Mots et données infinies

Claire David

Juillet 2004

Stage de DEA effectué au sein du LIAFA sous la direction d'Anca Muscholl et en collaboration avec Luc Segoufin (INRIA Orsay) entre le 15 mars et le 30 juin 2004. L'objet de ce stage était d'étudier des aspects logiques et automates des langages de mots finis, définis sur un alphabet infini du type $\Sigma \times D$ où Σ est un alphabet fini de lettres et D un ensemble infini de données.

Le sujet de ce stage concerne la modélisation des phénomènes réguliers utilisant des alphabets infinis. Les alphabets infinis trouvent des applications dans la modélisation des systèmes distribués, par exemple pour identifier un nombre arbitraire de processus, ou bien dans la représentation des données (non-bornées) de documents XML. Les opérations que l'on permet ici sur l'alphabet infini ne sont que des comparaisons d'égalité sur les données infinies, donc on ne fait pas d'hypothèse particulière sur leur domaine.

La gestion des données infinies est un problème difficile, parce qu'il n'existe pas d'extension canonique des systèmes à nombre fini d'états (réguliers) ayant de bonnes propriétés de décidabilité. L'objectif du stage a été de proposer des fragments de logique (premier/second ordre) et/ou des automates décidables pour la description de tels langages.

Une première partie de ce mémoire est consacrée aux définitions générales du vocabulaire, des logiques (premier ordre et second ordre augmenté d'un prédicat d'égalité pour les données) et des automates (automates classiques augmentés avec des registres). Une seconde partie est consacrée à un résumé de travaux antérieurs sur le même sujet. On donne ensuite dans la troisième partie une série d'exemples de propriétés définissables par des formules des différentes logiques. Dans une quatrième partie, on montre que l'utilisation, dans les formules, de trois variables du premier ordre rend les logiques considérées indécidables. En ce qui concerne les automates, deux registres suffisent aussi à rendre le problème du vide indécidable. Il ne faut donc autoriser qu'au maximum deux variables du premier ordre pour la logique et n'autoriser qu'un seul registre aux automates. On donne dans cette partie les caractérisations connues dans le cas sans donnée de la logique à deux variables et on donne à partir de cela une définition plus restreinte des automates à registres (déterministes à un registre, partiellement ordonnés entre deux mises à jour). Dans une cinquième partie, on reprend les exemples donnés pour les différentes logiques et on regarde ce que l'on peut exprimer avec nos automates. Le résultat principal (sixième partie) est que la logique du premier ordre à deux variables avec l'ordre est décidable et que la satisfaisabilité de telles formules est un problème NEXPTIME-complet. Dans une septième partie, on montre que le vide est indécidable pour nos automates précédemment définis. Enfin, la dernière partie est consacrée à un résumé des résultats obtenus et à quelques idées et explications sur la suite de ces travaux.

1 Introduction et définitions

Dans cette partie, on commence par définir le vocabulaire général qui sera utilisé dans ce mémoire. On définit ensuite les logiques auxquelles on s'intéresse et la façon dont on traite les données. Enfin, on explique la manière que l'on a choisie pour étendre les automates classiques afin de tenir compte des données.

1.1 Vocabulaire

On utilisera le vocabulaire suivant :

– Σ représente un alphabet fini de lettres
– D représente un alphabet infini de données
– Pour un mot w de $(\Sigma \times D)^*$, on désigne par "classe" l'ensemble des positions de w étiquetées par une même donnée.
– Lorsque l'on parle de l'étiquette d'une position on désigne la lettre de Σ qui lui est associée.

1.2 Logiques

Afin de traiter les données, on ajoute à la logique classique sur les mots un prédicat binaire $\sim$ qui permet de tester l'égalité des données entre deux positions d'un mot.

Logiques du premier ordre (FO) :

On distingue deux logiques du premier ordre selon l'ajout ou non du prédicat successeur : $FO(<, \sim)$ et $FO(<, \sim, +1)$.

Syntaxe : La logique $FO(<, \sim)$ représente l'ensemble des formules définies par la grammaire suivante (avec $a \in \Sigma$) :

$$\begin{array}{rcl} \varphi & \rightarrow & a(x) \mid x < y \mid x \sim y \mid \\ & & \exists x\ \psi(x) \mid \varphi \vee \varphi \mid \neg\varphi \end{array}$$

Les formules de $FO(<, \sim, +1)$ sont définies par la même grammaire à laquelle on a rajouté le successeur :

$$\begin{array}{rcl} \varphi & \rightarrow & a(x) \mid x < y \mid x \sim y \mid x = y+1 \mid \\ & & \exists x \ \psi(x) \mid \varphi \vee \varphi \mid \neg \varphi \end{array}$$

Sémantique : Soit $w \in (\Sigma \times D)^*$.
Soit $\varphi(x_1, x_2, \cdots x_n)$ une formule dont l'ensemble des variables libres est Var$= \{x_1,\ x_2, \cdots\ x_n\}$.
On interprète les variables libres de la formule comme des positions du mot $w \in (\Sigma \times D)^*$ en définissant une fonction d'interprétation $F : \text{Var} \rightarrow \text{dom}(w) = \{1 \cdots |w|\}$.

On définit par induction structurelle la sémantique de ces formules sur les mots de la manière suivante :

$$\begin{array}{lll} w, F \models & a(x) & \text{ssi } \Pi_\Sigma(F(x)) = a \text{ ie } F(x) \text{ est étiqueté par } a \text{ dans le mot } w \\ w, F \models & x < y & \text{ssi } F(x) \text{ précède strictement } F(y) \text{ dans } w \\ w, F \models & x \sim y & \text{ssi } F(x) \text{ et } F(y) \text{ sont étiquetées par la même donnée dans } w \\ w, F \models & y = x+1 & \text{ssi } F(y) \text{ est le successeur de } F(x) \text{ dans } w \\ w, F \models & \varphi_1 \vee \varphi_2 & \text{ssi } w, F \models \varphi_1 \text{ ou } w, F \models \varphi_2 \\ w, F \models & \neg\varphi & \text{ssi } w, F \not\models \varphi \\ w, F \models & \exists x \ \psi(x) & \text{ssi il existe } F' \text{ telle que } F'(y) = F(y) \text{ pour tout } y \neq x \text{ et } w, F' \models \psi(x) \end{array}$$

Remarque Si on s'autorise au moins trois variables, il est inutile de différencier les cas avec et sans successeur car celui-ci s'exprime à l'aide de l'ordre de la manière suivante :

$$y = x+1 \text{ ssi } x < y \ \wedge \ \forall z \ [\ x < z \ \Rightarrow \ [z = y \ \vee \ y < z]\ ]$$

On s'intéresse cependant par la suite aux formules utilisant au plus deux variables ce qui justifie cette distinction.

On notera dans la suite $FO^k(<, \sim)$ la restriction de $FO(<, \sim)$ à k variables. En d'autres termes, $FO^k(<, \sim)$ représente l'ensemble des formules de $FO(<, \sim)$ n'utilisant que k variables.

Comme d'habitude, on utilise aussi dans les formules $\wedge$, $\forall$, true, false, max(x) pour exprimer que la position x est la dernière du mot et min(x) pour exprimer que la position x est la première du mot. On utilisera aussi max$(x+k)$ (k étant un entier positif) pour exprimer le fait que la position $k^{\text{ième}}$ après la position x est la dernière du mot.
Par exemple, pour $k = 2$, la formule a la forme suivante :

$$\begin{array}{rl} \max(x+k) := & \exists y \ [(x < y) \ \wedge \ [\exists x \ (y < x) \ \wedge \ \max(x)]] \\ \wedge & \neg[\exists y \ [(x < y) \ \wedge \ [\exists x \ (y < x) \ \wedge \ [\exists y \ (x < y) \ \wedge \max(y)]]]] \end{array}$$

Logique Existentielle Monadique du Second Ordre (EMSO)

A partir des logiques précédentes on définit deux logiques monadiques existentielles du second ordre $EMSO^2(<, \sim)$ et $EMSO^2(<, \sim, +1)$.
$EMSO^2(<, \sim)$ [resp $EMSO^2(<, \sim, +1)$] représente l'ensemble des formules de la forme :

$$\exists X_0 \cdots \exists X_k \ \phi(X_0, \cdots, X_k)$$

où $\phi(X_0, \cdots, X_k)$ est une formule $FO^2(<, \sim)$ [resp $FO^2(<, \sim, +1)$] à laquelle on a rajouté le prédicat binaire de base $x \in X_i$. Etant donné un mot w, les variables du second ordre sont interprétées comme des ensembles de positions du mot w. La sémantique du nouveau prédicat étant la sémantique usuelle :

$$w, F \models x \in X \text{ ssi } F(x) \in F(X)$$

1.3 Automates

Afin de pouvoir traiter les données on rajoute aux automates sur les mots classiques des registres. L'automate peut en plus de lire la lettre, stocker la donnée courante dans le registre ou encore comparer la donnée courante à celle stockée dans le registre. Comme pour les automates classiques, on a une notion d'automate déterministe que l'on précise ci-dessous. De même, on dit qu'un automate est uni-directionnel si sa tête de lecture ne peut se déplacer que de la gauche vers la droite. Au contraire un automate est dit bi-directionnel si sa tête de lecture peut se déplacer à droite comme à gauche. Dans la suite on s'intéresse à des automates bi-directionnels (car les automates uni-directionnels ne sont pas suffisamment expressifs). On dit qu'un calcul d'automate bi-directionnel sur un mot w est acceptant ssi il atteint un état final.

Transitions :

$$\text{q} \xrightarrow{a\ (\sim_i)_{i\in I}\ (\nsim_j)_{j\in J},\ (\mathrm{up}_k)_{k\in K},\ \delta} \text{q'}$$

- a représente une lettre de l'alphabet $\Sigma \cup \{\triangleright, \triangleleft\}$ où $\triangleright$ est un marqueur de début de mot et $\triangleleft$ est un marqueur de fin de mot. On rajoute ces deux marqueurs à tous les mots considérés car les automates peuvent être bi-directionnels.
- $\sim_i$ signifie que la position lue est dans la classe définie par la donnée stockée dans le registre i
- $\nsim_j$ signifie que la position lue est dans une classe différente de la classe définie par la donnée stockée dans le registre j
- up_k signifie qu'on fait une mise à jour du registre k c-à-d qu'on stocke la donnée courante dans le registre k

– δ représente le déplacement de la tête de lecture : "d" pour droite, "g" pour gauche, et $-$ si on ne bouge pas (ceci étant équivalent à faire deux transitions succesives, droite puis gauche)

Un automate à registre est dit déterministe si étant donné un état q, une lettre a et $(\sim_i)_{i \in I}$ $(\nsim_j)_{j \in J}$ un état par rapport aux registres, il existe une unique transition du type précédent et si l'état initial est unique. Pour l'état initial, on suppose que les registres contiennent tous la donnée associée à la première position du mot.

2 Travaux existant sur le sujet

2.1 Automates uni-directionnels à plusieurs registres : Bouyer-Petit-Thérien [BPT01]

Dans leur article, P. Bouyer, A. Petit et D. Thérien définissent des automates uni-directionnels à plusieurs registres, dans le but d'étudier une classe de langages contenant plus que les langages reconnaissables par des automates temporisés déterministes. Ils donnent pour cela une caractérisation algébrique des langages correspondants. Le vide de leurs automates à plusieurs registres, uni-directionnels est décidable. Pour montrer ce résultat, les auteurs construisent à partir d'un automate à registres un automate fini classique qui reconnaît exactement la projection sur l'alphabet fini du langage reconnu par l'automate à registres. Cette construction qui prouve la décidabilité du problème du vide montre aussi que les automates uni-directionnels ont, du point de vue de l'alphabet fini, une expressivité réduite.
On retiendra de cet article que les automates à registres, uni-directionnels reconnaissent des langages dont la projection sur l'alphabet fini est régulière.

2.2 Neven-Schwentick-Vianu [NSV01]

Dans leur article, F. Neven, Th. Schwentick et V. Vianu définissent deux types d'automates afin de gérer les données :

– Les automates à registres.
– Les automates à jetons qui plutôt que de stocker des données, marquent des positions avec les jetons. Ils peuvent ensuite non seulement comparer la donnée courante avec les données des positions marquées mais aussi savoir quels sont les éventuels jetons qui marquent déjà la position courante.

Dans [NSV01], ils définissent aussi la logique du premier ordre qu'il notent FO^* et celle du second ordre (MSO^*) afin de les comparer aux automates.
Les résultats sont les suivants :

Automates à registres

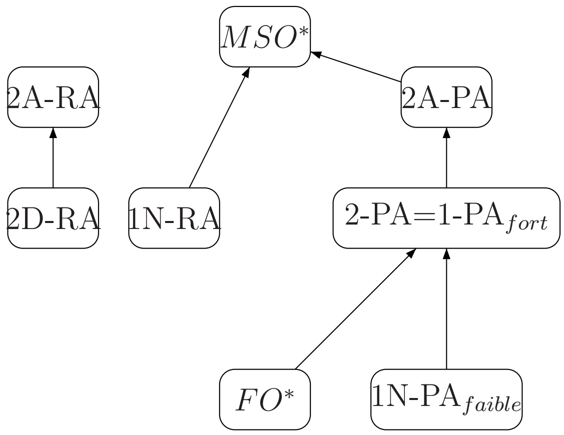
- Les automates uni-directionnels (1-RA) sont inclus dans la logique du second ordre.
- Les automates bi-directionnels sont incomparables aux deux logiques.
  - Les automates bi-directionnels déterministes (2D-RA) sont incomparables avec la logique du second ordre.
    Ils peuvent reconnaître le langage des mots de la forme $u\#v$ tels que u et v ont le même nombre de classes différentes. Or, si la logique du second ordre pouvait définir ce langage, $\{a^n b^n | n \geq 0\}$ serait définissable par une formule du second ordre sans donnée et donc régulier.
  - Les automates bi-directionnels alternants (2A-RA) n'arrivent pas à capturer toute la logique du premier ordre.
- Le vide des automates uni-directionnels est décidable mais l'universalité ne l'est pas.
- Le vide des automates bi-directionnels est indécidables.

Automates à jetons Dans le cas uni-directionnel, on sépare ces automates en deux sous classes selon la façon dont on introduit les nouveaux jetons. On peut introduire les jetons à la position courante (on dira que l'automate est faible) ou introduire les jetons au début du mot (on dira que l'automate est fort). Dans le cas bi-directionnel, ce choix n'a pas d'importance compte tenu du fait que l'automate peut revenir en arrière.
Les résultats obtenus sont les suivants :

- Le déterminisme ne change pas l'expressivité des automates uni-directionnels-forts (1-PA$_{fort}$) et bi-directionnels (2-PA) qui ont le même pouvoir d'expression.
- La classe des langages définissables par la logique du premier ordre est contenue strictement dans la classe des automates uni-directionnels, forts et déterministes.
- La logique du premier ordre et les automates uni-directionnels faibles non-déterministes (1N-PA$_{faible}$) sont incomparables.
- Les automates bi-directionnels alternants (2A-PA) sont inclus dans la logique du second ordre.
- Le vide des automates uni-directionnels faibles et déterministes (1D-PA$_{faible}$) est indécidable.

Les automates à jetons qui semblent plus en adéquation avec la logique sont indécidables.

La figure suivante résume les résultats (les flêches correspondant aux inclusions) :



3 Expressivité des différentes logiques - Exemples

Cette partie nous permet, à travers des exemples, de nous faire une idée de l'expressivité des différentes logiques. A condition d'utiliser correctement les données, peut définir tous les langages réguliers et des langages dont la projection sur Σ n'est pas régulière et même non hors-contexte.

3.1 Exemples de propriétés exprimables dans $FO^2(<,\sim)$

3.1.1 Premier

L'élément x est le premier de sa classe :

$$\mathbf{Premier(x)} := \forall y\ [\ y < x \rightarrow x \nsim y\ ]$$

3.1.2 Au moins k classes

Il y a au moins k classes distinctes dans le mot w :

$$\mathbf{Classe_{\geq k}} := \exists x\ \varphi_{\geq k}(x)$$

La formule $\varphi_{\geq k}(x)$ vérifie que la position x est la première position de sa classe et qu'à sa droite, le mot w contient au moins $k-1$ positions elles aussi premières de leur classe. On définit la formule $\varphi_{\geq k}(x)$ par induction de la manière suivante :

$$\begin{array}{lrl} k = 1 & \varphi_{\geq 1}(x) := & \text{Premier}(x) \\ k = n+1 & \varphi_{\geq n+1}(x) := & \text{Premier}(x) \wedge \exists y\ [\ x < y\ \wedge\ x \nsim y\ \wedge \varphi_{\geq n}(y)\ ] \end{array}$$

3.1.3 Exactement k classes

Il y a exactement k classes distinctes dans le mot w ssi il y en a au moins k et il n'y en a pas au moins $k+1$.

$$\textbf{Classe}_{\mathbf{=k}} := \text{Classe}_{\geq k} \wedge \neg\text{Classe}_{\geq k+1}$$

3.1.4 Au moins k éléments

Il y a au moins k éléments dans la classe de x :

$$\textbf{Element}_{\geq \mathbf{k}}(\mathbf{x}) := \exists y\, [x \sim y \ \wedge \ Premier(y) \ \wedge \varphi_{\geq k}(y)]$$

Le début de la formule $\text{Element}_{\geq k}(x)$ identifie le premier élément de la classe de x. Puis la formule $\varphi_{\geq k}(y)$ compte qu'il y a, à droite de y, $k-1$ positions différentes dans la même classe que y. On définit la formule $\varphi_{\geq k}(y)$ par induction de la manière suivante :

$$\begin{array}{lrl} k = 1 & \varphi_{\geq 1}(y) := & true \\ & & \vdots \\ k = n+1 & \varphi_{\geq n+1}(y) := & \exists x \ [\ y < x \ \wedge \ x \sim y \ \wedge \varphi_{\geq n}(x)\ ] \end{array}$$

3.1.5 Exactement k éléments

Il y a exactement k éléments dans la même classe que x dans le mot w ssi il y en a au moins k et il n'y en a pas au moins $k+1$.

$$\textbf{Element}_{\mathbf{=k}}(\mathbf{x}) := \text{Element}_{\geq k} \wedge \neg\text{Element}_{\geq k+1}$$

3.1.6 $\{\mathbf{a^n b^n} \mid \mathbf{n} \geq \mathbf{0}\}$

$\Sigma = \{a, b\}$
On appelle $L_{a^n b^n} \subseteq (\Sigma \times D)^*$, le langage des mots w tels que :

- $\Pi_\Sigma(w) \in \{a^n b^n \mid n \geq 0\}$
- Les données du mot forment n classes différentes : chacune contenant exactement une position étiquetée a et une autre étiquetée b.

La formule suivante définit ce langage :

$$\begin{array}{rl} \varphi_{\mathbf{a^n b^n}} := & \forall x \ a(x) \leftrightarrow [\forall y \ (y < x) \rightarrow a(y)] \\ \wedge & \forall x \ \exists y \ [x \neq y \ \wedge \ x \sim y] \\ \wedge & \forall x \forall y \ (x \sim y) \rightarrow [a(x) \ \wedge b(y)] \vee [a(y) \wedge b(x)] \vee (x = y) \end{array}$$

La première partie de la formule assure que la projection sur Σ du mot est bien dans $a^* b^*$. Le seconde assure que toute classe a au moins deux éléments. Enfin, la dernière partie de la formule assure que chaque classe du mot contient exactement une occurrence de a et une occurrence de b.

3.1.7 $\{\mathbf{a^n b^n c^n} \mid \mathbf{n} \geq \mathbf{0}\}$

$\Sigma = \{a, b, c\}$
On considère $L_{a^n b^n c^n} \subset (\Sigma \times D)^*$, le langages des mots w tels que :
- $\Pi_\Sigma(w) = a^n b^n c^n$ pour $n \geq 0$
- Les données du mot forment n classes différentes : chacune contenant exactement une position étiquetée a, une autre étiquetée b et une autre étiquetée c

La formule suivante définit ce langage :

$$\begin{array}{rll} \varphi_{\mathbf{a^n b^n c^n}} := & \forall x \forall y & [x < y \ \wedge \ a(y)] \ \rightarrow \ a(x) \\ \wedge & \forall x \forall y & [x < y \wedge b(y)] \ \rightarrow \ a(x) \vee b(x) \\ \wedge & \forall x \forall y & [x \sim y \ \rightarrow \neg[(a(x) \wedge a(y)) \ \vee \ (b(x) \wedge b(y)) \ \vee \ (c(x) \wedge \ c(y))]] \\ \wedge & \forall x & [\exists y \ [a(y) \wedge x \sim y] \ \wedge \ \exists y \ [b(y) \wedge x \sim y] \wedge \exists y \ [c(y) \wedge x \sim y]] \end{array}$$

Les deux premières lignes de la formule assurent que la projection sur Σ est bien dans $a^*b^*c^*$. La seconde partie de la formule assure que chaque classe du mot contient au plus une occurrence de chaque lettre. Enfin, la dernière partie de la formule assure que chaque classe contient au moins une occurrence de chaque lettre.

Cette formule se généralise facilement à un alphabet $\Sigma = \{a_1, a_2, \ldots a_k\}$. On peut ainsi reconnaître tout langage $L_{a_1^n a_2^n \ldots a_k^n}$ par une formule de $FO^2(<, \sim)$.

3.2 Exemples de propriétés exprimables dans $EMSO^2(<, \sim)$

3.2.1 Il existe x tel que $x \sim x + 2$

Pour définir ce langage par une formule $EMSO^2(<, \sim)$, on utilise deux variables du second ordre X et Y. La première partie de la formule assure que tous les éléments de X sont dans la même classes. La seconde partie assure que Y est un singleton. La troisième partie exprime le fait que le prédécesseur de la position correspondant à Y est dans l'ensemble X. Enfin, la dernière partie assure que le successeur de la position correspondant à Y est dans l'ensemble X.

$$\begin{array}{rl} \exists X \exists Y & \forall x \in X \ \forall y \in X \ x \sim y \\ \wedge & \exists y \in Y \ \forall x \in Y \ x = y \\ \wedge & \exists x \in X \ \forall y \ [x < y \ \rightarrow \ \exists x \in Y \ x \leq y] \\ \wedge & \exists x \in X \ \forall y \ [y < x \ \rightarrow \ \exists x \in Y \ y \leq x] \end{array}$$

On remarque que cette propriété n'est pas exprimable par une formule $FO^2(<, \sim, +1)$.

3.3 Exemples de propriétés exprimables dans $FO^2(<, \sim, +1)$

3.3.1 Les langages réguliers

Le langage régulier :
Soit $\mathcal{L}$ un langage régulier de mots de Σ^*.

Soit $\mathcal{A}_\mathcal{L}$ un automate déterministe complet qui reconnaît ce langage. On note q_0 son état initial.

On construit une formule qui définit les mots de $(\Sigma \times D)^*$ dont la projection sur Σ est dans le langage $\mathcal{L}$ et dont les données codent le calcul de l'automate sur le mot comme il est expliqué dans la suite.

Quelques définitions préliminaires :

- On numérote les états de l'automate. Soit n le nombre d'états.

- On définit la fonction de codage des états de la manière suivante :

$$\text{codage}(q) = 0.0.un^1(q)$$

  où $un^1(q)$ est obtenu à partir de $un(q) = 0^q.1^{n-q}$ en intercalant un bit 1 entre chaque bit de $un(q)$

- On définit une fonction étendue de transitions :

$$\text{Transition} : \Sigma^{2n+2} \times Q \to Q$$

  Cette fonction associe à un mot u de taille $2n+2$ et un état q, l'état atteint dans l'automate à partir de q en lisant le mot u.

- On définit pour $q \in Q$ et $k \leq 2n+2$, l'ensemble L_q^k comme étant l'ensemble des mots de longueur k acceptés par l'automate à partir de l'état q

- On dira qu'une position d'un mot avec données code le bit 1 ssi elle et la position suivante du mot sont dans la même classe. On dira au contraire qu'une position code le bit 0 ssi elle et la position suivante du mot sont dans des classes différentes. La dernière position du mot, qui n'a pas de position successeur, codera par défaut le bit 0.

- Dans les formules, on utilisera les notations suivantes : pour b un bit donné, $\sim_b$ équivaut à $\nsim$ si $b = 0$ ou à $\sim$ si $b = 1$.

Le langage avec données :
On définit le langage $\mathcal{L}_D \subseteq (\Sigma \times D)^*$ comme étant l'ensemble des mots w qui vérifient :

- $\Pi_\Sigma(w) \in L$
- Les données codent le calcul de l'automate par blocs de taille $2n+2$.
  Soit $q_0.q_1.q_2...q_m$ la suite des états visités par l'automate lors de la lecture du mot w (m étant la longueur de w). Les données de w codent, selon la longueur du mot, la suite de bits :

$$codage(q_{2n+2}), codage(q_{4n+4}), \ \ldots \ codage(q_{k.(2n+2)}), 0^{m \bmod (2n+2)}$$

Afin de toujours avoir un bloc de 0 à la fin du mot, si $m = 0 \bmod (2n+2)$, le dernier bloc de taille $2n+2$ code 0^{2n+2} plutôt que de coder un état.

Les formules :

On définit maintenant des formules auxiliaires pour tout mot $v \in \Sigma^*$:

– A partir de la position x, le mot est étiqueté sur Σ par le mot $v = a_1 \cdots a_k$:

Si v est de longueur paire :

$$\begin{aligned}\mathbf{v(x)} := \quad a_1(x) \quad & \wedge \exists y[y = x+1 \wedge a_2(y) \\ & \wedge \exists x[x = y+1 \wedge a_3(x) \\ & \quad \wedge \cdots \wedge \exists y[y = x+1 \wedge a_k(y)]]]\end{aligned}$$

Si v est de longueur impaire :

$$\begin{aligned}\mathbf{v(x)} := \quad a_1(x) \quad & \wedge \exists y[y = x+1 \wedge a_2(y) \\ & \wedge \exists x[x = y+1 \wedge a_3(x) \\ & \quad \wedge \cdots \wedge \exists x[x = y+1 \wedge a_k(x)]]]\end{aligned}$$

– A partir de la position x inclue, les données des k positions suivantes du mot codent la suite de bit $b_1 \cdots b_k \in \{0,1\}^*$.

$$\begin{array}{ll} k = 1 & \mathbf{D_0}(x) := \ \exists y\ [y = x+1 \wedge\ x \nsim y]\ \vee\ \max(x) \\ k = 1 & \mathbf{D_1}(x) := \ \exists y\ [y = x+1\ \wedge x \sim y] \\ & \\ k = k+1 & \mathbf{D_{b_1 \ldots . b_{k+1}}}(x) := \ \exists y\ [y = x+1\ \wedge\ x \sim_{b_1} y\ \wedge\ D_{b_2 \ldots . b_{k+1}}(y)] \end{array}$$

– A partir de la position x exclue, les données des k positions précédentes du mot codent la suite de bit $b_1 \cdots b_k \in \{0,1\}^*$

$$\begin{array}{ll} k = 1 & \mathbf{G_{b_1}}(x) := \ \exists y\ [y = x-1\ \wedge\ x \sim_{b_1} y] \\ k = k+1 & \mathbf{G_{b_1 \ldots . b_{k+1}}}(x) := \ \exists y\ [y = x-1\ x \sim_{b_{k+1}} y\ \wedge\ G_{b_1 \cdots b_k}(y)] \end{array}$$

– La position x marque le début de la partie finale du mot (le bloc de 0)

$$\begin{array}{rcl} \mathbf{Final}(x) & := & [G_{1.1}(x)\ \vee\ G_{1.0}(x)\ \vee\ \min(x)] \\ & \wedge & \forall y\ [x \leq y\ \rightarrow\ D_0(y)] \\ & \wedge & \bigvee_{k \leq 2n+1} \max(x+k) \end{array}$$

La première ligne assure que la position x succède à un bloc du type "$1.-$" ou que c'est le début du mot. La seconde assure qu'à partir de x inclue, les données codent uniquement des 0. La dernière partie assure que le bloc final de 0 n'est pas plus long qu'un bloc d'état.

– les données du mot w codent un mot de la forme : $\{\text{codage(q)} \mid 1 \leq q \leq n\}^*.0^{\leq 2n+2}$

$$\begin{array}{rcl} \mathbf{Donnees}_{\mathcal{A}_\mathcal{L}} & := & \exists x\ [\min(x)]\ \wedge [\text{Final}(x)\ \vee\ D_{0.0.1}(x)] \\ & \wedge & \exists x\ \text{Final}(x) \\ & \wedge & [\forall x\ D_{0.0.1}(x) \rightarrow\ \bigvee_{q \in Q} D_{\text{codage}(q)}(x)] \\ & \wedge & [\forall x\ \bigvee_{q \in Q} G_{\text{codage}(q)}(x)\ \rightarrow\ [D_{0.0.1}(x)\ \vee\ \text{Final}(x)]] \end{array}$$

La première ligne assure que le mot commence par un bloc qui code un état ou est de longueur trop petite et code dans ce cas uniquement des 0. La seconde ligne assure que le mot finit par une suite de 0 de longueur inférieure ou égale à $2n+2$. La suivante exprime que la suite de bit "0.0.1" correspond toujours au début d'un bloc d'état. La dernière ligne assure qu'un bloc d'état est suivi d'un bloc état ou de la fin du mot.

– Reste à s'assurer que l'on code bien le calcul de l'automate et que celui-ci est acceptant :

$$\begin{array}{lll}\mathbf{Calcul}_{\mathcal{A}_\mathcal{L}} & := & \forall x\ [\min(x) \wedge \mathbf{Final}(x)] \rightarrow \bigvee_{k=1..2n+1}[\max(x+k)\ \wedge \bigvee_{u \in L_{q_0}^{k+1}} \mathbf{u}(x)] \\ & \wedge & \forall x\ [\min(x) \wedge D_{0.0.1}(x)] \rightarrow \bigvee_{u \in \Sigma^{2n+2}}[\mathbf{u}(x)\ \wedge\ D_{\text{codage(Transition}(u,q_0))}(x)] \\ & \wedge & \forall x\ [G_{\text{codage}(q)}(x)\ \wedge\ D_{0.0.1}(x)] \rightarrow \bigvee_{u \in \Sigma^{2n+2}}[\mathbf{u}(x)\ \wedge\ D_{\text{codage(Transition}(u,q))}(x)] \\ & \wedge & \forall x\ [G_{\text{codage}(q)}(x)\ \wedge\ \mathbf{Final}(x)] \rightarrow \bigvee_{k=1..2n+1}[\max(x+k)\ \wedge \bigvee_{u \in L_q^{k+1}} \mathbf{u}(x)]\end{array}$$

La première ligne traite le cas où le mot est de longueur inférieure à $2n+2$. La seconde ligne et les suivantes traitent du cas contraire. La seconde ligne assure que le premier bloc code bien l'état atteint à partir de l'état initial. La troisième assure la cohérence des transitions entre blocs. Enfin la dernière assure la cohérence entre le dernier bloc et la fin du mot et vérifie que le mot est bien accepté.

Finalement :

$$w \in \mathcal{L}_D \text{ ssi } w \models \mathbf{Donnees}_{\mathcal{A}_\mathcal{L}}\ \wedge\ \mathbf{Calcul}_{\mathcal{A}_\mathcal{L}}$$

On peut donc avec cette méthode reconnaître n'importe quel langage régulier à condition de l'augmenter avec des données judicieusement placées.

3.4 Exemples de propriétés exprimables dans $EMSO^2(<, \sim, +1)$

3.4.1 Toutes les positions paires sont la même classe

On considère que la première position du mot est 1, donc impaire.

$$\begin{array}{ll}\exists X \exists Y & \forall x \in Y\ \forall y \in Y\ x \sim y \\ & \exists x \in X\ min(x) \\ & \forall x \in X\ [\exists y = x+1] \rightarrow y \in Y \\ & \forall x \in Y\ [\exists y = x+1] \rightarrow y \in X\end{array}$$

La première partie de la formule exprime que tous les éléments de Y sont dans la même classe. La seconde partie exprime que la première position du mot est dans l'ensemble X. Les deux dernières parties expriment que tout successeur d'un élément de X est dans Y et que tout successeur d'un élément de Y est dans X.

3.4.2 Compter modulo k

$k = 2$: on peut, comme dans l'exemple précédent, distinguer l'ensemble des positions paires et l'ensemble des positions impaires d'un mot.

De la même façon, avec k variables du second ordre on peut distinguer les différentes groupes de positions modulo k.
La variable X_i représente l'ensemble des positions i modulo k. On commence par exprimer le fait que les variables du second ordre correspondent à une partition de l'ensemble des positions du mot. Puis on initialise les ensembles en disant que la première position appartient à l'ensemble X_1. Enfin, on répartit les positions une par une à partir de la première dans les différents ensembles.

$$\begin{array}{ll} \exists X_0 \exists X_1 \ldots \exists X_{k-1} & \forall x \ \ [x \in X_0 \ \vee \ x \in X_1 \ldots \ \vee \ x \in X_{k-1}] \\ & \forall x \ \ \neg \bigvee_{i \neq j} [x \in X_i \ \wedge \ x \in X_j] \\ & \exists x \ \ [\min(x)] \wedge [x \in X_1] \\ & \forall x \in X_0 \quad [\exists y = x+1] \rightarrow \ y \in X_1 \\ & \forall x \in X_1 \quad [\exists y = x+1] \rightarrow \ y \in X_2 \\ & \vdots \\ & \forall x \in X_{k-1} \ [\exists y = x+1] \rightarrow \ y \in X_0 \end{array}$$

4 Résultats d'indécidabilité : satisfaisabilité de FO^3 et vide des automates à deux registres

Les exemples précédents nous permettent de voir que deux variables du premier ordre suffisent pour définir des langages non réguliers et même non hors-contextes. Dans cette partie, on montre que l'utilisation de trois variables du premier ordre rend les logiques considérées indécidables. En ce qui concerne les automates, deux registres suffisent aussi à rendre le problème du vide indécidable.

4.1 Indécidabilité du vide pour $FO^3(<, \sim)$

On réduit le Problème de Correspondance de Post (PCP) au problème de satisfaisabilité de $FO^3(<, \sim, +1) = FO^3(<, \sim)$.

Le Problème de Correspondance de Post :

Donnée : $(u_i, v_i)_{(i=1 \cdots k)}$ où u_i, $v_i \in \Sigma^*$

Question : Existe-t-il une suite finie non vide d'indices $i_0 \cdots i_n$ telle que $u_{i_0} u_{i_1} \cdots u_{i_n} = v_{i_0} v_{i_1} \cdots v_{i_n}$?

Le codage :

Etant donné une instance de PCP, on considère l'alphabet $\Sigma' = \Sigma \cup \overline{\Sigma}$.
On construit une formule $FO^3(<, \sim, +1)$ qui définit les mots w de $(\Sigma' \times D)^*$ de la forme :

- $\Pi_{\Sigma'}(w) \in \{u_i\overline{v_i} \mid 1 \leq i \leq k\}^+$ c-à-d les lettres du mots w représentent une suite finie non vide de paires de PCP dont on a écrit la seconde partie v_i de la paire sur l'alphabet barré $\overline{\Sigma}$.
- il existe $w_0 \in \Sigma^*$ tel que $\overline{\Pi_\Sigma(w)} = \Pi_{\overline{\Sigma}}(w) = \overline{w_0}$
- Les données sont réparties de la manière suivante : à chaque position de w_0 est associée une donnée différente qui étiquette la position correspondante dans la partie Σ [resp $\overline{\Sigma}$] du mot.

La formule

Tout d'abord quelques formules intermédiaires :

- La position x est étiquetée par une lettre de Σ :
  $\mathbf{\Sigma(x)} := \bigvee_{a \in \Sigma} a(x)$
- La position x est étiquetée par une lettre de $\overline{\Sigma}$:
  $\overline{\mathbf{\Sigma}}\mathbf{(x)} := \bigvee_{a \in \overline{\Sigma}} a(x)$
- A partir de la position x, le mot est étiqueté sur Σ' par le mot $v = a_0 a_1 \cdots a_n$. On a déjà défini une formule qui vérifie cette propriété dans les exemples précédents (pour coder le langage régulier).

- Le mot est bien dans $\{u_i\overline{v_i} \mid 1 \leq i \leq k\}^+$:

$$\begin{aligned}\textbf{Paires} := \; & \exists x \; [\; \min(x) \wedge \textstyle\bigvee_{(i=1..k), a \in \Sigma} u_i\overline{v_i}a(x) \;\; \vee \max(x + |u_i v_i| - 1)] \\ & \wedge \forall x \forall y \;\; [y = x + 1 \wedge \overline{\Sigma}(x) \wedge \Sigma(y)] \rightarrow \\ & \qquad \left[\bigvee_{i=1..k, a \in \Sigma} u_i\overline{v_i}a(y)\right] \; \vee \; \left[\bigvee_{i=1..k} u_i\overline{v_i}(y) \;\wedge\; \max(x + |u_i v_i| - 1)\right]\end{aligned}$$

- Toutes les classes du mot contiennent exactement deux éléments, une position étiquetée par une lettre de Σ et une autre étiquetée par la lettre correspondante dans $\overline{\Sigma}$.

$$\begin{aligned}\textbf{Shuffle} := \; & \forall x \; \exists y \; [x \neq y \;\wedge\; x \sim y] \\ \wedge \; & \forall x \forall y \;\; [x \sim y \;\wedge\; x \neq y] \rightarrow [\textstyle\bigvee_{a \in \Sigma}[a(x) \wedge \overline{a}(y)] \vee [a(y) \wedge \overline{a}(x)]]\end{aligned}$$

- L'ordre d'apparition des données sur l'ensemble des positions du mot étiquetées par des lettres de Σ est le même que sur l'ensemble des positions étiquetées par des lettres de $\overline{\Sigma}$:

$$\begin{aligned}\textbf{PairesOrdonnees} := \forall x \forall y \;\; & [\Sigma(x) \wedge \Sigma(y) \wedge x < y] \\ & \rightarrow [\exists z \; \overline{\Sigma}(z) \wedge \;(x \sim z)\; \wedge \exists x \; [z < x \wedge \overline{\Sigma}(x) \wedge x \sim y]]\end{aligned}$$

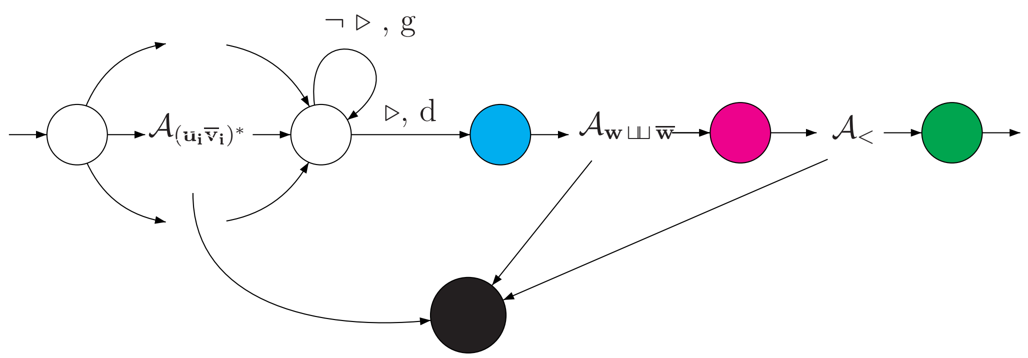
La formule générale qui reconnaît le langage précédemment défini :

$$\mathbf{PCP_{(u_i,v_i)}} := \mathbf{Paires} \wedge \mathbf{Shuffle} \wedge \mathbf{PairesOrdonnees}$$

Le problème PCP a une solution ssi le langage est non vide (c-à-d la formule est satisfaisable). Donc PCP étant indécidable, Sat-$FO^3(<, \sim)$ l'est aussi.

4.2 Indécidabilité des automates bi-directionnels à deux registres

On montre facilement de la même manière que le vide des automates à deux registres bi-directionnels est indécidable. Etant donné une instance du problème PCP, on construit un automate qui reconnaît le même langage que la formule précédente.
L'automate est de la forme suivante :

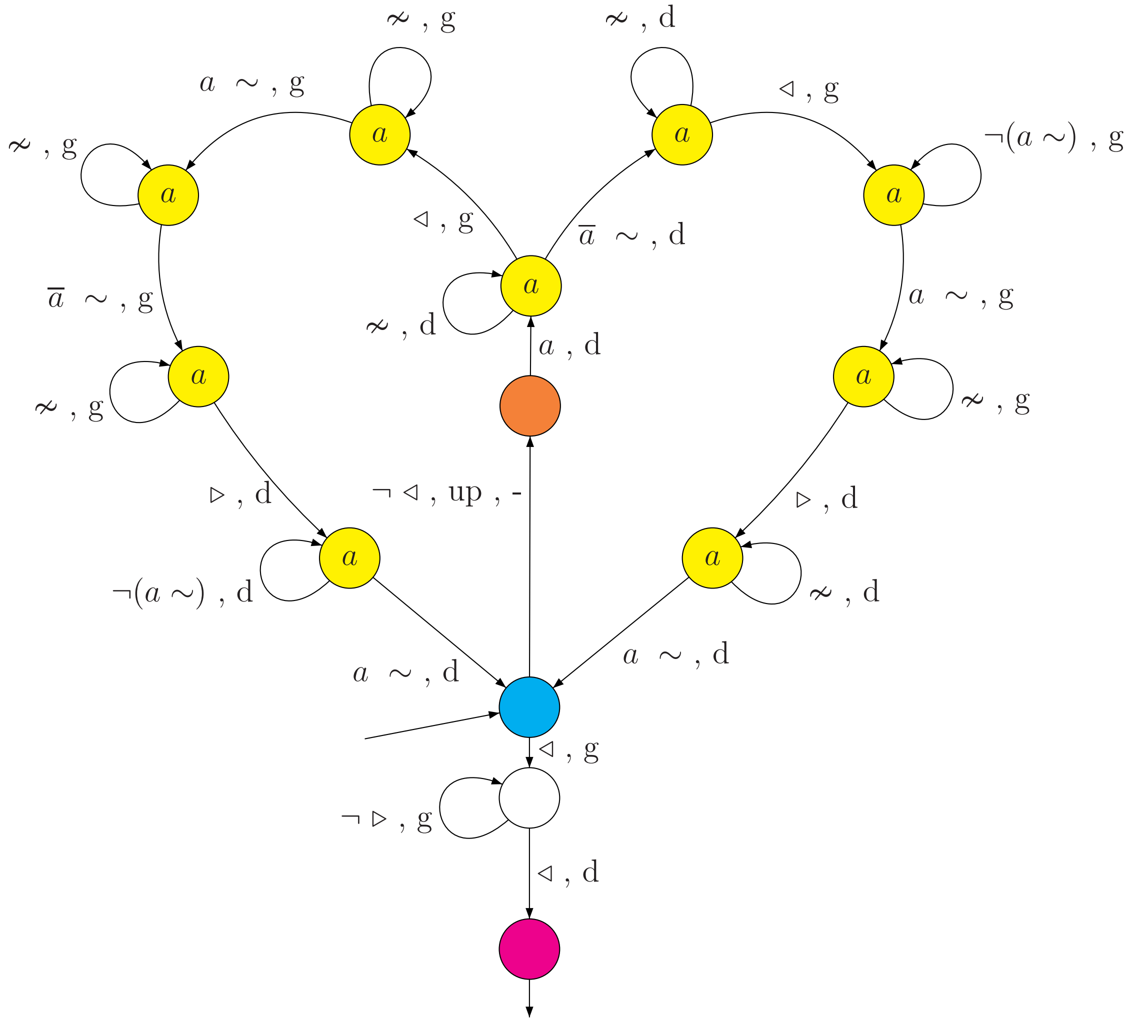


La première partie, $\mathcal{A}_{(u_i\overline{v}_i)^*}$, correspond à un automate classique (sans donnée) qui reconnaît le langage $\{u_i.\overline{v_i} \mid 1 \leq i \leq k\}^*$. Une fois le mot accepté par ce premier automate, on revient au début du mot et on vérifie que toutes les classes du mot contiennent exactement deux éléments, une position étiquetée par une lettre de Σ et une autre étiquetée par la lettre correspondante dans $\overline{\Sigma}$. On utilise pour cela un second automate, noté $\mathcal{A}_{w \sqcup\!\sqcup \overline{w}}$, qui accepte le mot en terminant son calcul au début du mot. Enfin la troisième partie, correspondant à l'automate noté $\mathcal{A}_<$, vérifie que les deux projections coïncident complètement.

L'automate $\mathcal{A}_{\mathbf{w} \sqcup\!\sqcup \overline{\mathbf{w}}}$

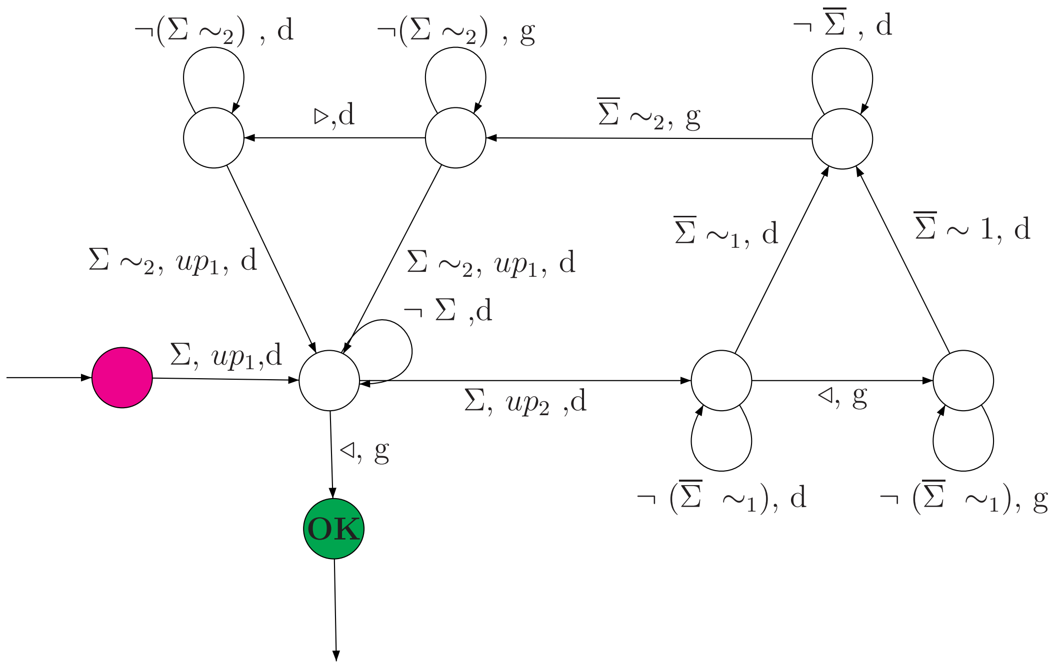
Pour vérifier que toutes les classes du mot contiennent exactement deux éléments, une position étiquetée par une lettre de Σ et une autre étiquetée par la lettre correspondante dans $\overline{\Sigma}$, on utilise un automate avec un registre. L'automate procède de la manière suivante : il parcourt le mot de gauche à droite et à chaque position, il fait une mise à jour du registre puis vérifie à droite puis à gauche qu'il y a exactement une seule autre position dans le mot qui a la même donnée et que cette position est étiquetée par la lettre dans l'alphabet complémentaire. (L'automate peut revenir à la position de départ en recherchant à gauche la donnée stockée dans le registre.)

L'automate a une forme rosace avec un "pétale" par lettre $a \in \Sigma \cup \overline{\Sigma}$. L'automate avec le pétale correspondant à la lettre a (représenté par les états étiquetés a) est le suivant :



L'automate $\mathcal{A}_<$

Une fois ces deux vérifications faites, il reste à contrôler que l'ordre d'apparition des données sur l'ensemble des positions du mot étiquetées par des lettres de Σ est le même que sur l'ensemble des positions étiquetées par des lettres de $\overline{\Sigma}$.
L'automate suivant assure cette propriété en vérifiant la chose suivante : à tous les couples (x_1, x_2) de positions successives dans la projection du mot sur l'alphabet Σ correspond un couple (y_1, y_2) de positions successives dans la projection du mot sur l'alphabet Σ' tel que x_1 et y_1 sont dans la même classe et x_2 et y_2 sont dans la même classe.



4.3 Conclusion

Etant donné que le vide de $FO^3(<, \sim)$ est indécidable, on s'intéresse, par la suite, aux formules de la logique du premier ordre avec au plus deux variables. Dans ce cas, on ne peut exprimer le successeur avec l'ordre, il faut donc distinguer $FO^2(<, \sim)$ et $FO^2(<, \sim, +1)$. En ce qui concerne les automates à registres, il faut considérer au plus un registre ou trouver un moyen de brider le pouvoir des automates.

4.4 FO^2 sans donnée

Sans donnée, sur un alphabet fini A la classe des langages $\mathrm{FO}^2(<)$ a de nombreuses caractérisations. Elles sont regroupées dans les articles suivants : [TW98] [STV01] [EVW97].

Définitions

- Σ_2
  Formules du type : $\exists x_0 ... \exists x_{k-1} \forall y_0 ... \forall y_{l-1}\ \varphi(x_0, ...x_{k-1}, y_0, ...y_{l-1})$ où $\varphi(x_0, ...x_{k-1}, y_0, ...y_{l-1})$ est une formule sans quantificateurs.
- Π_2
  Formules du type : $\forall x_0 ... \forall x_{k-1}\ \exists y_0 ... \exists y_{l-1} \varphi(x_0, ...x_{k-1}, y_0, ...y_{l-1})$ où $\varphi(x_0, ...x_{k-1}, y_0, ...y_{l-1})$ est une formule sans quantificateurs.

- TL(F,F^-) logique temporelle munie des opérateurs "un jour dans le futur " et "un jour dans le passé"

- UL classe des langages non-ambigus :
  Un langage de la forme $A_0^*.a_1.A_1^*...a_k.A_k^*$ où $A_i \subseteq A$ est dit non-ambigu ssi il existe une seule et unique façon de décomposer un mot u du langage de la manière suivante : $u = u_0.a_1.u_1.a_2...a_k.u_k$ avec $u_i \in A_i^*$
  Un langage est dans la classe UL s'il peut s'écrire comme une union finie disjointe de langages non-ambigus de la forme $A_0^*.a_1.A_1^*...a_k.A_k^*$.

- DA est la variété des semi-groupes finis vérifiant l'égalité :

$$(xyz)^\omega y(xyz)^\omega = (xyz)^\omega$$

  (où x^ω représentant la puissance de x qui est idempotente)

- On définit la relation de congruence $\equiv_m$ sur les mots de la manière suivante :
  - Pour $n = 0$, on a $u \equiv_0 v$ pour tous les mots u et v
  - par induction sur $n + |\text{alphabet}(u)|$, on définit $u \equiv_n v$ ssi :
    - alphabet(u)=alphabet(v)
    - pour tout $a \in$ alphabet(u)
      si $u = u_0.a.u_1$ et $v = v_0.a.v_1$ où les a distingués correspondent au premier du mot
      alors $u_0 \equiv_n v_0$ et $u_1 \equiv_{n-1} v_1$
    - pour tout $a \in$ alphabet(u)
      si $u = u_0.a.u_1$ et $v = v_0.a.v_1$ où les a distingués correspondent au dernier du mot
      alors $u_0 \equiv_{n-1} v_0$ et $u_1 \equiv_n v_1$

  On dit d'un langage de mots écrit sur l'alphabet A qu'il est n-testable ssi il est une union finie de classes de $\equiv_n$.

- po-2-DFA : Classe des automates déterministes bi-directionnels dont les états sont partiellement ordonnés. L'ordre partiel sur les états $\prec$ impose que lorsque l'automate est dans un état q, il peut soit, rester dans cet état q, soit passer dans un état $q' \prec q$. Si on regarde le graphe formé par les états de l'automate, il ne contient pas de cycle de longueur supérieure à 1.

- Langage Turtle : Pour a une lettre de l'alphabet A, et δ une direction (droite, gauche), le couple (a, δ) représente une instruction Turtle qui signifie "Aller dans la direction δ jusqu'a trouver la lettre a".
  On appelle programme Turtle une suite finie de telles instructions. Un mot $w \in A^*$ est accepté par un programme turtle si la suite d'instructions correspondante arrive à s'exécuter sans erreur.
  Un langage Turtle est un langage qui est reconnu par un programme Turtle.

Caractérisations de $FO^2(<)$

$$
\begin{aligned}
\mathrm{FO}^2(<) &= \mathrm{TL}(F, F^-) \\
&= \mathrm{UL} \\
&= \Sigma_2 \cap \Pi_2(<) \\
&= \mathrm{DA} \\
&= \text{po-2-DFA} \\
&= n - \text{testable pour un certain } n \\
&= \text{Combinaisons booléennes de langages Turtle}
\end{aligned}
$$

Passage d'un automate po-2-DFA à une formule $FO^2(<)$

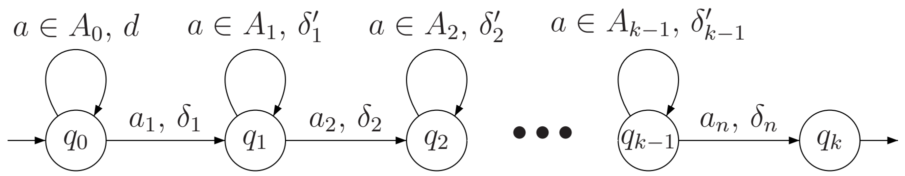
Afin de mieux comprendre le lien entre les automates et les formules dans le cas avec données, on aimerait déjà comprendre ce lien dans le cas sans donnée. Pour cela on cherche à avoir une preuve directe de l'équivalence formule-automate (chose qui n'a pas été faite). Malheureusement, si on arrive à passer directement de l'automate à la formule, nous n'avons pas trouvé de solution facile directe pour la transformation inverse.

Comme dit précédemment, le graphe des états d'un automate partiellement ordonné n'a pas d'autre cycle que les boucles simples sur les états. Dans un calcul, une fois que l'on est sorti d'un état, on ne pourra jamais y revenir.

On considère un calcul acceptant. On remarque que comme l'automate est déterministe, on ne peut changer de sens qu'en changeant d'état, car sinon le calcul boucle. On considère la suite des états correspondant à ce calcul sous la forme $q_0^{i_0}.q_1^{i_1}\cdots q_{k-1}^{i_{k-1}}q_k \in Q_f$. A cause de l'ordre partiel sur les états, les q_i sont tous différents. On remarque aussi que tout calcul de la forme $q_0^+.q_1^+\cdots q_{k-1}^+q_k \in Q_f$ est un calcul acceptant. Ces suites $(q_0, q_1\cdots q_{k-1}q_k \in Q_f)$ définissant des calculs acceptants sont en nombre fini.

L'idée de la transformation de l'automate en formule est de considérer chacune de ces suites définissant des calculs acceptants et de construire une formule qui définit les mots acceptés par le type de calcul correspondant. La formule générale sera bien sûr la disjonction des formules correspondantes sur tous les différents types de calcul acceptant de l'automate.

On considère le type de calcul correspondant à la suite $(q_0, q_1\cdots q_{k-1}q_k \in Q_f)$. Cela correspond au sous-automate suivant :



Les A_i représentent des ensembles de lettres de l'alphabet et les δ_i représentent des directions (d pour droite, g pour gauche). On remarque que la boucle sur l'état initial a forcément pour direction la droite et que l'automate étant déterministe, $a_i \notin A_{i-1}$.

On définit par induction la formule $\varphi_i(x)$ qui identifie l'endroit du calcul où l'automate lit la lettre a_i et fait la transition $q_{i-1} \to q_i$

- $i = 0$

$$\varphi_0(x) := a_1(x) \ \wedge \ \forall y < x \ A_0(y)$$

- $\vdots$
- $i = i + 1$ et $\delta_i' = d$

$$\begin{array}{rcl} \varphi_{i+1}(x) & := & a_{i+1}(x) \ \wedge \ \forall y < x \ [A_i(y) \ \vee \ [\exists x \ [\varphi_i(x) \ \wedge \ y < x]]] \\ & \wedge & \exists y < x \ \varphi_i(y) \end{array}$$

- $i = i + 1$ et $\delta_i' = g$

$$\begin{array}{rcl} \varphi_{i+1}(x) & := & a_{i+1}(x) \ \wedge \ \forall y > x \ [A_i(y) \ \vee \ [\exists x \ [\varphi_i(x) \ \wedge \ y > x]]] \\ & \wedge & \exists y > x \ \varphi_i(y) \end{array}$$

La formule correspondant au type de calcul définit par $(q_0, q_1 \cdots q_{k-1} q_k)$ est :

$$\varphi_{q_0 \cdots q_k} = \exists x \ \varphi_k(x)$$

La formule générale est :

$$\varphi = \bigvee_{(q_0, q_1 \cdots q_{k-1} q_k) acceptant} \varphi_{(q_0, q_1 \cdots q_{k-1} q_k)}$$

Avec le successeur : caractérisations de $FO^2(<, +1)$

Si on rajoute le successeur, on a des caractéristiques analogues. Les résultats analogues sont les suivants :

$$\begin{array}{rcl} \mathrm{FO}^2(<, +1) & = & \mathrm{TL}(F, X, F^-, X^-) \text{ où } X \text{ correspond à l'opérateur temporel "Next"} \\ & = & \Sigma_2 \cap \Pi_2(<, +1) \\ & = & \mathrm{DA} * \mathrm{D} \end{array}$$

D représente des semi-groupes finis vérifiant l'égalité $yx^\omega = x^\omega$. D représente les langages qui sont des unions finies de singletons et de langages de la forme $A^*.u$ où u est un mot de A^*.

4.4.1 Automates po-2-DFA$_1$

Puisque sans donnée, la logique du premier ordre avec deux variables correspond aux automates bi-directionnels partiellement ordonnés déterministes, on va chercher, dans notre cas avec données, à restreindre les automates en tenant compte de cette propriété "déterministes

partiellement ordonnés". Les automates uni-directionnels déterministes partiellement ordonnés ne sont pas suffisamment puissants. En effet, on ne peut en restant dans un état parcourir le mot que dans une seule direction (car sinon on boucle). Le nombre de changements de direction dans un calcul est limité par le nombre d'états de l'automate. Et la projection du langage sur l'alphabet fini est régulière.
On va donc considérer des automates bi-directionnels déterministes et, afin de garder un pouvoir d'expression important, partiellement ordonnés entre deux mises à jour.
Avec deux registres le vide des automates est indécidable. En effet, l'automate précédemment construit pour PCP est partiellement ordonné si on prend le soin de faire des mises à jour (inutile) dans l'automate qui reconnaît $\{u_i\overline{v_i} \mid i = 1 \cdots k\}^*$ afin que tout cycle contienne au moins une transition avec mise à jour du registre. On construit ensuite un ordre partiel sur les états.

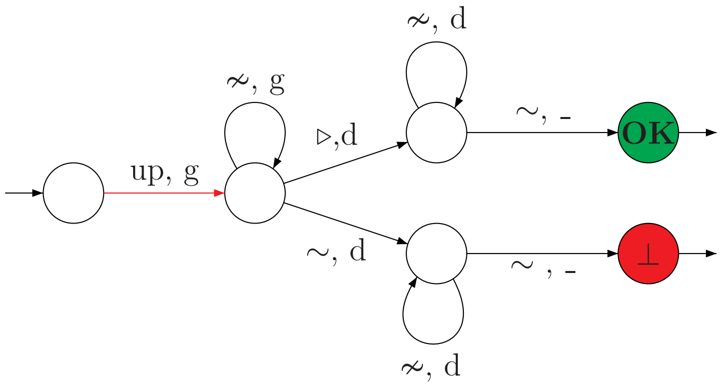
On va donc considérer la classe d'automate po-2-DFA$_1$ ie automates à un registre bi-directionnels déterministes partiellement ordonnés entre deux mises.

5 Exemples de propriétés exprimables dans po-2-DFA$_1$

Dans cette partie, on reprend les exemples donnés pour les différentes logiques et on regarde ce que l'on peut exprimer avec nos automates déterministes à un registre, partiellement ordonnés entre deux mises à jour.

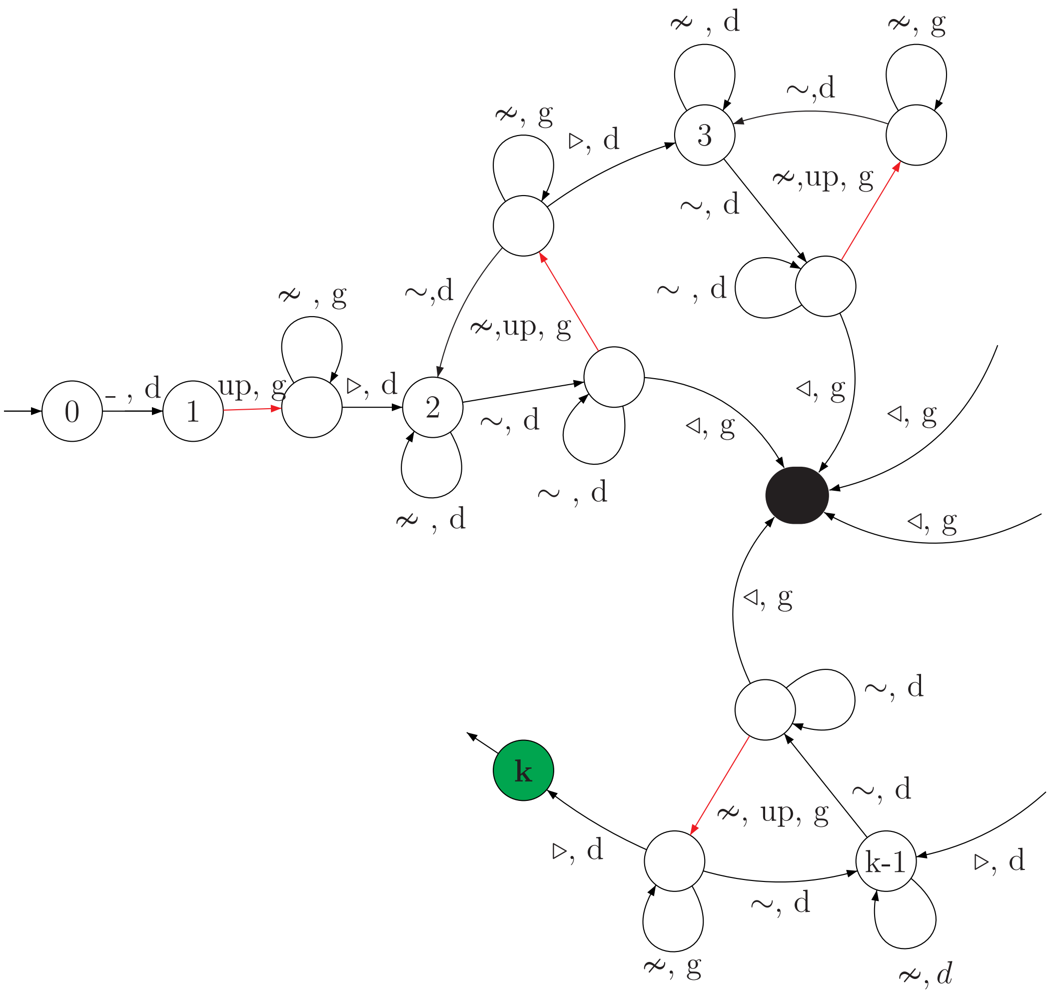
5.1 Premier

On construit un automate qui ayant la tête de lecture à la position x accepte le mot ssi x est le premier de sa classe. (L'automate termine son calcul dans la position x de départ). L'idée est de faire une mise à jour du registre puis de chercher à gauche un élément de la même classe. Si l'automate arrive au début du mot, la position x est bien la première de sa classe. L'automate revient à droite dans un état acceptant jusqu'à retrouver la position x (la première de la classe du registre). Si par contre l'automate rencontre à gauche un élément de la classe du registre, la position x n'est pas première de sa classe. L'automate revient à droite dans un état non acceptant jusqu'à retrouver la position x (la première de la classe du registre).



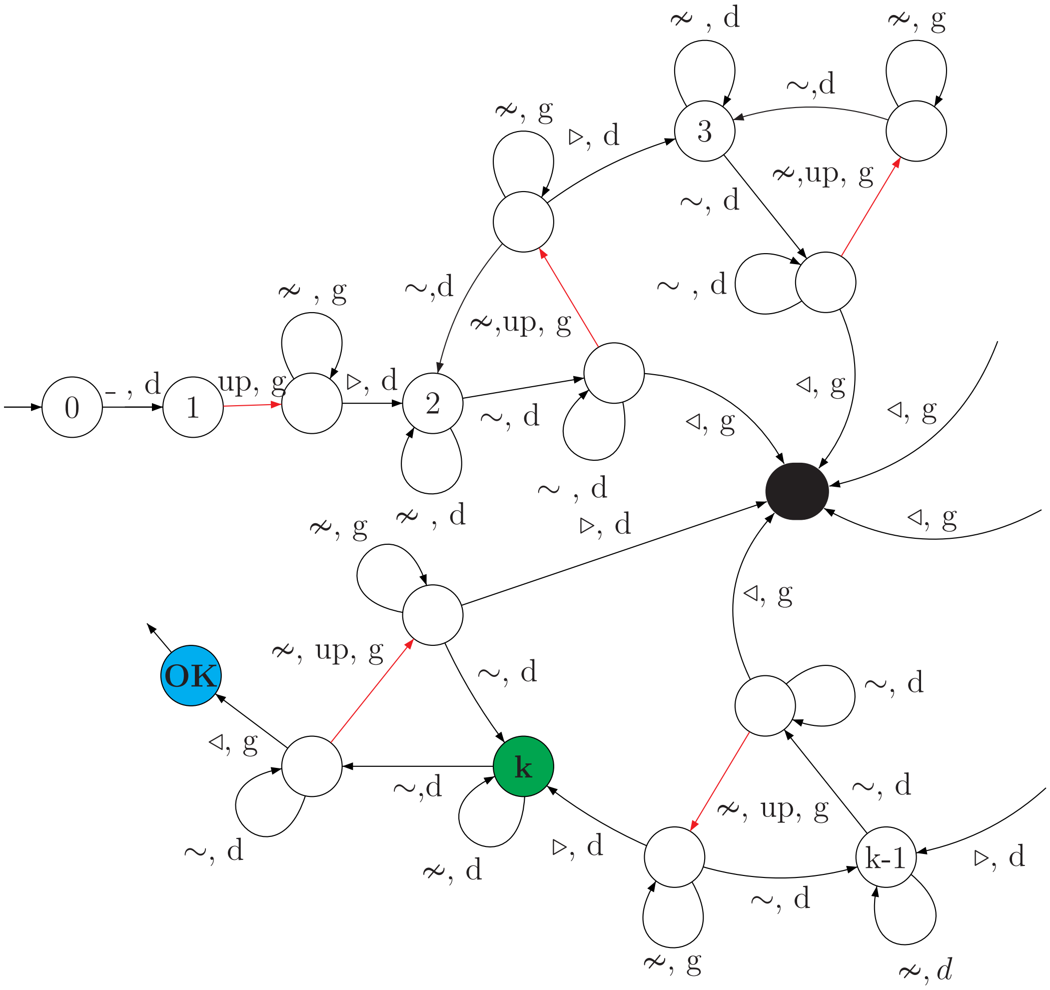
5.2 Au moins k classes

On construit un automate qui accepte les mots qui ont au moins k classes distinctes. L'idée est de parcourir le mot de droite a gauche en regardant si chaque position est première ou non de sa classe et en mémorisant dans les états combien de classes on a déjà rencontrées. Si on arrive à k classes, on accepte le mot. Si on arrive à la fin du mot avant d'avoir trouvé k classes, on rejette le mot.



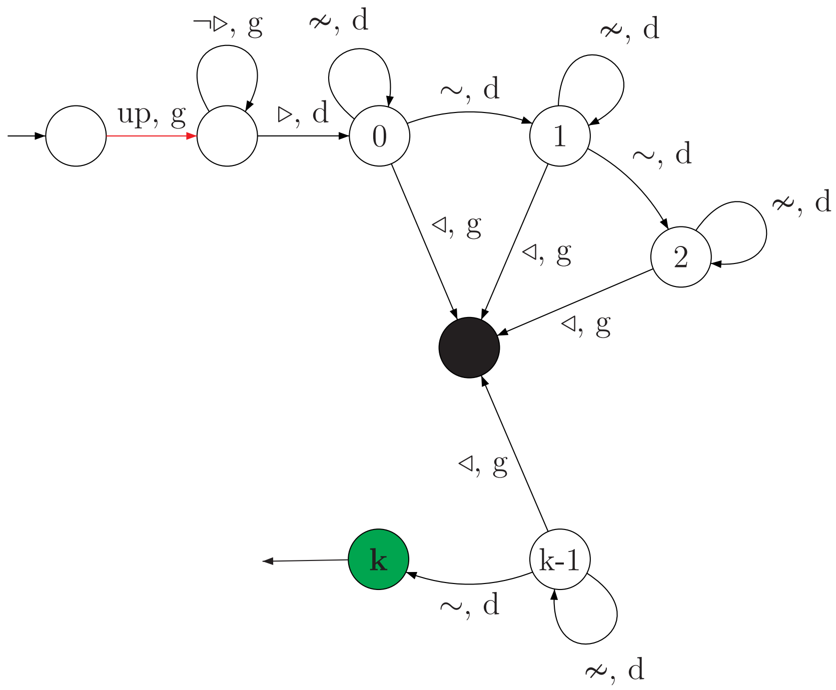
5.3 Il y a exactement k classes distinctes dans le mot

Il y en a au moins k et non il y en a au moins k+1. Comme l'automate précédent termine en début de mot, on peut construire un automate qui vérifie la propriété en mettant bout à bout l'automate qui vérifie qu'il y a au moins k classes et celui qui vérifie qu'il y en a au moins $k+1$ avec les états terminaux correctement choisis. Cette méthode permet aussi de construire un automate qui reconnaît les mots qui ont entre n et m classes distinctes.
On peut aussi modifier légèrement l'automate qui vérifie qu'il y a au moins k classes afin qu'il s'assure qu'il n'y en ait pas plus.



5.4 Au moins k éléments

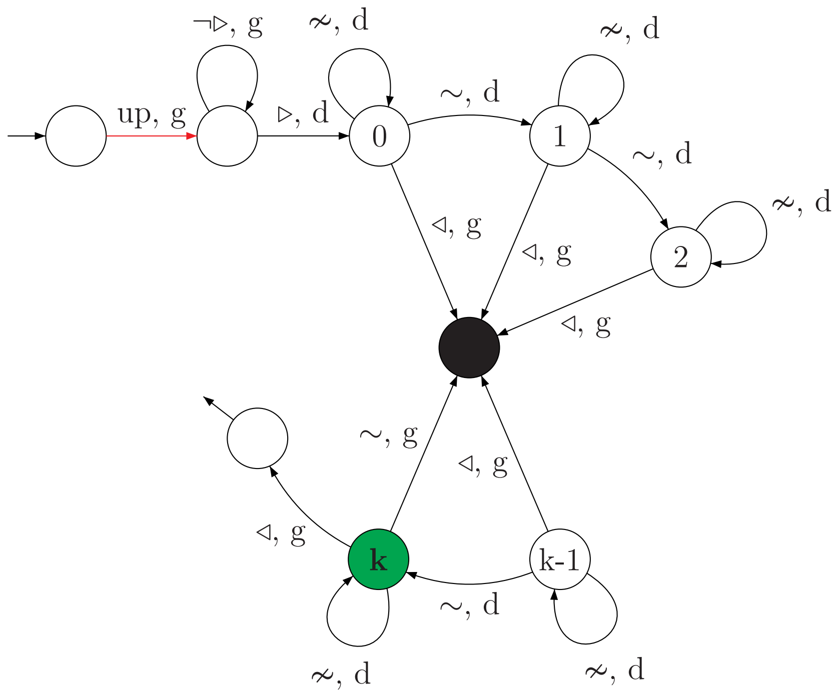
On a un automate qui ayant la tête de lecture à la position x vérifie si il y a au moins k éléments dans la classe de x.
L'automate fonctionne de la manière suivante : il stocke la donnée correspondante, revient au début du mot puis parcourt le mot de gauche à droite en comptant dans les états le nombre d'éléments de la classe.



On remarque que l'on peut aussi faire une version plus compliquée de l'automate qui garde en mémoire l'écart avec la position de départ afin de pouvoir terminer dans un état acceptant ou non avec la tête de lecture à la position x de départ.

5.5 Exactement k éléments

On veut un automate qui ayant la tête de lecture à la position x vérifie si il y a exactement k éléments dans la classe de x.
Comme pour les classes, on peut combiner les deux automates précédents pour vérifier qu'il y a au moins k éléments et qu'il n'y en a pas au moins $k+1$. On peut aussi modifier légèrement l'automate qui vérifie si il y a au moins k élément dans la classe de x pour obtenir l'automate suivant :



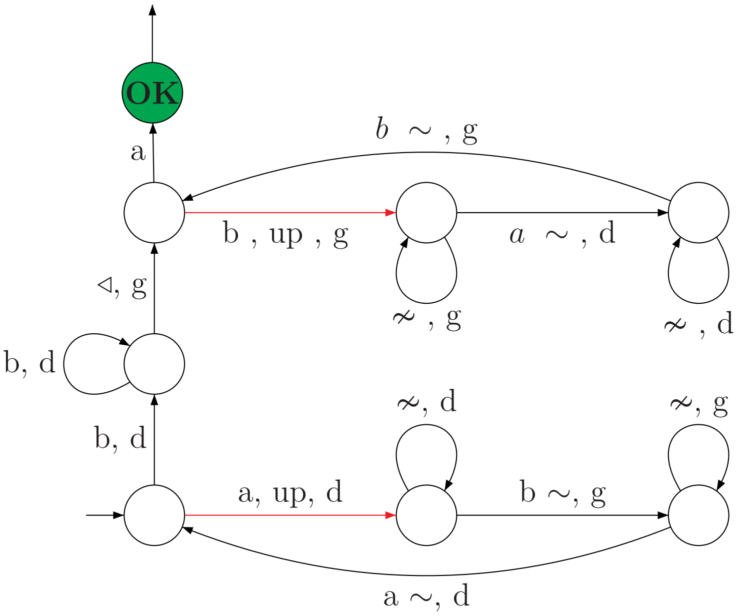
5.6 $\{\mathbf{a^n b^n} \mid \mathbf{n} \geq \mathbf{1}\}$

$\Sigma = \{a, b\}$

On appelle $L_{a^nb^n} \subset (\Sigma \times D)^*$, le langage des mots w tels que :

- $\Pi_\Sigma(w) = a^n b^n$ pour un certain $n \geq 1$
- Les données du mot w forment n classes différentes : chacune contenant exactement une position étiquetée a et une autre étiquetée b.

On reconnaît ce langage avec l'automate suivant :

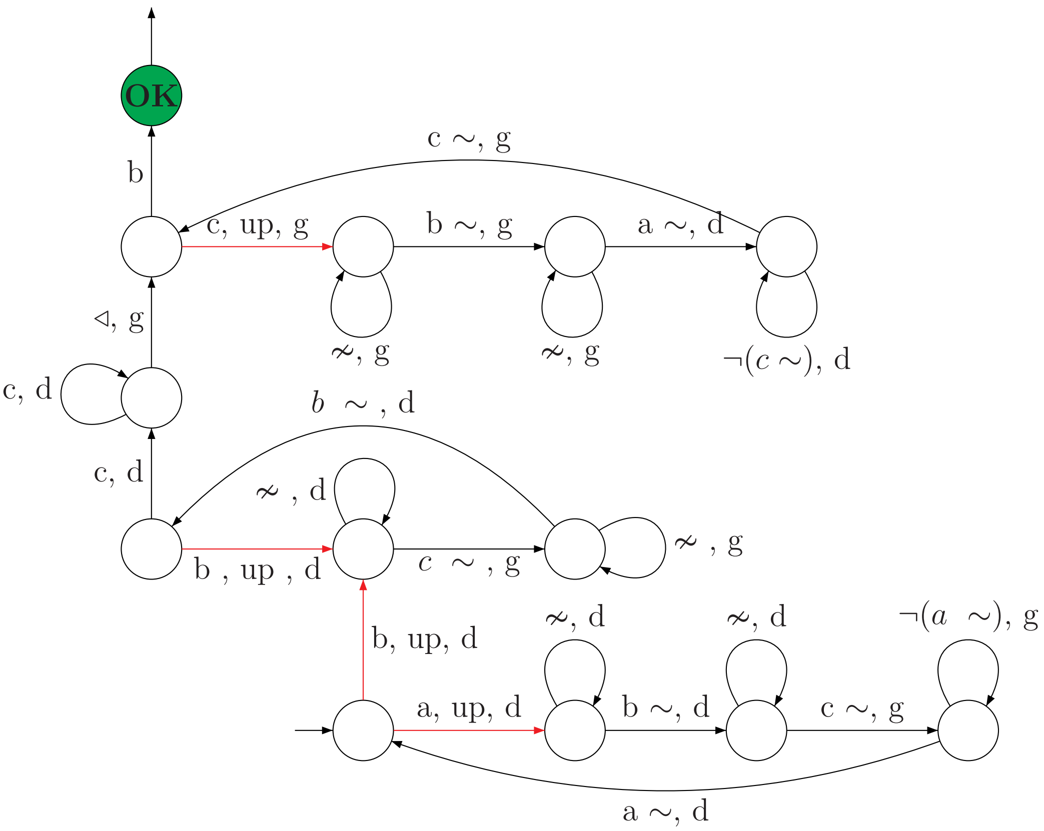


L'automate parcourt le mot de la manière suivante :
Au début, l'automate lit un a, met à jour le registre, cherche à droite la première lettre de la même classe et vérifie que c'est un b, puis revient au a (premier de la classe à gauche). Il passe ensuite à la position suivante et recommence. Si la position suivante est étiquetée b, l'automate change d'état et va à l'extrémité droite du mot en vérifiant qu'il ne rencontre que des b. Il met à jour le registre. Il cherche maintenant le premier élément de la même classe à sa gauche et vérifie que c'est un a. Il revient ensuite à la position suivante et ne recommence l'opération que si c'est un b. Lorsqu'il retrouve le dernier a, il accepte le mot

5.7 $\{\mathbf{a^n b^n c^n} \mid \mathbf{n} \geq \mathbf{1}\}$

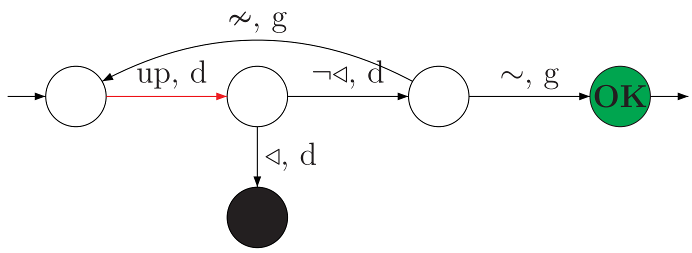
$\Sigma = \{a, b, c\}$
On considère $L_{a^n b^n c^n} \subset (\Sigma \times D)^*$, le langages des mots w tels que :
- $\Pi_\Sigma(w) \in a^n b^n {c^n}_n$
- Les données du mot w forment n classes différentes : chacune contenant exactement une position étiquetée a, une autre étiquetée b et une autre étiquetée c

On reconnaît ce langage avec l'automate suivant :



L'automate parcourt le mot de gauche à droite en regardant les a et les b de la manière suivante :
La première partie de l'automate vérifie, à partir du début, que pour chaque a les deux éléments suivant de sa classe à droite sont un b et un c et que le b apparaît avant le c. La seconde partie de l'automate qui commence lorsque l'automate en arrive au premier b vérifie que pour chaque b l'élément suivant de sa classe à droite est un c. Après avoir rencontré les a et les b l'automate rencontre le premier c, va à la fin du mot en vérifiant qu'il ne rencontre que des c et vérifie de droite à gauche que pour chaque c les deux éléments suivant qui apparaissent dans la même classe à gauche sont un b puis un c.

5.8 Il existe x tel que $x \sim x + 2$



5.9 Les réguliers

Soit R un langage régulier sur Σ^* . On note L_R, le plus grand ensemble de $(\Sigma \times D)^*$ dont la projection sur Σ^* est R.
On reconnaît un langage de cette forme avec un automate po-2-DFA1. En effet, on considère l'automate déterministe qui reconnaît la projection du langage et on rajoute des mises à jour (inutiles) afin de casser les cycles qui invalident la propriété "partiellement ordonné entre deux mises à jour".

5.10 Codage des réguliers

On sait reconnaître le codage précédemment décrit pour la logique. L'automate lit les blocs un à un. Il mémorise, dans les états, l'état de l'automate sans données codé par le bloc précédent et lit le bloc courant en mémorisant les lettres de l'alphabet fini et les bits codés par les données. A la fin du bloc il connait, avec les lettres, la transition qui doit avoir été effectuée et vérifie que les données codent bien l'état correspondant à cette transition. Il passe ensuite au bloc suivant en ne gardant en mémoire que l'état codé dans le bloc qu'il vient de lire.

6 Décidabilité et complexité pour la logique

Dans cette partie, on montre que la logique $FO^2(<, \sim)$ est décidable en ramenant la satisfaisabilité d'une formule $FO^2(<, \sim)$ à la satisfaisabilité d'une formule $FO^2(<)$. Pour cela on commence par donner une forme normale pour les formules $FO^2(<, \sim)$ qui a été définie pour les logiques FO^2 en général par Scott en 1962. On montre ensuite la décidabilité de $FO^2(<, \sim)$ puis on étudie la complexité du problème de satisfaisabilité ainsi traité.

6.1 Forme normale de Scott

Afin de montrer plus simplement la décidabilité de la logique $FO^2(<, \sim)$, il est utile de rappeler qu'il existe pour les formules logiques du premier ordre une forme normale simple ne mettant en jeu que des conjonctions de formules FO^2 en forme normale prenex n'utilisant que deux quantificateurs mais utilisant des prédicats unaires supplémentaires (un alphabet étendu).

L'idée est de construire récursivement la forme normale de l'intérieur vers l'extérieur de la formule de départ en faisant disparaître les quantificateurs un par un, et en rajoutant des prédicats d'arité 1 à l'alphabet ; prédicats que l'on peut alors définir en dehors de la formule générale. A la fin, on obtient une formule de la forme :

$$\forall x \forall y \ \chi_0(x,y) \wedge \bigwedge_{i=1}^{m} \forall x \exists y \ \chi_i(x,y)$$

Exemple : On considère l'alphabet Σ et la formule φ suivante :

$$\varphi := \forall x \ [\ a(x) \rightarrow [\ \exists y \ [[\ x < y \ \wedge \ b(y)] \rightarrow [\forall x \ y < x \rightarrow \overline{[\forall y \ y < x \rightarrow \ x \nsim y]} \ ]] \ ] \ ]$$

On commence par rajouter un prédicat $P_1(x)$ afin de supprimer la partie surlignée. On obtient un nouvel alphabet $\Sigma_1 = \Sigma \times \{0,1\}$ et une nouvelle formule :

$$\begin{array}{rl} \varphi_1 := & \forall x \ [\ a(x) \rightarrow [\ \exists y \ [[\ x < y \ \wedge \ b(y)] \rightarrow \overline{[\forall x \ y < x \rightarrow P_1(x)\ ]}] \ ] \ ] \\ \wedge & \forall x \ P_1(x) \leftrightarrow [\forall y \ y < x \rightarrow \ x \nsim y] \end{array}$$

On rajoute ensuite un prédicat $P_2(x)$ afin de supprimer la partie surlignée. On obtient un nouvel alphabet $\Sigma_2 = \Sigma \times \{0,1\}^2$ et une nouvelle formule :

$$\begin{array}{rl} \varphi_2 := & \forall x \ [\ a(x) \rightarrow [\ \exists y \ [[\ x < y \ \wedge \ b(y)] \rightarrow P_2(y)] \ ] \ ] \\ \wedge & \forall x \ P_1(x) \leftrightarrow [\forall y \ y < x \rightarrow \ x \nsim y] \\ \wedge & \forall x \ P_2(x) \leftrightarrow [\forall y \ x < y \rightarrow P_1(y)\ ] \end{array}$$

On a un nouvel alphabet $\Sigma' = \Sigma \times \{0,1\}^2$. On réarrange la formule et on obtient la formule $NF(\varphi)$:

$$\begin{array}{rl} NF(\varphi) := & \forall x \exists y \ [\ a(x) \rightarrow [[\ x < y \ \wedge \ b(y)] \rightarrow P_2(y)] \ ] \ ] \\ \wedge & \forall x \forall y \ [P_1(x) \leftrightarrow [\ y < x \rightarrow \ x \nsim y]] \ \wedge \ [P_2(x) \leftrightarrow [\ x < y \rightarrow P_1(y)\ ]] \end{array}$$

Plus formellement : Il existe une transformation $NF : FO^2 \rightarrow FO^2$ qui transforme toute formule φ de FO^2 en une formule $NF(\varphi)$ (sur un alphabet étendu) de la forme :

$$\forall x \forall y \ \chi_0(x,y) \wedge \bigwedge_{i=1}^{m} \forall x \exists y \ \chi_i(x,y)$$

où les formules χ_i sont des formules sans quantificateurs et telle que φ est satisfaisable *ssi* $NF(\varphi)$ est satisfaisable.

$NF(\varphi)$ et φ sont équivalentes au sens que tout modèle de $NF(\varphi)$ est un modèle de φ sur l'alphabet étendu et tout modèle de φ sur l'alphabet de départ peut-être complété en un modèle de $NF(\varphi)$. La transformation se fait en temps polynomial en la taille de φ et la taille de $NF(\varphi)$ est polynomiale en celle de φ. Par contre le nouvel alphabet est de taille exponentielle en la taille de φ.

6.2 $FO^2(<, \sim)$ est décidable

On sait que sans donnée la satisfaisabilité d'une formule $FO^2(<)$ ou $FO^2(<, +1)$ est décidable plus exactement NEXPTIME-complet avec un alphabet de la forme 2^{Prop} (où Prop est un ensemble fini de propositions atomiques) [EVW97].

Idée :

Pour montrer la décidabilité, l'idée est de ramener Sat-$FO^2(<, \sim)$ au problème sans donnée. Pour cela on montre que toute formule satisfaisable a un modèle dont le nombre de classes différentes est borné. Ainsi, en modifiant légèrement la formule et l'alphabet afin de coder les différentes classes dans les lettres, on montre que la formule est satisfaisable ssi une nouvelle formule sans donnée est elle même satisfaisable. Le problème Sat-$FO^2(<)$ sans donnée étant satisfaisable, Sat-$FO^2(<, \sim)$ l'est aussi .

Preuve :

On considère φ une formule de $FO^2(<, \sim)$.
D'après ce qui précède, on sait que cette formule a un modèle sur l'alphabet Σ ssi sa forme normale a un modèle sur un alphabet Σ'. On considère donc par la suite $NF(\varphi) := \forall x \forall y\ \chi_0(x, y) \wedge\ \bigwedge_{i=1}^{m} \forall x \exists y\ \chi_i(x, y)$ la forme normale de φ.

Construction d'un petit modèle :
Soit W un modèle de $NF(\varphi)$ sur l'alphabet Σ'.
On construit un nouveau mot W' à partir de W de la manière suivante : Pour chaque lettre a de l'alphabet Σ', on garde la classe d contenant le premier a dans w et la classe $d' \neq d$ qui contient le premier a du mot w qui n'est pas dans la classe d et symétriquement, on garde la classe d'' qui contient le dernier a du mot w et la classe qui contient le dernier a du mot w qui n'est pas dans la classe d''. On garde donc au plus $4|\Sigma'|$ classes. Montrons que le mot W' est toujours un modèle de la formule $NF(\varphi)$.

$W' \models \forall x \forall y\ \chi_0(x, y)$ **:**
Soit x et y deux positions dans le mots W'. Si on regarde ces deux positions dans le mot de départ W, celui-ci étant un modèle de $NF(\varphi)$, on a $W, x, y \models \chi_0(x, y)$. La clef réside dans le fait que χ_0 est sans quantificateur et donc ne porte que sur les lettres et la position de x par rapport à y et le fait que ces deux positions soient ou non dans la même classe. Comme ces propriétés ne changent pas entre W et W', alors on a $W', x, y \models \chi_0(x, y)$.

$W' \models \bigwedge_{i=1}^{m} \forall x \exists y\ \chi_i(x,y)$ **:**
Soit x une position dans W', on regarde la position correspondante dans W. Comme $W \models \forall x \exists y\ \chi_i(x,y)$, on a une position y dans W telle que $W, x, y \models \chi_i(x,y)$.
Si x et y sont dans la même classe pour W alors d'après la construction de W', x étant dans W', tous les éléments de sa classe y sont aussi, donc en particulier y. $W', x \models \exists y\ \chi_i(x,y)$ avec le même y.
Si y et x sont dans des classes différentes, il est possible que la classe correspondant à y ne soit plus dans le mot W'. On regarde la lettre qui étiquette y et la position de y par rapport à x. Supposons que y soit étiquetée par a et située à gauche de x. On regarde alors la position y_0 correspondant à première occurrence de a du mot W. Cette position est forcément située à gauche de y et donc de x et apparait dans le mot W' par construction. Si x et y_0 sont dans des classes différentes alors $W', x, y_0 \models \chi_i(x,y)$ (car χ_i est sans quantificateurs).
Si y_0 est dans la classe de x, on regarde dans W la position y_1 correspondant à la position de la première occurrence de a qui n'est pas dans la classe de y_0. Cette position existe (car y est étiquetée a et $y \nsim x$) et elle est forcément à gauche de y ou à la position y. Donc y_1 est aussi à gauche de x, étiquetée a et dans une classe différente donc $W', x, y_1 \models \chi_i(x,y)$.

$$W' \models NF(\varphi)$$

Formule sans donnée A partir d'un modèle W de $NF(\varphi)$, on peut donc toujours construire un autre modèle W' dont le nombre de classe est borné par $k = 4|\Sigma'|$. On peut alors ramener le problème de satisfaisabilité de $NF(\varphi)$ au problème de satisfaisabilité d'une formule FO^2 sans donnée de la manière suivante : on considère un nouvel alphabet $\Sigma'' = \Sigma' \times \{d_1, d_2, \cdots d_k\}$ afin de coder les classes dans les étiquettes. On construit une nouvelle formule φ'' en remplaçant dans la formule $NF(\varphi)$, la sous formule $x \sim y$ par $\bigvee_{i=1}^{k} d_i(x) \leftrightarrow d_i(y)$.

On a que φ'' est satisfaisable ssi $NF(\varphi)$ est satisfaisable et a un modèle ayant au plus $k = 4|\Sigma'|$ classes distinctes. Or ceci est équivalent au fait que $NF(\varphi)$ soit satisfaisable et donc au fait que φ soit satisfaisable.

Sat-FO^2 sans donnée étant décidable, Sat-$FO^2(<,\sim)$ l'est aussi.

6.3 Sat-$FO^2(<,\sim)$ est NEXPTIME

Regardons maintenant la complexité de la construction précédente.

Construction de φ'' :
La construction de $NF(\varphi)$ est polynomiale en la taille de la formule de départ φ. La construction de φ'' est polynomiale en la taille de $NF(\varphi)$ en considérant l'alphabet Σ''

comme $\Sigma'' = 2^{Prop}$ où $Prop$ est un ensemble polynomial de propositions atomiques. Donc, on écrit $\bigwedge_{p \in Prop} p(x) \leftrightarrow p(y)$ à la place de $\bigvee_{i=1}^{k} d_i(x) \leftrightarrow d_i(y)$

Taille de φ''
La taille finale de φ'' est bornée par $\mathcal{O}(|NF(\varphi)|Prop|))$. Or la taille de $NF(\varphi)$ est polynomiale en φ et $|Prop| = \mathcal{O}(\log |\Sigma'|)$ avec $|\Sigma'| = \mathcal{O}(2^{|\varphi|}|.\Sigma|)$. On a donc $|\varphi"| = poly(|\varphi|.|\Sigma|)$.

Sat-FO^2 sans donnée est NEXPTIME
Donc finalement Sat-$FO^2(<, \sim)$ est NEXPTIME.

6.4 Sat-$FO^2(<, \sim)$ est NEXPTIME-complet

La construction précédente nous donne que Sat-$FO^2(<, \sim)$ est NEXPTIME. Pour montrer la complétude, on va réduire un problème de dominos connu pour être lui même NEXPTIME-complet à Sat-$FO^2(<, \sim)$.

Données :

- $\mathcal{C}$ un ensemble fini de couleurs
- $\mathcal{D} \subseteq \mathcal{C}^4$ un ensemble fini de dominos
- $w \in \mathcal{D}^*$ une suite de dominos de longueur n correctement agencés

Question :

Peut-on paver un carré $2^n \times 2^n$ avec des dominos de $\mathcal{D}$ et de telle sorte que le motif w apparaisse dans le coin en haut à gauche ?

Ce problème est NEXPTIME-Complet. [Fü84]

Le codage :

Etant donnée une instance de Dominos :
On prend $\Sigma = \mathcal{D} \cup \#, \$$
On code un pavage du carré $\mathcal{P} : \{1, \cdots, 2^n\}^2 \to \mathcal{D}$ de la manière suivante :
- $w \in (\#.\$^*)^n.P.(\$^*\#)^n$.
- Chaque domino appartient à une classe différente qui contient les informations nécessaires pour coder sa position dans le pavage de la manière suivante :
  Si le domino de la classe d est à la position (i, j) (ligne,colonne) dans le pavage, on considère le codage binaire de i et j.
  - si le bit numéro p du codage de i est 1 alors la donnée d étiquette un \$ situé entre le $p^{\text{ème}}$ et le $p+1^{\text{ème}}$ # à partir du début du mot w.

– si le bit numéro p du codage de i est 0 alors le donnée d n'apparait pas entre le $p^{\text{ème}}$ et le $p+1^{\text{ème}}$ # à partir du début du mot w.
– si le bit numéro p du codage de j est 1 alors la donnée d étiquette un \$ situé entre le $p^{\text{ème}}$ et le $p+1^{\text{ème}}$ # à partir de la fin du mot w.
– si le bit numéro p du codage de j est 0 alors la donnée d n'apparait pas entre le $p^{\text{ème}}$ et le $p+1^{\text{ème}}$ # à partir de la fin du mot w.

On définit les deux ensembles finis suivants :
– DroiteGauche $\subseteq \mathcal{D} \times \mathcal{D}$ Ensemble des couples de dominos tels que l'on peut mettre le second à droite du premier en respectant les couleurs
– BasHaut $\subseteq \mathcal{D} \times \mathcal{D}$ Ensemble des couples de dominos tels que l'on peut mettre le second en dessous du premier en respectant les couleurs

Etant donnée une instance du problème domino, on construit une formule $FO^2(<,\sim)$ qui reconnaît les solutions codées comme expliqué ci-dessus.

Quelques formules intermédiaires :

– A partir de x, le mot compte au moins k occurrences de la lettre α à sa droite.

$$\begin{array}{ll} k=1 & \mathbf{D}_{\alpha}^{\geq \mathbf{1}}(x) := \ \exists y\ [x<y\ \wedge\ \alpha(y)] \\ k=k+1 & \mathbf{D}_{\alpha}^{\geq \mathbf{k+1}}(x) := \ \exists y\ [x<y\ \wedge\ \alpha(y)\ \wedge\ D_{\alpha}^{\geq k}(y)] \end{array}$$

– A partir de x, le mot compte au moins k occurrences de la lettre α à sa gauche.

$$\begin{array}{ll} k=1 & \mathbf{G}_{\alpha}^{\geq \mathbf{1}}(x) := \ \exists y\ [x>y\ \wedge\ \alpha(y)] \\ k=k+1 & \mathbf{G}_{\alpha}^{\geq \mathbf{k+1}}(x) := \ \exists y\ [x>y\ \wedge\ \alpha(y)\ \wedge\ G_{\alpha}^{\geq k}(y)] \end{array}$$

– La projection du mot est dans le langage $(\#.\$^*)^n.\mathcal{D}^*.(\$^*\#)^n$:

$$\begin{array}{rl} \mathbf{Mot} := & \exists x\ [\min(x)]\ \wedge\ D_{\#}^{\geq 2n}(x)\ \wedge\ \neg D_{\#}^{\geq 2n+1}(x) \\ \wedge & \forall x\ \mathcal{D}(x) \rightarrow [D_{\#}^{\geq n}(x)\ \wedge\ G_{\#}^{\geq n}(x)] \\ \wedge & \neg[\exists x\ \mathcal{D}(x)\ \wedge\ [\exists y\ x<y\ \wedge\ \$(y)\ \wedge\ [\exists x\ y<x\ \wedge\ \mathcal{D}(x)]]] \end{array}$$

La première ligne exprime le fait qu'il y a exactement $2n$ occurrences de # dans le mot.
La seconde assure que tout domino est entouré de n # à sa droite et à sa gauche.
La dernière assure que un \$ n'est jamais entouré de dominos.

– Les dominos appartiennent à des classes distinctes :

$$\mathbf{Classes} := \forall x \forall y\ [x \neq y\ \wedge\ \mathcal{D}(x)\ \wedge\ \mathcal{D}(y)] \rightarrow\ x \nsim y$$

– Pour le domino de la position x : Le bit numéro p du codage de i (ligne) est 1 (ie la donnée de la position x étiquette un \$ situe entre le $p^{\text{ème}}$ et le $p+1^{\text{ème}}$ # à partir du début du mot).

$$\mathbf{Bit_i^p}(x) := \ \mathcal{D}(x)\ \wedge\ [\exists y\ y<x\ \wedge\ x\sim y\ \wedge\ \$(y)\ \wedge\ G_{\#}^{p}\ \wedge\ \neg G_{\#}^{p+1}]$$

– Pour le domino de la position x : Le bit numéro p du codage de i (ligne) est 0 (ie la donnée de la position x n'apparait pas entre les $p^{\text{ème}}$ et le $p+1^{\text{ème}}$ # à partir du début du mot).

$$\neg\mathbf{Bit_i^p}(x)$$

– Pour le domino de la position x : Le bit numéro p du codage de j (colonne) est 1 (ie la donnée de la position x étiquette un \$ situe entre le $p^{\text{ème}}$ et le $p+1^{\text{ème}}$ # à partir de la fin du mot).

$$\mathbf{Bit_j^p}(x) := \ \mathcal{D}(x) \ \wedge \ [\exists y \ x < y \ \wedge \ x \sim y \ \wedge \ \$(y) \ \wedge \ D^p_{\#} \ \wedge \ \neg D^{p+1}_{\#}]$$

– Pour le domino de la position x : Le bit numéro p du codage de j (colonne) est 0 (ie la donnée de la position x n'apparait pas entre les $p^{\text{ème}}$ et le $p+1^{\text{ème}}$ # à partir de la fin du mot).

$$\neg\mathbf{Bit_j^p}(x)$$

– Les positions codées par les données des dominos sont toutes différentes :

$$\mathbf{Positions} := \forall x \forall y \ [x \neq y \wedge \mathcal{D}(x) \wedge \mathcal{D}(y)] \rightarrow \neg \bigwedge_{p=0..n-1} [Bit_j^p(x) \leftrightarrow Bit_j^p(y)] \wedge [Bit_i^p(x) \leftrightarrow Bit_i^p(y)]$$

– les positions x et y sont dans la même colonne j du pavage.

$$\mathbf{Egal}_j(x,y) := \bigwedge_{p=0..n-1} [Bit_j^p(x) \leftrightarrow Bit_j^p(y)]$$

– les positions x et y sont sur la même ligne i du pavage.

$$\mathbf{Egal}_i(x,y) := \bigwedge_{p=0..n-1} [Bit_i^p(x) \leftrightarrow Bit_i^p(y)]$$

– la position y est dans la colonne j à droite de la position x dans le pavage.

$$\begin{aligned}\mathbf{Succ_j(x,y)} := & \textstyle\bigwedge_{p=0..n-1} \ [\bigvee_{k<p} \neg Bit_j^k(x)] \rightarrow [Bit_j^p(x) \leftrightarrow Bit_j^p(y)] \\ & \wedge \textstyle\bigwedge_{p=0..n-1} \ [\bigwedge_{k<p} Bit_j^k(x)] \rightarrow [Bit_j^p(x) \leftrightarrow \neg Bit_j^p(y)]\end{aligned}$$

– la position y est dans la ligne i à droite de la position x dans le pavage.

$$\begin{aligned}\mathbf{Succ_i(x,y)} := & \textstyle\bigwedge_{p=0..n-1} \ [\bigvee_{k<p} \neg Bit_i^k(x)] \rightarrow [Bit_i^p(x) \leftrightarrow Bit_i^p(y)] \\ & \wedge \textstyle\bigwedge_{p=0..n-1} \ [\bigwedge_{k<p} Bit_i^k(x)] \rightarrow [Bit_i^p(x) \leftrightarrow \neg Bit_i^p(y)]\end{aligned}$$

– Le pavage est complet :

$$\begin{aligned}\mathbf{Complet} := \ & \exists x \ \textstyle\bigwedge_{p=0..n-1} \neg[Bit_i^p(x) \ \vee \ Bit_j^p(x)] \\ \wedge \ & \exists x \ \textstyle\bigwedge_{p=0..n-1} [Bit_i^p(x) \ \wedge \ Bit_j^p(x)] \\ \wedge \ & \forall x \ \neg[\textstyle\bigwedge_{p=0..n-1} Bit_i^p(x)] \rightarrow [\exists y \ Egal_j(x,y) \ \wedge \ Succ_i(x,y)] \\ \wedge \ & \forall x \ \neg[\textstyle\bigwedge_{p=0..n-1} Bit_j^p(x)] \rightarrow [\exists y \ Egal_i(x,y) \ \wedge \ Succ_j(x,y)]\end{aligned}$$

– Le pavage est correct :

$$\begin{aligned}\textbf{Correct} := \forall x \forall y \quad & [Egal_i(x,y) \ \wedge Succ_j(x,y)] \rightarrow droite(x) = gauche(y) \\ \wedge \quad & [Egal_j(x,y) \ \wedge Succ_i(x,y)] \rightarrow bas(x) = haut(y)\end{aligned}$$

avec

$$doite(x) = gauche(y) := \bigvee_{(d_x,d_y)\in DroiteGauche} [d_x(x) \wedge d_y(y)]$$

$$bas(x) = haut(y) := \bigvee_{(d_x,d_y)\in BasHaut} [d_x(x) \wedge d_y(y)]$$

– le motif $w = d_1.d_2...d_n$ appairaît en haut à gauche :

$$\begin{aligned}\textbf{Init}_\mathbf{w} := \ \ & \exists x \textstyle\bigwedge_{p=0..n-1} \neg[Bit_i^p(x) \ \vee \ Bit_j^p(x)] \ \wedge \ d_1(x) \\ & \wedge \exists y \ Egal_i(x,y) \ \wedge \ Succ_j(x,y) \ \wedge \ d_2(y) \\ & \quad \wedge \exists x \ Egal_i(x,y) \ \wedge \ Succ_j(y,x) \ \wedge \ d_3(x) \\ & \quad \vdots \\ & \quad\quad \wedge \exists y \ Egal_i(x,y) \ \wedge \ Succ_j(x,x) \ \wedge \ d_n(x)\end{aligned}$$

Finalement :

Un mot W est le codage d'une solution du problème domino ssi W est un modèle pour la formule **Mot** $\wedge$ **Classes** $\wedge$ **Positions** $\wedge$ **Complet** $\wedge$ **Correct** $\wedge$ $\textbf{Init}_w$.

On remarque que cette formule est de taille polynomiale en la taille du problème.

Conclusions

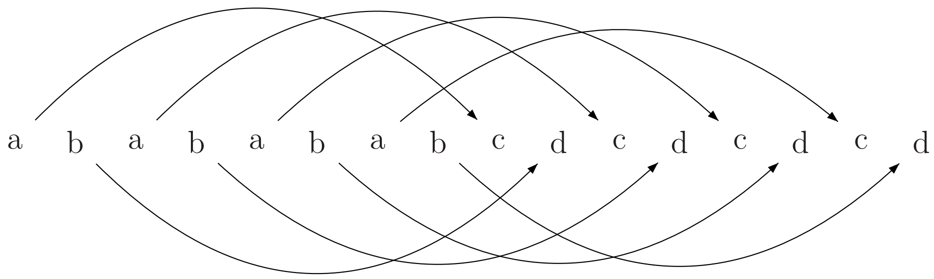
On a montré que Sat-$FO^2(<,\sim)$ est dans NEXPTIME. Etant donné une instance du problème domino, on a construit une formule de taille polynomiale qui est satisfaisable ssi le problème domino a une solution. Le problème domino étant NEXPTIME-complet, Sat-$FO^2(<,\sim)$ est NEXPTIME-complet.

6.5 Décidabilité de Sat-$EMSO^2(<,\sim)$

La décidabilité de Sat-$EMSO^2(<,\sim)$ découle directement de celle de Sat-$FO^2(<,\sim)$. On ramène la satisfaisabilité d'une formule $EMSO^2(<,\sim)$ de la forme $\exists X_0...\exists X_k \ \varphi(X_0,...,X_k)$ à la satisfaisabilité d'une formule $FO^2(<,\sim)$ en codant les variables du second ordre dans l'alphabet. On considère l'alphabet étendu $\Sigma^k = \Sigma \times \{0,1\}^k$ et on remplace dans la formule $\varphi(X_0,...,X_k)$, la sous formule "$x \in X_i$" par "$\pi_i(x) = 1$" qui traduit le fait que la position x est étiquetée par une lettre de $\Sigma \times \{0,1\}^{i-1} \times 1 \times \{0,1\}^{k-i}$.

6.6 Cas de $FO^2(<, \sim, +1)$

Pour le successeur, nous n'avons pas encore trouvé de solution du même genre. Il semble que l'idée de partir d'un modèle déjà construit et de supprimer des classes ne fonctionne pas. En effet si on regarde l'exemple suivant, il semble que l'on ne peut supprimer aucune classe. (Les flêches relient les éléments d'une même classe)



7 Le vide pour les automates du type po-2-DFA$_1$ est indécidable

On montre ici que le vide est indécidable pour les automates du type po-2-DFA$_1$ (déterministes, bi-directionnels, à un registre, partiellement ordonnés entre deux mises à jour). Pour cela, à partir d'un automate déterministe à deux compteurs $\mathcal{A}_\mathcal{C}$, on construit un automate à registre $\mathcal{A}_\mathcal{D}$ de type po-2-DFA$_1$ qui reconnaît les calculs acceptants de l'automate à deux compteurs $\mathcal{A}_\mathcal{C}$.

7.1 Automates déterministe à deux compteurs

$\mathcal{A}_\mathcal{C} = (Q, q_{initial}, Q_{final}, T)$
On a deux types de transitions avec $i \in \{1, 2\}$ (où p_i note le contenu du compteur i) :

$$q \longrightarrow q',\ p_i := p_i + 1$$

$$q \longrightarrow \begin{array}{l} \text{si } p_i > 0 \text{ alors } q',\ p_i := p_i - 1 \\ \text{si } p_i = 0 \text{ alors } q'' \end{array}$$

Les automates considérés étant déterministes, il existe pour chaque état une seule transition du type précédent possible (modulo le test à 0 des compteurs).

7.2 Codage d'un calcul

Codage d'une configuration
Une configuration de l'automate $\mathcal{A}_\mathcal{C}$ est définie par un état et les valeurs des deux registres.

L'idée est de coder chaque compteur en unaire sur une lettre qui lui est propre (a pour le premier compteur, b pour le second).
On prend $\Sigma = Q \cup \{a, b, \#\}$ et on code la configuration de l'automate $c = [q, p_1, p_2]$ par le mot suivant w_c :

- $\Pi_\Sigma(w_c) = \#.q.a^{p_1+1}.b^{p_2+1}$
- où les a sont dans des classes différentes.
- où les b sont dans des classes différentes.

On identifie dans la suite une configuration de $\mathcal{A}_\mathcal{C}$ et son codage.

Codage d'un calcul de l'automate $\mathcal{A}_\mathcal{C}$
Soit la suite de configuration de $\mathcal{A}_\mathcal{C}$ "$c_1.c_2 \cdots c_n$" correspondant à un calcul de l'automate $\mathcal{A}_\mathcal{C}$.
On code ce calcul avec un mot $W_r \in (\Sigma \times D)^*$ définit par : $W_r = w_{c_1}.w_{c_2}...w_{c_n}$ avec la restriction suivante sur les données de deux configurations successives :

- si le premier compteur garde la même valeur dans les deux configurations, alors à toute position étiquetée a de la première configuration correspond un a de la même classe dans la seconde et réciproquement.
- si le premier compteur est incrémenté entre la première et la seconde configuration, alors à toute position étiquetée a de la première configuration correspond un a de la même classe dans la seconde.
- si le premier compteur est décrémenté entre la première et la seconde configuration, alors à toute position étiquetée a de la seconde configuration correspond un a de la même classe dans la première.
- on a les mêmes restrictions pour le second compteur dont les positions correspondantes dans les configurations sont étiquetées par des b.

7.3 L'automate po-2-DFA$_1$ correspondant : $\mathcal{A}_\mathcal{D}$

Afin de vérifier qu'un mot w de $(\Sigma \times D)^*$ est le codage d'un calcul acceptant de $\mathcal{A}_\mathcal{C}$, l'automate $\mathcal{A}_\mathcal{D}$ procède en deux étapes :

La première étape consiste à vérifier que la projection du mot w sur l'alphabet fini est bien dans le langage $(\#.Q.a^+.b^+)^*$.(On peut faire en prenant un automate classique qui reconnaît ce langage auquel on rajoute des mises à jour afin de casser les cycles.)

L'automate va ensuite vérifier la cohérence du calcul en considérant un à un les blocs w_{c_i} étiquetés par $\#.q_i.a^{i_1}.b^{i_2}$ du mot w. (c-à-d les blocs correspondants au codage d'une configuration de $\mathcal{A}_\mathcal{C}$)

Sur un bloc le calcul de l'automate $\mathcal{A}_\mathcal{D}$ se décompose en deux étapes :

- Le choix d'une transition de $\mathcal{A}_\mathcal{C}$

– Si la transition de l'automate $\mathcal{A}_\mathcal{C}$ associée à l'état q_i est la suivante :

$$q_i \longrightarrow q',\ p_1 := p_1 + 1$$

L'automate $\mathcal{A}_\mathcal{D}$ va aller vérifier dans le mot w que la lettre q_{i+1} qui correspond à l'état du bloc $w_{c_{i+1}}$ est bien q'

– Si la transition de l'automate $\mathcal{A}_\mathcal{C}$ associée à l'état q_i est la suivante :

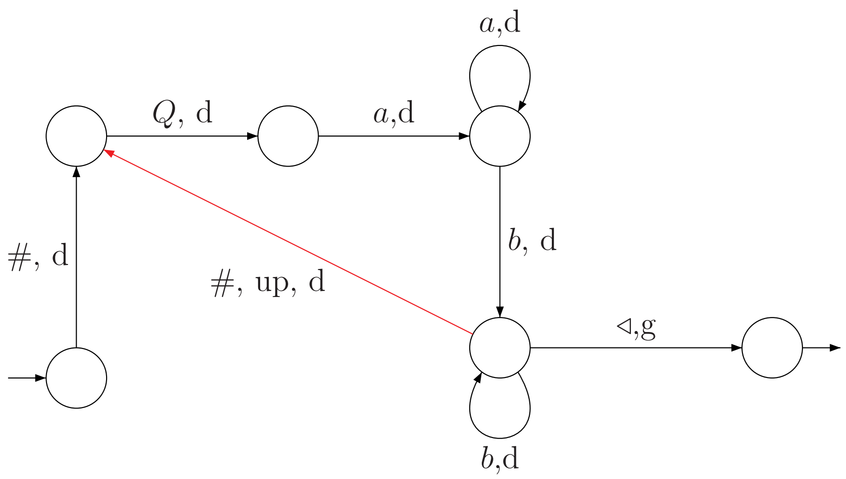
$$q \quad \longrightarrow \quad \begin{array}{l} \text{si } p_1 > 0 \text{ alors } q',\ p_1 := p_1 - 1 \\ \text{si } p_1 = 0 \text{ alors } q'' \end{array}$$

L'automate $\mathcal{A}_\mathcal{D}$ va aller regarder dans le mot w si le bloc w_{c_i} contient une ou plusieurs positions étiquetées a (ce qui correspond au fait que le premier compteur est nul ou non) puis en fonction du résultat va vérifier que la lettre q_{i+1} qui correspond à l'état du bloc $w_{c_{i+1}}$ est bien q' ou q''

– La vérification des compteurs :
Il reste à vérifier que les parties correspondant aux compteurs dans le bloc w_{c_i} évoluent correctement dans $w_{c_{i+1}}$ en fonction de la "transition choisie" dans l'étape précédente. On explique la façon de procéder dans la suite avec un exemple.

L'automate qui reconnaît la forme générale du mot :

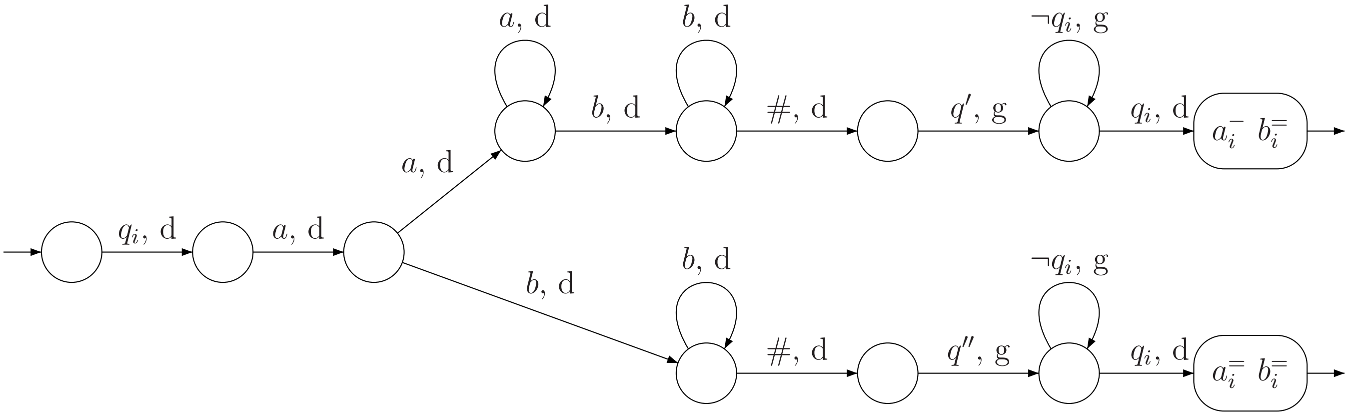
On peut facilement vérifier qu'un mot est étiqueté dans $(\#.Q.a^+.b^+)$.



Le choix d'une transition de $\mathcal{A}_{\mathcal{C}}$

Automate correspondant à la transition :

$$q \quad \longrightarrow \quad \begin{array}{l} \text{si } p_1 > 0 \text{ alors } q',\ p_1 := p_1 - 1 \\ \text{si } p_1 = 0 \text{ alors } q'' \end{array}$$



On note $a_i^=\ b_i^=$ pour signifier que le premier et le second compteur doivent être laissés inchangés entre le bloc configuration courant et le suivant. De même, on note $a_i^-\ b_i^=$ pour signifier que le premier compteur doit être décrémenté et le second laissé inchangé entre le bloc configuration courant et le suivant.

Cet automate est simplifié car pour la suite de la vérification, on veut garder en mémoire dans l'automate $\mathcal{A}_{\mathcal{D}}$ la transition précédente et la transition courante. L'état initial de l'automate dessiné ci-dessus est en réalité étiqueté par $a_{i-1}\ b_{i-1}$ et l'état final étiqueté $(a_{i-1}\ b_{i-1})(a_i\ b_i)$

Vérifier les transitions du point de vue des compteurs

L'automate $\mathcal{A}_{\mathcal{D}}$ est maintenant dans l'état $(a_{i-1}\ b_{i-1})(a_i\ b_i)$ avec sa tête de lecture au début du bloc w_{c_i}. On ne va pas vérifier complètement les contraintes mais seulement une partie des contraintes entre w_{c_i} et $w_{c_{i-1}}$ et une partie des contraintes entres w_{c_i} et $w_{c_{i+1}}$.

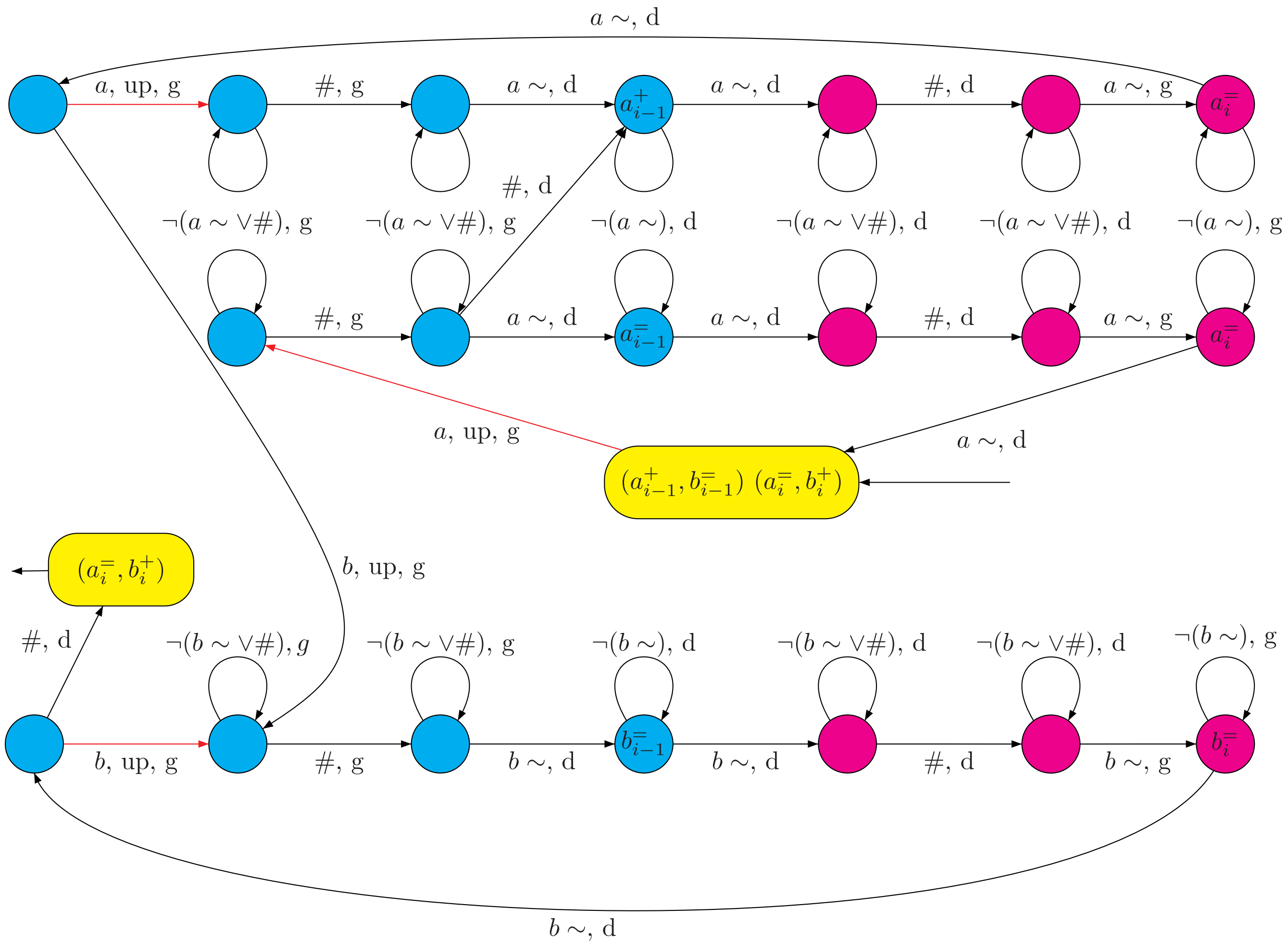
Plus précisément, dans l'état, $(a_{i-1}\ b_{i-1})(a_i\ b_i)$ on va vérifier :
- si on a a_{i-1}^- ou $a_{i-1}^=$, on vérifie que pour toutes les positions de w_{c_i} étiquetée par un a, il existe dans $w_{c_{i-1}}$ une position dans la même classe étiquetée elle même par un a
- si on a a_{i-1}^+, on vérifie que pour toute les positions de w_{c_i} étiquetée par un a sauf une, il existe dans $w_{c_{i-1}}$ une position dans la même classe étiquetée elle même par un a
- si on a a_i^+ ou $a_i^=$, on vérifie que pour toute les positions de w_{c_i} étiquetée par un a, il existe dans $w_{c_{i+1}}$ une position dans la même classe étiquetée elle même par un a
- si on a a_i^-, on vérifie que pour toute les positions de w_{c_i} étiquetée par un a sauf une, il existe dans $w_{c_{i+1}}$ une position dans la même classe étiquetée elle même par un a

– on fait la même chose pour les b.

En même temps que l'on fait ces vérifications, on en profite pour vérifier que, pour le bloc-configuration courant, les a [resp b] sont dans des classes différentes.

La façon dont l'automate traite les configurations, fait que les contraintes seront vérifiées en cascade au fur et à mesure des configurations. Lors de l'étape correspondant à la configuration w_{c_i}, on finit de vérifier que les compteurs sont correctement modifiés entre les configurations $w_{c_{i-1}}$ et w_{c_i} et on commence à vérifier les compteurs pour le passage de w_{c_i} à $w_{c_{i+1}}$.

Exemple d'une étape : $(a_{i-1}^{+}\ b_{i-1}^{=})(a_i^{=}\ b_i^{+})$



Conclusion
L'automate $\mathcal{A}_\mathcal{D}$ ainsi décrit reconnaît le codage de l'ensemble des calculs acceptants de l'automate $\mathcal{A}_\mathcal{C}$ de départ déterministe à deux compteurs. Le vide des automates déterministe à deux compteurs est indécidable [Min67] donc le vide de nos automates po-2-DFA$_1$ est lui aussi indécidable.

8 Résumé des résultats

8.1 Logique

Expressivité
Les logiques étudiées ont une expressivité intéressante. Sans donnée la logique du premier ordre ne définit qu'un sous-ensemble des langages réguliers (les langages sans étoile). Ici, avec uniquement deux variables, on arrive à définir, sans successeur, des langages dont la projection n'est pas régulière voire même non hors-contexe. En s'autorisant le successeur, on arrive à définir des langages dont la projection sur l'alphabet fini est n'importe quel régulier.

Les résultats principaux
Les résultats importants concernant les logiques étudiées sont :
- La frontière de décidabilité entre $FO^3(<,\sim)$ et $FO^2(<,\sim)$
- Le résultat de complexité : Sat-$FO^2(<,\sim)$ est NEXPTIME-complet.

Le successeur
L'ajout du successeur augmente l'expressivité de la logique. Avec le successeur, on peut définir, pour n'importe quel langage régulier, une version augmentée avec des données de ce langage.
Nous n'avons pas de résultat de décidabilité ni d'indécidabilité pour le cas avec successeur $FO^2(<,\sim,+1)$. La méthode utilisée pour la décidabilité de $FO^2(<,\sim)$ ne semble pas pouvoir s'étendre (cf exemples des classes imbriquées). Si on veut réussir à montrer la décidabilité, il faudra peut-être plutôt s'orienter vers une méthode qui analyse la formule et construit directement le modèle. Mais, le successeur suffit peut-être à rendre la logique indécidable. Aucune des deux pistes n'est privilégiée pour l'instant.

8.2 Automates

Résultats
En ce qui concerne les automates, les résultats ne sont pas vraiment satisfaisants. Le résultat principal à noter est que malgré la restriction "partiellement ordonnés entre deux mises à jour", le vide des automates po-2-DFA$_1$ est indécidable.

Comparaison avec la logique

Le problème se pose déjà dans le cas sans donnée. Il a été montré que $FO^2(<) =$ po-2-DFA (par des arguments algébriques) et pourtant on n'arrive pas, à partir de la formule, à construire l'automate correspondant.
On pouvait espérer trouver dans une telle construction des idées pour : définir des automates plus proches des logiques pour le cas avec données ou encore montrer une inclusion logique automate.

9 Perspectives

Du point de vue de la logique, notre but est d'avoir une logique décidable la plus expressive possible.

- On va donc chercher à voir ce que l'on peut rajouter à la logique du $FO^2(<, \sim)$ tout en restant décidable. On va regarder plus en détail $FO^2(<, \sim, +1)$; on peut aussi envisager de rajouter un prédicat de successeur de classe (qui permettrait de parler du prochain élément de la classe correspondant à une position).
- Une autre direction est de changer de type de logique. Le cas sans données nous suggère de regarder ce que l'on pourrait faire avec une logique temporelle traitant les données.

Du point de vue des automates, on aimerait déjà réussir à avoir une classe d'automates dont le vide est décidable. Pour cela, il faut trouver des restrictions supplémentaires à imposer à nos automates po-2-DFA_1 pour les rendre décidables. Voici quelques idées de restrictions :

- Le fait que les automates soient bi-directionnels est important, on a donc pensé à restreindre les mises à jour en demandant qu'elles soient monotones pour un calcul acceptant. Cette restriction n'a pas été retenue car le codage des automates à deux compteurs peut-être reconnu avec un automate respectant cette hypothèse à condition de modifier légèrement l'automate po-2-DFA_1 utilisé.
- Une autre idée est d'imposer un ordre partiel sur les transitions de mise à jour. La plupart des exemples de langages reconnaisables par les po-2-DFA_1 semblent toujours pouvoir être reconnus avec cette hypothèse supplémentaire. La preuve d'indécidabilité avec deux registres fonctionne toujours. Par contre, l'exemple des langages réguliers reconnus par blocs ne semble plus être reconnaissable. Cette hypothèse impose en quelque sorte d'avoir à faire un type de mise à jour donné suivi d'une vérification ne nécessitant pas de mise à jour (et ceci autant de fois que nécessaire) puis de passer à une autre étape de vérification. Cette hypothèse parait intéressante car il semble que le codage des calculs d'automates à deux compteurs ne soit plus reconnaissable avec cette hypothèse supplémentaire.

  Cette hypothèse permettra peut-être d'inclure les automates dans la logique du premier ordre.

– Une autre solution envisagée s'inspire des programmes Turtle pour le cas sans données. On pourrait envisager de dissocier deux types de transitions, des transitions qui cherchent le premier élément dans la classe du registre ou dans une classe différente mais sans voir les lettres qui étiquettent les positions parcourues (des transitions aveugles sur les lettres) et des transitions classiques qui regardent les étiquettes de positions mais pas leurs classes (aveugles sur les données). L'idée est que lorsque l'on cherche le premier élément de la classe à droite, l'automate ne sait pas par exemple s'il a dépassé une position étiquetée par le séparateur # ou non. On peut donc penser qu'un tel automate ne pourra plus reconnaître des calculs codés par des suites de configurations de taille non bornée.
  On remarque que la logique du premier ordre $FO^2(<, \sim)$ ne semble pas pouvoir détecter qu'entre deux positions apparaît une lettre donnée. Cette hypothèse empêche aussi les automates de le faire. C'est peut-être cette différence qui rend la logique décidable.

A terme, on aimerait avoir des automates et des logiques qui soient équivalents ou au moins comparables. Et pour cela, il serait intéressant de trouver une méthode dans le cas sans donnée pour passer d'une formule $FO^2(<)$ à un automate po-2-DFA, afin de comprendre les liens qui existent entre les automates et la logique.

Références

[Fü84] M. Fürer. The computational complexity of the unconstrained domino problem (with implications for logical decision problems). *In Logic and Machines : Decision Problems and Complexity*, pages 312-319. Volume 171 in Lect. Notes in Comput. Sci., Springer, 1984

[EVW97] K. Etessami, M. Y. Vardi, Th. Wilke. First-order logic with two variables and unary temporal logic In Proc. of 12th IEEE Symp. on Logic in Computer Science (LICS'97) , pp. 228-235, 1997. Inf. Comp. 179(2) 2002

[TW98] D. Thérien, W. Wilke, Over words, two variables are as powerful as one quantifier alternation Proceedings of the 30th ACM STOC Conference, 1998, 234-240.

[BPT01] P. Bouyer, A. Petit, D. Thérien. An algebraic characterization of data and timed languages. In Proc. 12th Int. Conf. Concurrency Theory (CONCUR 2001), Aalborg, Denmark, Aug. 2001, volume 2154 of Lecture Notes in Computer Science, pages 248-261. Springer, 2001.

[STV01] Th. Schwentick, D. Thérien, H. Vollmer. Partially-ordered Two-way Automata : A New Characterization of DA. In Proc. 5th Developments in Language Theory (DLT 2001), 239-250, 2001.

[NSV01] Fr. Neven, Th. Schwentick, V. Vianu. Towards regular languages over infinite alphabets.
In Proc. 26th Symposium on Mathematical Foundations of Computer Science (MFCS 2001), pages 560-572, 2001. Abstract

[Min67] M. Minsky. Computation : Finite and Infinite Machines.
Prentice Hall International, 1967