ensemble fini non vide de lettres (un alphabet), S est un ensemble fini non vide d'états, i est l'état initial, $T : S \times A \to S$ est la fonction de transition, et $F \subseteq S$ est l'ensemble des états finaux. Une suite finie de lettres de A est appelée un mot, l'ensemble des mots est noté A^*, et on note $s \to^a s'$ lorsque $s' = T(s, a)$. On dit qu'un mot $a_1 \dots a_n \in A^*$ est accepté par $\mathcal{A}$ lorsqu'on a $i \to^{a_1} s_1 \cdots \to^{a_n} s_n$ avec $s_1, \dots, s_n \in S$ et $s_n \in F$. On note $\mathcal{L}(\mathcal{A})$ l'ensemble des mots acceptés par $\mathcal{A}$.

Une proposition p avec n variables libres $x_1, \dots, x_n$ est associée à un automate fini $\mathcal{A}$ sur l'alphabet $\{0, 1\}^n$, qui reconnaît l'écriture binaire des solutions de p.

Pour tout entier naturel n, on note $\overline{u}^2$ son écriture binaire, où u est un mot sur l'alphabet $\{0, 1\}$: si $u = u_1 \dots u_m$, on a $n = \overline{u_1 \dots u_m}^2 = \sum_{i=1}^{m} 2^{m-i} u_i$: on suit ici la convention usuelle

big-endian, ou gros-boutiste, où le chiffre de poids fort vient en premier.

En pratique, pour les calculs, on préfère un codage *little-endian*, ou petit-boutiste, qui convient mieux à la propagation des retenues. On code les n-uplets d'entiers naturels en des mots sur l'alphabet $\{0,1\}^n$ par la fonction $[\![\cdot]\!] : (\{0,1\}^n)^* \to \mathbb{N}^n$ qui associe au mot $u = u_1 \dots u_m$ le n-uplet $[\![u]\!] = (v_1, \dots v_n)$, avec $v_i = \overline{(u_m)_i \dots (u_1)_i}^2$ (on note ici que l'écriture est renversée : le codage est donc petit-boutiste), où $(u_j)_i$ désigne la i^{e} composante du n-uplet u_j.

Par exemple, le mot $(0,1)(1,0)(0,1)$ code la paire $(2,5)$: en effet $2 = \overline{10}^2$ et $5 = \overline{101}^2$. Le codage n'est pas injectif : on peut ajouter des 0 à la fin du mot (autrement dit, des 0 à gauche du nombre dans l'écriture usuelle) sans changer sa valeur, et le mot $(0,1)(1,0)(0,1)(0,0)$ code également $(2,5)$. Le mot $(1,1,0)(1,0,0)(1,1,0)(1,0,0)$ code le triplet $(15,5,0)$.

1.2 Représentation des automates finis en Coq

On conseille d'utiliser le type suivant pour représenter les lettres des alphabets. On pourra au besoin ajouter une contrainte sur la longueur des n-uplets, mais on pourra le plus souvent s'en passer.

```
Definition Symbol := list bool.
```

On conseille de coder les états des automates par un segment initial des entiers naturels, de 0 jusqu'à $s-1$. Ainsi l'ensemble des états d'un automate est caractérisé par son cardinal, s. On conseille également de fixer l'état initial à 0.

Ces choix permettent de représenter un automate par la donnée du nombre de ses états, sa fonction de transition, et la fonction caractéristique de l'ensemble de ses états finaux.

```
Record Automaton := mkAutomaton {
  state_count: nat;
  transition: nat -> Symbol -> nat;
  accept: nat -> bool;
}.
```

1.3 Automates pour les propositions de Presburger

On fixe un entier i et un entier $k = \overline{k_1 \dots k_m}^2$. Les solutions de la proposition $x_i = k$ sont reconnaissables par un automate à $m+2$ état. En partant de l'état 0, pour $j < m$, on passe de l'état j à l'état $j+1$ si on lit la lettre a telle que $a_i = k_{m-j}$, l'état m est final, et si on est sur l'état m, on peut rester sur l'état m si on lit la lettre a telle que $a_i = 0$. Toutes les autres transitions mènent à l'état puits $m+1$.

Écrire une fonction `automaton_number_equal: nat -> nat -> Automaton` qui construit un tel automate étant donnés i et k.

Écrire une fonction `automaton_accept: list nat -> Automaton -> bool` qui vérifie qu'un n-uplet donné est accepté par un automate.

Prouver que l'automate construit par `automaton_number_equal` accepte les n-uplets solutions de $x_i = k$.

Soient trois entiers i, j, k. Les solutions de la proposition $x_i + x_j = x_k$ sont reconnaissables par un automate à 3 états. La fonction de transition code l'addition posée colonne par colonne, l'état 0 correspond à l'état sans retenue, l'état 1 correspond à l'état avec retenue. L'état 0 est final et l'état 2 sert de puits.

Écrire une fonction `automaton_sum: nat -> nat -> nat -> Automaton` qui construit un tel automate étant donnés i, j et k. Prouver que l'automate construit par `automaton_sum` accepte les n-uplets solutions de $x_i + x_j = x_k$.

Soit un $\mathcal{A} = (A, S, i, T, F)$. L'automate $\overline{\mathcal{A}} = (A, S, i, T, S \setminus F)$ reconnaît le langage $\mathcal{L}(\overline{\mathcal{A}}) = A^* \setminus \mathcal{L}(\mathcal{A})$.

Écrire une fonction `automaton_complement: Automaton -> Automaton` et prouver que l'automate construit reconnaît le complémentaire du langage reconnu par l'automate d'entrée.

Soient deux automates $\mathcal{A}_1 = (A, S_1, i_1, T_1, F_1)$ et $\mathcal{A}_2 = (A, S_2, i_2, T_2, F_2)$ sur le même alphabet A. On construit $\mathcal{A}_{1\cap 2} = (A, S_1 \times S_2, (i_1, i_2), T_{1\times 2}, F_{1\cap 2})$ en posant **[corrigé le 2025-04-17]** $T_{1\times 2}$:

$((s_1, s_2), a) \mapsto (T_1(s_1, a), T_2(s_2, a))$ et $F_{1\cap 2} = F_1 \times F_2$: on a $\mathcal{L}(\mathcal{A}_{1\cap 2}) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$. De même, on construit $\mathcal{A}_{1\cup 2} = (A, S_1 \times S_2, (i_1, i_2), T_{1\times 2}, F_{1\cup 2})$ en posant $F_{1\cup 2} = (F_1 \times S_2) \cup (S_1 \times F_2)$: on a $\mathcal{L}(\mathcal{A}_{1\cup 2}) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$.

Écrire deux fonctions

```
automaton_and: Automaton -> Automaton -> Automaton
```

et

```
automaton_or: Automaton -> Automaton -> Automaton
```

qui construisent les automates $\mathcal{A}_{1\cap 2}$ et $\mathcal{A}_{1\cup 2}$. Prouver que l'automate construit par `automaton_and` accepte un n-uplet lorsque celui-ci est accepté par les deux automates d'entrée, et que l'automate construit par `automaton_or` accepte un n-uplet lorsque celui-ci est accepté par au moins un des deux automates d'entrée.

Soit une proposition p sur les variables $x_1, \ldots, x_n$ et un automate $\mathcal{A} = (\{0,1\}^n, S, i, T, F)$ qui reconnaît les solutions de p. Les solutions de la proposition $\exists x_1(p)$ sont reconnues par l'automate $\exists_1\mathcal{A} = (\{0,1\}^{n-1}, 2^S, \{i\}, T', \exists_1 F)$ où **[corrigé le 2025-04-17]** $T' : (s, (a_2, \ldots, a_n)) \in 2^S \times \{0,1\}^{n-1} \mapsto \{T(s', (a, a_2, \ldots, a_n)) \mid s' \in s, a \in \{0,1\}\}$ et $\exists_1 F = \{s \in 2^S \mid \exists s' \in s, s' \in F\}$.

Écrire une fonction

```
automaton_exists: Automaton -> Automaton
```

qui construit l'automate $\exists_1\mathcal{A}$.

[corrigé le 2025-04-17] On obtient l'automate qui reconnaît les solutions de $\forall x_1(p)$ par double complémentation : $\forall x_1(p) = \neg\exists x_1(\neg p)$.

Prouver que l'automate construit par `automaton_exists` accepte un n-uplet $(x_2, \ldots, x_n)$ **[corrigé le 2025-04-17]** écrit sur ℓ chiffres lorsqu'il existe un entier x_1 écrit sur ℓ chiffres tel que l'automate d'entrée accepte $(x_1, \ldots, x_n)$.

Écrire une fonction `automaton_choice: Automaton -> option (list nat)` qui renvoie un témoin `Some env` s'il existe un n-uplet `env` reconnu par l'automate, et `None` si l'automate ne reconnaît aucun mot (on pourra faire un parcours de graphe en largeur). Montrer sa correction.

2 Décidabilité du noyau

On définit la grammaire des propositions suivantes, où les variables sont représentées par des indices de De Bruijn : la variable d'indice 0 est liée par le quantificateur existentiel ou universel le plus proche de la variable, la variable d'indice 1 par le quantificateur suivant, etc. Les variables sont libres lorsque leur indice est plus grand que le nombre de quantificateurs.

```
Inductive presburger_proposition_kernel :=
| VarEqNat (k n: nat)
| VarEqSum (ks k1 k2: nat)
| KOr (p q: presburger_proposition_kernel)
| KAnd (p q: presburger_proposition_kernel)
| KNot (p: presburger_proposition_kernel)
| KExists (p: presburger_proposition_kernel)
| KForall (p: presburger_proposition_kernel).
```

Écrire une fonction

```
automaton_of_presburger : presburger_proposition_kernel -> Automaton
```

Écrire une fonction

```
prop_of_presburger : list nat -> presburger_proposition_kernel -> Prop
```

qui interprète les propositions de Presburger.

Montrer que l'automate construit reconnaît exactement les solutions de la proposition de Presburger.

3 Syntaxe de surface pour les formules de Presburger

On définit la grammaire des propositions suivantes, où les variables sont nommées.

```
Require Import String.

Definition var := string.

Inductive presburger_expression :=
| Var (_: var)
| Nat (_: nat)
| Add (_ _: presburger_expression)
| Mul (_: nat) (_: presburger_expression).

Inductive comparison := Eq | Lt | Le | Gt | Ge.

Inductive presburger_proposition :=
| Compare (_: presburger_expression) (_: comparison) (_: presburger_expression)
| Forall (_: var) (_: presburger_proposition)
| Exists (_: var) (_: presburger_proposition)
| Or (_ _: presburger_proposition)
| And (_ _: presburger_proposition)
| Not (_: presburger_proposition).
```

Écrire une fonction

```
presburger_evaluate : (var -> nat) -> presburger_expression -> nat
```

qui évalue les expressions et une fonction

```
presburger_interpret : (var -> nat) -> presburger_proposition -> Prop
```

qui interprète les propositions de Presburger.

En compilant vers des propositions noyaux, montrer que les propositions sont décidables :

```
Theorem presburger_decidable : forall p : presburger_proposition,
  { exists env, presburger_proposition env p } +
  { forall env, ~ presburger_proposition env p }.
```