Le ι-calcul* un langage de contraintes d'ordre supérieur

Fred Mesnard Antoine Rauzy

Iremia, Université de La Réunion
15, avenue René Cassin - BP 7151 -
97 715 Saint Denis Messag. Cedex 9
E-mail: fred@univ-reunion.fr URL : www.univ-reunion.fr/~gcc

LaBRI, URA CNRS 1304
351, cours de la Libération
33 405 Talence Cedex
E-mail: rauzy@labri.u-bordeaux.fr

Résumé

L'analyse de programmes en général, et l'analyse de programmes déclaratifs en particulier, demande de calculer des points fixes. Dans le travail présenté ici, nous cherchons à définir un cadre calculatoire générique, appelé ι-calcul, pour les différents calculs de points fixes utilisés dans le cadre de la vérification de programmes. Nous définissons tout d'abord la syntaxe abstraite du ι-calcul et la sémantique opérationnelle associée. Nous instancions en suite notre calcul pour retrouver un interprète de la logique CTL, du μ-calcul propositionnel, un calculateur de points fixes sur les polyhèdres rationnels et enfin un calculateur de points fixes sur les intervalles d'entiers.

1 Introduction

L'analyse de programmes en général, et l'analyse de programmes déclaratifs en particulier, demande de calculer des points fixes. Les vérificateurs de modèles [15], les outils d'analyse de programmes Prolog [16] ou de programmes réactifs [11] mettent donc en œuvre des algorithmes de calcul de points fixes de systèmes d'équations. Ces systèmes d'équations définissent des relations portant sur des domaines différents (booléens, nombres rationnels, termes), codées de manières différentes (diagrammes binaires de décision, polyhèdres, termes prolog, ...).

Le μ-calcul [17, 1] offre un cadre théorique pour définir ce type de calculs. Il introduit, en plus des opérateurs usuels de la logique du premier ordre, l'opérateur de plus petit point fixe μ et l'opérateur de plus grand point fixe ν. Ces deux opérateurs permettent une forme de quantification sur les relations. Le μ-calcul est de fait une restriction de la logique monadique du second ordre. C'est donc principalement un langage ensembliste. Or, tous les domaines évoqués plus haut ne se prêtent pas forcément très bien à une présentation ensembliste. En effet, les opérations ensemblistes peuvent ne pas être définissables sur le domaine considéré, ou être trop coûteuses à mettre en œuvre (comme, par exemple, l'union de deux polyhèdres). Réciproquement, les opérations sur le domaine considérés peuvent ne pas être des opérateurs ensemblistes (c'est le cas des différents opérateurs d'élargissement et de rétrécissement de [9]).

Dans cet article, nous introduisons un nouveau langage logique, le ι-calcul qui :

- généralise le μ-calcul et offre un cadre calculatoire unifié pour tous les calculs de points fixes ;
- est paramétrable par le domaine d'interprétation considéré ;

*Prononcez "iota-calcul" !

- permet d'introduire,via la notion de squelette de calcul, des opérateurs d'élargissements et de rétrécississement qui accèlerent les calculs (voire les font converger) ;
- rend explicite l'opération de renommage d'une relation mise en œuvre lors des appels récursifs. Cette opération change les variables du domaine (variables d'ordre 0) sur lesquelles portent la relation. Or elle peut être coûteuse (par exemple si on utilise un codage symbolique des relations), c'est pourquoi il nous semble important de la rendre explicite.

Le ι-calcul est un langage de contraintes. D'une part, les constantes du ι-calcul sont des relations prédéfinies ou, autrement dit, des contraintes. D'autre part, il est définit dans l'esprit des langages de programmation logique avec contraintes [7, 12, 6, 13] : il fournit les primitives permettant de poser les équations au point fixe définissant les relations, et se charge de résoudre ces équations.

Notre travail doit donc être vu principalement comme une reflexion méthodologique : nous avons cherché à définir un cadre calculatoire générique pour les différents calculs de points fixes utilisés dans le cadre de la vérification de programmes.

Voici le plan de notre proposition. Nous définissons la syntaxe abstraite du ι-calcul et la sémantique opérationnelle associée. Nous instancions en suite notre calcul pour retrouver un interprète de la logique CTL, du μ-calcul propositionnel, un calculateur de points fixes sur les polyhèdres rationnels et sur les intervalles d'entiers. Ces quatre outils ont été prototypés en Prolog et on trouvera en annexe une brève description, néanmoins exécutable, du cœur générique de ces interprètes.

2 Syntaxe abstraite et sémantique opérationnelle

2.1 Syntaxe abstraite

On se donne le vocabulaire suivant :
- Un ensemble $\mathcal{C} = \{c_1, c_2, \dots\}$ de symboles de contraintes.
- Un ensemble $\mathcal{R} = \{R_1, R_2, \dots\}$ de variables de relations.
- Un ensemble $\mathcal{O} = \{o_1, o_2, \dots\}$ de symboles d'opérations.
- Un ensemble $\mathcal{T} = \{t_1, t_2, \dots\}$ de symboles de tests.

On suppose $\mathcal{C}$, $\mathcal{R}$, $\mathcal{O}$ et $\mathcal{T}$[1] disjoints deux à deux. À chaque symbole de contrainte, chaque variable de relation et chaque symbole d'opération on associe une arité n (un entier positif ou nul). On note $\alpha(s)$ l'arité du symbole s. Les tests sont tous supposés binaires (d'arité 2). Les contraintes sont des relations prédéfinies. Les opérations et les tests portent sur les relations.

On suppose que $\mathcal{C}$ contient au moins une contrainte, que $\mathcal{O}$ contient au moins un symbole d'opération 0-aire et au moins un symbole d'opération binaire et que $\mathcal{T}$ contient au moins un symbole de test.

L'ensemble des *formules* du ι-calcul est le plus petit ensemble tel que :
- Les contraintes sont des formules.
- Les relations sont des formules.
- Si o est un symbole d'opération d'arité n et f_1,...,f_n sont des formules, alors $o(f_1, \dots, f_n)$ est une formule.

Un *système d'équations* du ι-calcul est un ensemble fini d'équations de la forme $R = f$ où R est une variable de relation et f est une formule.

L'ensemble des *tests* du ι-calcul est le plus petit ensemble tel que :
- Si t est un symbole de test et si f_1 et f_2 sont des formules, alors $t(f_1, f_2)$ est un test.

L'ensemble des *squelettes de calcul* du ι-calcul est le plus petit ensemble tel que :
- Si o est un symbole d'opération binaire alors $o*$ est un squelette de calcul.
- Si o est un symbole d'opération binaire et si σ est un squelette de calcul alors $o.\sigma$ est un squelette de calcul.

L'ensemble des *programmes* du ι-calcul est le plus petit ensemble tel que :
- ϵ est un programme (le programme vide).
- Si P_1 et P_2 sont des programmes alors $P_1; P_2$ est un programme.

[1] Involontaire : les auteurs ne s'en sont aperçus qu'à la relecture ...

– Si t est un test, o est un symbole d'opération 0-aire, σ est un squelette de calcul, P est un programme et finalement E est un système d'équations, alors $\iota\ [t\ o.\sigma]\ P\ \{E\}$ est un programme (dont P est un sous-programme).

On note que l'opération 0-aire o a été ajoutée en tête de la séquence σ (pour des raisons qui apparaitront plus tard).

Dans la suite, nous ne considérerons plus que des programmes P vérifiant les propriétés suivantes :

– Toute variable de relation R apparaissant dans P apparaît une et une seule fois en tant que membre gauche d'une équation de la forme $R = f$. On dit que R est uniquement définie dans P.
– Si P est de la forme $P_1; P_2$ alors aucune des variables de relations définies dans P_2 n'apparaît dans P_1.

2.2 Indiçages

Dans la syntaxe définie plus haut, il n'apparaît aucun symbole d'ordre 0 (variable ou constante). La notion d'indiçage permet de faire référence aux objets du domaine d'interprétation (cf. section 2.3) et au codage effectif des relations.

Une suite d'indices de longueur n est une suite d'entiers positifs $0 < i_1 < \ldots < i_n$, notée $\langle i_1, \ldots, i_n\rangle$. On note $\mathcal{I}_n$ l'ensemble des suites d'indices de longueur n et $\mathcal{I} = \bigcup_{n=1}^{\infty} \mathcal{I}_n$.

On note $\|I\|$ la longueur de la suite d'indices I.

Une fonction d'indices d'arité n est une fonction de $\mathcal{I}^n$ dans $\mathcal{I}$.

Un *indiçage* I du ι-calcul une fonction qui associe :

– À chaque symbole de contrainte c d'arité n, une suite d'indices $I(c)$ de longueur n.
– À chaque variable de relation R d'arité n, une suite d'indices $I(R)$ de longueur n.
– À chaque opérateur o d'arité n, une fonction d'indices $I(o)$ d'arité n.

Il suit qu'un indiçage I associe à chaque formule f une suite d'indices $I(f)$. Si f est de la forme $o(f_1, \ldots, f_n)$, $I(f) = I(o)(I(f_1), \ldots, I(f_n))$.

Un indiçage I est *correct* pour un programme P, si pour chaque variable de relation R définie par une équation $R = f$ du système d'équations E d'un sous-programme de P de la forme $\iota\ [t\ \sigma]\ Q\ \{E\}$, alors pour chaque operateur binaire o apparaissant dans σ, on a $I(R) = I(o(R, f))$.

2.3 Sémantique opérationnelle

La sémantique d'un programme P du ι-calcul est paramétrée par le domaine D d'interprétation et par un indiçage I. On note $\mathcal{D} = \bigcup_{n=1}^{\infty} D^n$ et $\|r\|$ l'arité de la relation r (autrement dit, $r \subseteq D^{\|r\|}$).

Une *interprétation* $v_{D,I}$ d'un programme P est donc la donnée :

– d'un domaine d'interprétation D ;
– d'un indiçage I correct pour P ;
– d'une relation $v_{D,I}(c) \subseteq D^{\|I(c)\|}$ pour chaque symbole de contrainte c ;
– d'une fonction $v_{D,I}(o)$ de $(\mathcal{I} \times \mathcal{D})^k \times \mathcal{I}$ dans $\mathcal{D}$ pour chaque symbole d'opération k-aire. $v_{D,I}(o)$ n'a en fait besoin d'être définie que pour les $k+1$-uplets $(\langle \vec{i_1}, r_1\rangle, \ldots, \langle \vec{i_k}, r_k\rangle, \vec{i})$ tels que $\|\vec{i_s}\| = \|r_s\|$ pour $s = 1, \ldots, k$ et $I(o)(\vec{i_1}, \ldots, \vec{i_k}) = \vec{i}$. Elle doit de plus vérifier $\|\vec{i}\| = \|v_{D,I}(o)(\vec{x})\|$ pour tout $k+1$-uplet $\vec{x} = (\langle \vec{i_1}, r_1\rangle, \ldots, \langle \vec{i_k}, r_k\rangle, \vec{i})$. Finalement,
– d'une fonction $v_{D,I}(t)$ de $(\mathcal{I} \times \mathcal{D})^2$ dans $\{0, 1\}$ pour chaque symbole de test t. Comme précédemment, $v_{D,I}(t)$ n'a en fait besoin d'être définie que pour les couples $(\langle \vec{i_1}, r_1\rangle, \langle \vec{i_2}, r_2\rangle)$ tels que $\|\vec{i_s}\| = \|r_s\|$ pour $s = 1, 2$.

L'objectif d'un programme P est de calculer par itérations successives une affectation ρ des variables de relations apparaissant dans P. La valeur d'une variable de relation R dans l'affectation ρ est un sous-ensemble de $D^{\|I(R)\|}$.

On note $[\,]$ l'affectation partielle vide. On note $\rho[r_1/R_1, \ldots, r_s/R_s]$ l'affectation ρ dans laquelle les relations r_1, ..., r_s sont affectées respectivement aux variables R_1, ..., R_s.

D'une manière générale, la valeur $\rho(f)$ d'une formule f est un sous-ensemble de $D^{||I(f)||}$. Elle est calculé inductivement à partir de la valeur des contraintes et de l'affectation (courante) ρ des variables de relations de la façon suivante :

- Si c est un symbole de contrainte, alors
$$\rho(c) \stackrel{\text{def}}{=} v_{D,I}(c)$$
- Si $f = o(f_1, \ldots, f_k)$ où o est un symbole d'opération k-aire et f_1,...,f_k sont des formules, alors
$$\rho(o(f_1, \ldots, f_k)) \stackrel{\text{def}}{=} v_{D,I}(o)\,(\langle I(f_1), \rho(f_1)\rangle, \ldots, \langle I(f_k), \rho(f_k)\rangle, I(o(f_1, \ldots, f_k))$$

On peut alors définir l'opérateur $T_{D,I}$ permettant de calculer l'affectation décrite par le programme P. $T_{D,I}$ prend un programme et une affectation en paramètres et retourne une affectation. Pour obtenir l'affectation recherchée, on calcule $T_{D,I}[P, []]$. Soit ρ une affectation, $T_{D,I}[P, \rho]$ est définit comme suit :

- Si $P = \epsilon$, alors
$$T_{D,I}[P, \rho] \stackrel{\text{def}}{=} \rho$$
- Si P est de la forme $P_1; P_2$, alors
$$T_{D,I}[P, \rho] \stackrel{\text{def}}{=} T_{D,I}[P_2, T_{D,I}[P_1, \rho]]$$
- Si P est de la forme $\iota\ [t\ \sigma]\ Q\ \{E\}$, où $E = R_1 = f_1, \ldots, R_s = f_s$, alors
  - Si σ est de la forme $o.\sigma'$ avec o un opérateur 0-aire, alors
  $$T_{D,I}[P, \rho] \stackrel{\text{def}}{=} T_{D,I}[\iota\ [t\ \sigma']\ Q\ \{E\}, \rho[v_{D,I}(o)(I(R_1))/R_1, \ldots, v_{D,I}(o)(I(R_s))/R_s]]$$
  - Sinon σ est de la forme $o.\sigma'$ ou de la forme $o*$ avec o une opération binaire (dans le deuxième cas, on pose $\sigma' = o*$). Soit $\rho' = T_{D,I}[Q, \rho]$.
    - Si, pour tout $i \in [1, s]$, on a $v_{D,I}(t)(\langle I(R_i), \rho'(R_i)\rangle, \langle I(R_i), \rho'(o(R_i, f_i))\rangle) = 1$, alors
    $$T_{D,I}[P, \rho] \stackrel{\text{def}}{=} \rho'$$
    - Sinon, soit $\rho'' = \rho'[\rho'(o(R_1, f_1))/R1, \ldots, \rho'(o(R_s, f_s))/R_s]$, alors
    $$T_{D,I}[P, \rho] \stackrel{\text{def}}{=} T_{D,I}[\iota\ [t\ \sigma']\ Q\ \{E\}, \rho'']$$

On trouvera dans [8, 9] des conditions suffisantes assurant la convergence finie d'un ι-calcul.

3 Mise en œuvre de la logique CTL

3.1 La logique CTL

La logique CTL [5] a été introduite pour exprimer des propriétés comportementales de systèmes de processus communicants. Le comportement du système considéré est modélisé par un automate d'états fini. Les propriétés comportementales du système peuvent donc être exprimées par des propriétés des états du graphe.

Les formules CTL sont donc interprétés sur un graphe $G = (V, E)$. Elles caractérisent des ensembles d'états vérifiant une certaine propriété. L'ensemble des formules CTL est le plus petit ensemble tel que :

- `true` et `false` sont des formules qui représentent respectivement V et $\emptyset$.
- Les prédicats unaires sont des formules. On définit un prédicat unaire `v` pour chaque sommet v du graphe. `v` représente le singleton $\{v\} \in V$.
- Si f et g sont des formules alors $f \vee g$ et $\neg f$ sont des formules représentant respectivement l'union des ensembles de sommets représentés par f et g et le complémentaire de l'ensemble de sommets représentés par f.
- Si f est une formule alors `EX`f est une formule. `EX`f représente l'ensemble des états ayant au moins un successeur vérifiant f.
- Si f est une formule alors `EF`f et `EG`f sont des formules.
  - `EF`f représente les états dont il part un chemin passant par un état vérifiant f.
  - `EG`f représente les états s tels que tous les chemins partant de s passent par un état vérifiant f.
- Si f et g sont des formules alors `E`$(f$`U`$g)$ et `A`$(f$`U`$g)$ sont des formules.

– $\mathtt{E}(f\mathtt{U}g)$ représente les états dont il part un chemin ne passant que par des états vérifiant f jusqu'à rencontrer un état vérifiant g.
– $\mathtt{A}(f\mathtt{U}g)$ représente les états tels que tous les chemins qui en sont issus ne passent que par des états vérifiant f jusqu'à rencontrer un état vérifiant g.

On définit d'autres opérateurs à partir ceux définis ci-dessus :

$$\begin{aligned} f \wedge g &\equiv \neg(\neg f \vee \neg g) \\ \mathtt{AX}f &\equiv \neg\mathtt{EX}\neg f \\ \mathtt{AF}f &\equiv \neg\mathtt{EG}\neg f \\ \mathtt{AG}f &\equiv \neg\mathtt{EF}\neg f \end{aligned}$$

3.2 Codage des opérateurs CTL en ι-calcul

Les opérateurs CTL sont codés par des opérateurs du ι-calcul. CTL est une logique monadique. Toutes ses formules caractérisent donc des relations unaires que l'on code sur la suite d'indice $\langle 1 \rangle$. On peut donc garder cette dernière implicite dans les définitions qui suivent.

– Les formules `true` et `false` sont codées par les opérateurs 0-aires de mêmes noms qui retournent respectivement les relations V et $\emptyset$.
– Les prédicats unaires `v` sont codés par des opérateurs 0-aires `vertex[v]` qui retournent le singleton $\{v\}$ ($v \in V$).
– La négation est codée par un opérateur unaire `not` : $\mathtt{not}(r) = \complement_V r$.
– La conjonction est codée par un opérateur binaire `and` : $\mathtt{and}(r_1, r_2) = r_1 \cap r_2$ (de même on introduit l'opérateur `or` pour coder la conjonction).
– L'opérateur `EX` est codé par un opérateur unaire de même nom, qui a tout $r \subseteq V$ fait correspondre l'ensemble suivant : $\mathtt{EX}(r) = \{s \in V ; \exists t \in V \ t.q. \ (s,t) \in E \wedge t \in r\}$.
– Pour coder les opérateurs `EF`, `EG`, `E(.U.)` et `A(.U.)`, on a besoin d'introduire une variable auxiliaire R. Les formules $\mathtt{EF}(f)$, $\mathtt{EG}(f)$, $\mathtt{E}(f\mathtt{U}g)$ et $\mathtt{A}(f\mathtt{U}g)$ sont codées par les programmes suivants :

$$\begin{aligned} \mathtt{EF}(f) &: \iota\,[= \mathtt{false.or}*]\,\epsilon\,\{R = \mathtt{or}(F, \mathtt{EX}(R))\} \\ \mathtt{EG}(f) &: \iota\,[= \mathtt{true.and}*]\,\epsilon\,\{R = \mathtt{and}(F, \mathtt{EX}(R))\} \\ \mathtt{E}(f\mathtt{U}g) &: \iota\,[= \mathtt{false.or}*]\,\epsilon\,\{R = \mathtt{or}(G, \mathtt{and}(F, \mathtt{EX}(R)))\} \\ \mathtt{A}(f\mathtt{U}g) &: \iota\,[= \mathtt{true.and}*]\,\epsilon\,\{R = \mathtt{or}(G, \mathtt{and}(F, \mathtt{AX}(R)))\} \end{aligned}$$

où F et G sont respectivement les codages des formules f et g.

Dans les outils de vérification de modèles, plusieurs codages ont été utilisés pour stocker la valeur des formules CTL. Ainsi, dans l'outil MEC [2, 3] qui met en œuvre un μ-calcul sans point fixe imbriqué (voir section 4), le stockage se fait par marquage des sommets du graphe. Ce dernier est explicitement stocké. Ce codage permet d'introduire des primitives extra-logiques comme l'opérateur `loop(.,.)` de MEC qui construit l'ensemble des sommets appartenant à une composante fortement connexe vérifiant certaines conditions. Cet opérateur compense, au moins en pratique, l'absence de point fixe imbriqué. Un tel prédicat pourraît être ajouter sans problème en tant qu'opérateur du ι-calcul.

Dans l'outil SMV [15], le graphe et la valeur des formules CTL sont codés symboliquement via des fonctions booléennes elles-même codées par des diagrammes binaires de décision. Ce codage permet de stocker des graphes gigantesques [4] pourvu qu'ils présentent certaines régularités.

3.3 Exemples

Considérons le graphe G représenté Fig. 1.

La propriété "être un sommet d'indice pair" est définie par la formule suivante :

$$\mathtt{or}(\mathtt{vertex}[2], \mathtt{or}(\mathtt{vertex}[4], \mathtt{vertex}[6]))$$

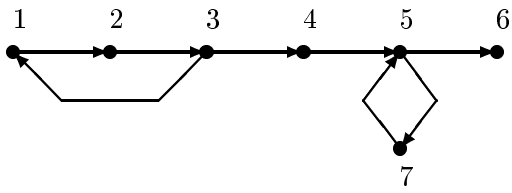

FIG. 1 – Un graphe

Pour des raisons de commodité, on peut nommer cette relation via un constructeur ι. On écrit le programme P_1 suivant :

$$P_1 \quad : \quad \iota\ [= \texttt{false.or}*]\ \epsilon\ \{Pair = \texttt{or}(\texttt{vertex}[2], \texttt{or}(\texttt{vertex}[4], \texttt{vertex}[6]))\}$$

Les prédécesseurs d'un sommet d'indice pair sont caractérisés par la formule $\texttt{EF}(Pair)$. On écrit le programme P_2 suivant :

$$P_2 \quad : \quad P_1; \iota\ [= \texttt{false.or}*]\ \epsilon\ \{PrecPair = \texttt{or}(Pair, \texttt{EX}(PrecPair))\}$$

Les sommets qui sont sur des chemins infinis sur lesquels il est toujours possible d'atteindre un sommet d'indice pair sont caractérisés par la formule CTL $\texttt{AG}(\texttt{EX}(\texttt{EF}(Pair)))$. Littéralement, cette formule se lit : "toujours, il existe un successeur à partir duquel on peut joindre un sommet d'indice pair". Le EX garantit l'existence d'un successeur, donc d'un chemin infini. En ι-calcul, cela donne le programme P_3 suivant :

$$P_3 \quad : \quad P_2; \iota\ [= \texttt{true.and}*]\ \epsilon\ \{InfPrecPair = \texttt{and}(PrecPair, \texttt{EX}(InfPrecPair))\}$$

On remarque que le programme P_3 ne permet pas de caractériser les états qui sont sur des chemins passant infiniment souvent par un état d'indice pair. En fait, cette propriété n'est pas expressible dans la logique CTL car elle requiert le calcul de deux points fixes imbriqués. Elle est en revanche expressible en ι-calcul, par exemple avec le programme suivant :

$$\begin{array}{lll} P_4 & : & P_1\ ; \\ & & \quad \iota\ \ [= \texttt{true.and}*] \\ & & \quad\ \ \iota\ [= \texttt{false.or}*]\ \epsilon\ \{PrecPair = \texttt{or}(\texttt{and}(Pair, InfPair), \texttt{EX}(PrecPair)))\} \\ & & \quad\ \ \{InfPair = \texttt{EX}(PrecPair)\} \end{array}$$

Sur le graphe de la Fig. 1, l'historique du calcul est le suivant :

Étape	$InfPair$	$PrecPair$	Action
1	$\{1,2,3,4,5,6,7\}$		initialisation de $InfPair$
2	$\{1,2,3,4,5,6,7\}$	$\emptyset$	initialisation de $PrecPair$
3	$\{1,2,3,4,5,6,7\}$	$\{2,4,6\}$	itération sur $PrecPair$
4	$\{1,2,3,4,5,6,7\}$	$\{1,2,3,4,5,6\}$	itération sur $PrecPair$
5	$\{1,2,3,4,5,6,7\}$	$\{1,2,3,4,5,6,7\}$	itération sur $PrecPair$
6	$\{1,2,3,4,5,6,7\}$	$\{1,2,3,4,5,6,7\}$	itération sur $PrecPair$
7	$\{1,2,3,4,5,7\}$	$\{1,2,3,4,5,6,7\}$	itération sur $InfPair$
8	$\{1,2,3,4,5,7\}$	$\{1,2,3,4,5,6,7\}$	itération sur $InfPair$
9	$\{1,2,3,4,5,7\}$	$\emptyset$	initialisation de $PrecPair$
10	$\{1,2,3,4,5,7\}$	$\{2,4\}$	itération sur $PrecPair$
11	$\{1,2,3,4,5,7\}$	$\{1,2,3,4\}$	itération sur $PrecPair$
12	$\{1,2,3,4,5,7\}$	$\{1,2,3,4\}$	itération sur $PrecPair$
13	$\{1,2,3\}$	$\{1,2,3,4\}$	itération sur $InfPair$
13	$\{1,2,3\}$	$\{1,2,3,4\}$	itération sur $InfPair$
14	$\{1,2,3\}$	$\emptyset$	initialisation de $PrecPair$
15	$\{1,2,3\}$	$\{2\}$	itération sur $PrecPair$
16	$\{1,2,3\}$	$\{1,2\}$	itération sur $PrecPair$
17	$\{1,2,3\}$	$\{1,2,3\}$	itération sur $PrecPair$
18	$\{1,2,3\}$	$\{1,2,3\}$	itération sur $PrecPair$
19	$\{1,2,3\}$	$\{1,2,3\}$	itération sur $InfPair$

4 Mise en œuvre du μ-calcul propositionnel

4.1 Le μ-calcul propositionnel

Le μ-calcul propositionnel [14, 1] est très semblable au ι-calcul, dont il est une spécialisation. Il est construit à partir de deux ensembles de symboles de variables :

- Un ensemble de variables propositionnelles $\mathcal{X} = \{x_1, \dots\}$.
- Un ensemble de variables relationnelles $\mathcal{R} = \{R_1, \dots\}$. À chaque variable de relation R, on associe son arité $\alpha(R)$.

Les formules de base du μ-calcul sont les formules propositionnelles quantifiées augmentées des appels de relations :

- Les deux constantes `0` et `1` sont des formules.
- Les variables propositionnelles sont des formules.
- Si F et G sont des formules et x est une variable alors $F \vee G$, $F \wedge G$, $\neg F$, $\exists x.F$ et $\forall x.F$ sont des formules.
- Si R est une variable relationnelle d'arité n et x_1, ..., x_n sont des variables propositionnelles alors $R(x_1, \dots, x_n)$ est une formule.

On note $var(F)$ les variables libres (au sens usuel) d'une formule F.

Les formules du μ-calcul propositionnelles sont les formules de base augmentées des définitions de points fixes :

- Les formules de base sont des formules.
- Si R est un symbole de relation d'arité n et F est une formule alors $\mu R.F$ et $\nu R.F$ sont des formules.

Une variable de relation R est liée dans une formule F si toutes ses occurrences dans une sous-formule $\mu R.G$ ou $\nu R.G$ de F. Elle est libre sinon.

4.2 Codage du μ-calcul propositionnel dans le ι-calcul

Dans le μ-calcul booléen, le domaine d'interprétation est $\{0, 1\}$. Par conséquent, chaque indice représente une variable booléenne ; les contraintes et les relations d'arité n sont des sous-ensembles de $\{0, 1\}^n$.

Les contraintes : On définit un certain nombre de contraintes en donnant leurs indiçages et leurs valeurs :

- Vrai ($X = 1$) : `true` $= (\langle 1\rangle, \{1\})$.
- Égalité ($X = Y$) : `eql` $= (\langle 1,2\rangle, \{00, 11\})$.
- Négation ($X = \neg Y$) : `not` $= (\langle 1,2\rangle, \{01, 10\})$.
- Conjonction ($X = Y \wedge Z$) : `and` $= (\langle 1,2,3\rangle, \{000, 001, 010, 111\})$.
- Implication ($X = Y \Rightarrow Z$) : `imp` $= (\langle 1,2,3\rangle, \{010, 100, 101, 111\})$.
- $\vdots$

L'opérateur `rename` : On définit une famille d'opérateurs unaires qui permettent de changer les indices sur lesquels portent une formule. L'opérateur `rename`$[j_1/i_1, \ldots, j_s/i_s]$ change l'indice i_1 en l'indice j_1, l'indice i_2 en l'indice j_2 et ainsi de suite. Dans la mesure où les suites d'indices sont supposées ordonnées, ce changement d'indices peut entrainer un changement du codage de la relation associée à la formule considérée.

Par exemple, `rename`$[4/1, 6/3]$(`and`) $= (\langle 2,4,6\rangle, \{000, 001, 100, 111\})$.

Les opérateurs ensemblistes : On définit les opérateurs ensemblistes usuels : union $\cup$, intersection $\cap$, différence $\setminus$, complémentation $\complement$.

Les trois premiers sont des opérateurs binaires. Leur fonction d'indiçage est la jointure $\bowtie$ des suites d'indices : $I_1 \bowtie I_2$ est la suite ordonnée obtenue après union des ensembles d'indices apparaissant dans I_1 et I_2. Par exemple, $\langle 1,2,5\rangle \bowtie \langle 2,3,6\rangle = \langle 1,2,3,5,6\rangle$. La fonction d'indiçage de la complémentation est simplement l'identité.

Le calcul de $(I_1, V_1) \cup (I_2, V_2)$ se fait donc en deux temps : on étend d'abord V_1 et V_2 en des relations sur $I_1 \bowtie I_2$ (en complétant les n-uplets avec toutes les valeurs possibles des indices manquants), puis on prend l'union des deux ensembles de n-uplets (idem pour $\cap$ et $\setminus$).

Par exemple :

- $\complement$(`and`) $= (\langle 1,2,3\rangle, \{011, 100, 101, 110\})$.
- $\cap$(`eql`, `rename`$[3/2]$(`eql`)) $= (\langle 1,2,3\rangle, \{000, 001, 110, 111\} \cap \{000, 010, 101, 111\})$
  $= (\langle 1,2,3\rangle, \{000, 111\})$

Les quantificateurs : On définit les quantificateurs comme des opérateurs unaires. De la même façon que pour l'opérateur `rename`, on considère en fait des familles de quantificateurs.

Ainsi, l'effet de $\forall[i](f)$ est d'ôter l'indice i de la suite d'indices de la formule f et de projeter universellement la relation associée à f sur les variables restantes, ou autrement dit, de calculer l'intersection de l'ensemble des n-uplets de f dans lesquels l'indice i vaut 0 et de l'ensemble des n-uplets de f dans lesquels l'indice i vaut 1.

Par exemple, $\forall[1]$(`and`) $= (\langle 2,3\rangle, \phi)$ et $\forall[3]$(`and`) $= (\langle 1,2\rangle, \{00\})$.

Les opérateurs de points fixes Le plus petit point fixe μ et le plus grand point fixe ν se programment à l'aide de la construction ι. Dans les deux cas, le test est l'égalité. L'opérateur 0-aire permettant d'initialiser le calcul fait correspondre la relation vide (sur la suite d'indices de longueur n considérée) dans le cas du plus petit point fixe, et la relation pleine, c'est à dire $\{0,1\}^n$, dans le cas du plus grand point fixe. Le squelette de calcul est $\cup*$ dans le cas du plus petit point fixe et $\cap*$ dans le cas du plus grand point fixe.

4.3 Exemple

Considérons la définition Prolog de la relation `eclate(Xs,Ys,Zs)` qui éclate la liste `Xs` en deux listes `Ys` et `Zs` de longueur sensiblement égale.

```
eclate([],[],[]).
eclate([X|Xs],[X|Ys],Zs) :- eclate(Xs,Zs,Ys).
```

La version booléenne de cette définition permet une analyse de clôture de ce code :

```
eclate(1,1,1).
eclate(X & Xs, X & Ys, Zs) :- eclate(Xs,Zs,Ys).
```

que nous pouvons calculer par le programme suivant :

$$
\begin{aligned}
P : \iota\ [= \texttt{false.or}*]\ \epsilon \\
\{eclate = \quad & (\texttt{true}(1) \cap \texttt{true}(2) \cap \texttt{true}(3)) \bigcup \\
& \exists[4,5,6,7](\quad \texttt{and}(1,7,4) \cap \texttt{and}(2,7,6) \cap \texttt{eql}(3,5) \\
& \cap \texttt{rename}[4/1, 5/2, 6/3](eclate))\}
\end{aligned}
$$

pour conclure :

$$eclate = (\langle 1,2,3\rangle, \{000, 001, 010, 111\})$$

Ce résultat s'interprète comme suit : après chaque réfutation du but `:- eclate(L1,L2,L3).`, les trois listes sont soit toutes closes (i.e. ne contiennent pas de variable), soit `L1` est non-close et au plus une des deux listes `L2` ou `L3` est close.

5 Interprétation abstraite de PLC(N)

Dans cette section, nous discutons de deux instanciations du ι-calcul pour inférer des propriétés de programmes logiques avec contraintes sur le domaine N, l'ensemble des entiers naturels. Nous supposons que le vocabulaire de notre langage comprend les fonctions $+$, 0, 1, ...et les relations $=$ et $\geq$, habillées de leurs sémantiques habituelles.

5.1 Par polyhèdres rationnels

Pour ce calcul, le domaine d'interprétation est Q, l'ensemble des nombres rationnels. Chaque indice représente une variable à valeur dans Q (nous ne le répétons pas dans les définitions qui suivent) ; les contraintes et relations d'arité n sont des sous-ensembles de Q^n.

Les contraintes : On définit un certain nombre de contraintes en donnant leurs indiçages et leurs valeurs :

- Zéro ($X = 0$) : `eq0` $= (\langle 1\rangle, \{0\})$.
- Un ($X = 1$) : `eq1` $= (\langle 1\rangle, \{1\})$.
- Égalité ($X = Y$) : `eql` $= (\langle 1,2\rangle, \{(x,x)\})$.
- Supérieur ou égal ($X \geq Y$) : `geq` $= (\langle 1,2\rangle, \{(x,y) | x \geq y\})$.
- Somme ($X = Y + Z$) : `add` $= (\langle 1,2,3\rangle, \{(x,y,z) | x = y + z\})$.
- $\vdots$

Les opérateurs `false` et `true` : Les opérateurs 0-aires `false` et `true` définissent respectivement les relations vides $(\langle \ldots \rangle, \phi)$ et les relations pleines $(\langle 1, \ldots, n\rangle, Q^n)$.

L'opérateur $\exists_{\_[\ldots]}$: On définit une famille d'opérateurs unaires qui permettent de projeter une formule sur une suite donnée d'indices. Par exemple, $\exists_{\_[1]}(\texttt{add}) = (\langle 1\rangle, \{x\})$.

L'opérateur `rename` : On définit une famille d'opérateurs unaires qui permettent de changer les indices sur lesquels portent une formule, comme décrit dans la section 4.1.

Les opérateurs ensemblistes binaires : On définit l'opérateur intersection $\cap$ comme indiqué section 4.1. L'opérateur d'union est remplacé par l'enveloppe convexe $\uplus$. Il n'est défini que pour deux relations d'indiçage identique et calcule l'enveloppe convexe des deux ensembles de valeurs. Par exemple, `eq0` $\uplus$ `eq1` $= (\langle 1\rangle, \{x | 0 \leq x \leq 1\})$. Enfin, pour assurer la finitude des calculs, l'opérateur d'élargissement $\bigtriangledown$, introduit initialement dans [10], n'est défini que pour deux relations R_1 et R_2 d'indiçage identique, dont l'ensemble des valeurs est exprimable (au moins pour R_1) comme conjonction finie d'équations et inéquations linéaires, i.e. $R_1 = \{(x_1, \ldots, x_n) | c_1, \ldots, c_k\}$. Alors $R_1 \bigtriangledown R_2 = \{(x_1, \ldots, x_n) | c_{i_1} \ldots c_{i_l}\}$ où seuls les c_{i_j} incluant (au sens ensembliste) R_2 sont sélectionnées. Par exemple, posons $R_1 = (\langle 1, 2\rangle, \{(x, y) | x \geq 1, 0 \geq y\})$ et $R_2 = (\langle 1, 2\rangle, \{(x, y) | x \geq 2, y \geq 1\})$. On a : $R_1 \bigtriangledown R_2 = (\langle 1, 2\rangle, \{(x, y) | x \geq 1\})$.

Un exemple : Nous reprenons l'exemple de la section 4.3, en nous intéressant cette fois à la taille des listes, définie par $|[]| = 0$ et $|[X|Xs]| = 1 + |Xs|$. La définition PLC(N) de la relation `eclate(Xs,Ys,Zs)` devient alors :

```
eclate(0,0,0).
eclate(1+Xs,1+Ys,Zs) :- eclate(Xs,Zs,Ys).
```

Nous pouvons calculer des points fixes (c.à.d. des propriétés[2]) de cette relation par la famille $(P_n)_{n \in N}$ de programmes suivants :

$$
\begin{aligned}
P_n : \iota\ [= \texttt{false}.\uplus^n.\ \bigtriangledown *]\ \epsilon \\
\{\mathit{eclate} = \quad & (\texttt{eq0}(1) \cap \texttt{eq0}(2) \cap \texttt{eq0}(3)) \bigcup \\
& \exists_{-[1,2,3]}(\qquad \texttt{eq1}(7) \cap \texttt{add}(1,4,7) \cap \\
& \qquad\qquad\quad \texttt{and}(2,6,7) \cap \texttt{eql}(3,5), \\
& \qquad\qquad\quad \texttt{rename}[4/1, 5/2, 6/3](\mathit{eclate}))\}
\end{aligned}
$$

et obtenir, par évaluation des P_n :

n	*eclate*
0	`true`
1	`true`
2	`true`
3	`add`
4,...	$(\langle 1, 2, 3\rangle, \{(x, y, z) \mid x \in Q, y \in Q, z \in Q, x = y + z, y \geq z \geq 0, z + 1 \geq y\})$

Les trois premiers programmes ($0 \leq n \leq 2$) ne génèrent aucune information utile. À partir de $n = 3$, on sait qu'après chaque réfutation du but `eclate(L1,L2,L3)`, $|L1| = |L2| + |L3|$. Le programme P_4 nous apprend que $|L2|$ est toujours supérieur ou égal à $|L3|$, mais au plus d'une unité, ce qui précise l'expression "sensiblement égale" employée section 4.3. Puis, pour les n supérieurs à 5, le modèle inféré reste identique. Nous sommes donc en présence d'un méta-point fixe : P_4 calcule la "meilleure" approximation rationnelle pour *eclate*. Ce n'est pas toujours le cas (considérer par exemple les approximations rationnelles de la relation non-linéaire $Y = X^2$). Pour conclure, notons qu'un raisonnement par cas sur la parité de $|L1|$, hors de portée du cadre de cette section, permet d'affiner le modèle numérique de la relation *eclate* et que nous sommes encore loin du sens de la version originale de cette relation !

5.2 Par intervalles entiers

Pour ce dernier calcul, le domaine d'interprétation est $\bar{Z}$, l'ensemble des entiers relatifs, augmenté de $\{-\infty, +\infty\}$. L'addition et les relations d'égalité et d'inégalité sont étendues en conséquence. Cette section est inspirée de l'exemple 32 de [9].

[2] Les formules $\forall[\mathit{Prog} \Rightarrow \mathit{Prop}]$ sont valides dans Q, donc *a fortiori* valides dans N.

Les contraintes : On définit une famille de contraintes unaires (min et max sont des éléments de $\bar{Z}$) :

- Intervalle ($in[min, max](X)$) : $\texttt{in}[\texttt{min},\texttt{max}] = (\langle 1\rangle, \{x | min \leq x \leq max\})$.
- Somme ($X = Y + Z$) : $\texttt{add} = (\langle 1,2,3\rangle, \{(x,y,z) | x = y + z\})$.

Les opérateurs `false` et `true` : Les opérateurs 0-aires `false` et `true` définissent respectivement les relations vides ($\langle \ldots \rangle, \phi$) et les relations pleines ($\langle 1,\ldots,n\rangle, \bar{Z}^n$).

L'opérateur $\exists_{\_[\ldots]}$: On définit une famille d'opérateurs unaires qui permettent de projeter une formule sur une suite donnée d'indices.

L'opérateur `rename` : On définit une famille d'opérateurs unaires qui permettent de changer les indices sur lesquels portent une formule.

Les opérateurs ensemblistes binaires : On définit les opérateur intersection $\cap$ et union $\cup$ comme indiqué section 4.1.

Pour assurer la finitude des calculs, l'opérateur d'élargissement $\triangledown$ n'est défini que pour deux relations R_1 et R_2 d'indiçage identique, dont l'ensemble des valeurs est exprimable (pour R_1 et R_2) comme conjonction finie d'inéquations de la forme $min_i \leq x_i \leq max_i$ pour chaque variable indiçée x_i. Soient $R_1 = \{(x_1,\ldots,x_n) | c_1^1,\ldots,c_n^1\}$ et $R_2 = \{(x_1,\ldots,x_n) | c_1^2,\ldots,c_n^2\}$. Alors $R_1 \triangledown R_2 = \{(x_1,\ldots,x_n) | \ldots, c_i^1 \triangledown c_i^2, \ldots\}$. Définissons à présent cet opérateur d'élargissement pour deux intervalles. Il vérifie $I \triangledown \phi = \phi \triangledown I = I$ et $[a,b] \triangledown [c,d] = [si\ c < a\ alors\ -\infty\ sinon\ a,\ si\ d > b\ alors\ +\infty\ sinon\ b]$.

Pour améliorer la précision des calculs, on introduit l'opérateur de rétrécissement $\triangle$, défini comme ci dessus en posant $I \triangle \phi = \phi \triangle I = \phi$ et $[a,b] \triangle [c,d] = [si\ a = -\infty\ alors\ c\ sinon\ a,\ si\ b = +\infty\ alors\ d\ sinon\ b]$.

Un exemple Considérons la définition PLC(N) suivante :

```
p(7).
p(X) :- X =< 77, X = Y+2, p(Y).
```

et le programme $P(\sigma)$ associée :

$$
\begin{array}{l}
P(\sigma) : \iota\ [= \texttt{false}.\sigma]\ \epsilon \\
\qquad \{p = \quad (\texttt{in}[7,7](1) \bigcup \\
\qquad\qquad \exists_{-[1]}(\qquad \texttt{in}[-\infty,77](1) \cap \texttt{in}[2,2](3) \cap \texttt{add}(1,2,3) \cap \\
\qquad\qquad\qquad\qquad \texttt{rename}[2/1](p))\}
\end{array}
$$

On obtient :

σ	p
$\triangledown^*$	$(\langle 1\rangle, \{x \mid 7 \leq x \leq +\infty\})$
$\triangledown^*.\triangle^*$	$(\langle 1\rangle, \{x \mid 7 \leq x \leq 77\})$

Notons que la preuve de la requête `:- p(X).` termine dans PLC(N) et ne termine pas dans PLC(Z) (au sens de la terminaison universelle). En revanche, dans la mesure où un squelette de calcul se conclut par la séquence $\triangledown^*$, la terminaison de l'évaluation de $P(\sigma)$ est assurée. Remarquons également le gain en précision obtenu grâce à l'opérateur de rétrécissement.

6 Conclusion

Dans cet article, nous avons jeté les bases syntaxiques et sémantiques d'un nouveau langage générique de contraintes d'ordre supérieur : le ι-calcul. Nous avons illustré la polyvalence du langage en détaillant quatre instances du ι-calcul. Nous avons prototypé le noyau du ι-calcul et chacune de ces quatre instances en Prolog. La compacité du code Prolog (environ 1400 lignes de codes au total) doit bien sûr beaucoup à la richesse des bibliothèques proposées par Sicstus Prolog.

De par nos expériences respectives, nous sommes persuadés que le ι-calcul serait d'une grande utilité pour la réalisation d'outils sémantiques d'aide à la programmation. Il reste néanmoins beaucoup à faire pour, d'une part affiner les bases théoriques d'un tel langage, et d'autre part sélectionner les structures de données adéquates pour une implantation efficace du ι-calcul et de ses dérivés.

A Une mise en œuvre en Prolog

Le programme Sicstus Prolog suivant est une mise en œuvre complète du calcul de $T_{D,I}[P,\rho]$, tel qu'il a été défini section 2.3. Il utilise le module de tables associatives de Sicstus. Les contraintes, les relations, les opérateurs et les tests, qui dépendent du domaine d'interprétation D, sont supposés définis dans le module `D` passé en paramètre. Le lecteur remarquera que ce court programme suit trait pour trait, aux prédicats utilitaires près, la sémantique du ι-calcul.

```
:- module(iota,[iota/3]).
:- use_module(library(assoc)).

%%% evalProgram(-Domain,-Program,+Rho)
evalProgram(D,P,Rho) :-
        empty_assoc(Rho0),
        evalProgram(P,D,Rho0,Rho).

%%% evalProgram(-Program,-Domain,-RhoIn,+RhoOut)
evalProgram([],_D,RhoIn,RhoOut).
evalProgram([P|Ps],D,RhoIn,RhoOut) :-
        evalIota(P,D,RhoIn,Rho1),
        evalProgram(Ps,D,Rho1,RhoOut).

%%% evalIota(-iota(Test,Squeleton,Q,Equations),-D,-RhoIn,+RhoOut)
evalIota(iota(T,[Op|Ops],Q,E),D,RhoIn,RhoOut) :-
        D:clause(operation(Op,[],_),_),  % 0-ary operator
        initSubst(E,D,Op,RhoIn,Rho1),
        evalIota(iota(T,Ops,Q,E),D,Rho1,RhoOut).
evalIota(iota(T,S,Q,E),D,RhoIn,RhoOut) :-
        popSqueleton(S,Op,Ops),
        evalProgram(Q,D,RhoIn,Rho1),
        updateSubst(E,D,Op,Rho1,Rho2),
        testFixPoint(E,D,T,Rho1,Rho2,Reached),
        (Reached=1 -> RhoOut=Rho1 ; evalIota(iota(T,Ops,Q,E),D,Rho2,RhoOut)).

%%% popSqueleton(-SqueletonIn,+Op,+SqueletonOut)
popSqueleton([Op|Ops],Op,Ops) :-
        D:clause(operation(Op,[_,_],_),_). % 2-ary operator
popSqueleton([star(Op)],Op,[star(Op)]) :-
        D:clause(operation(Op,[_,_],_),_). % stared 2-ary operator

%%% initSubst(-Eqs,-Domain,-Op,-RhoIn,+RhoOut)
```

```
initSubst([],_D,_O,Rho,Rho).
initSubst([R=_F|Eqs],D,Op,RhoIn,RhoOut) :-
        D:relation(R,_Ar,Ind),
        D:operation(Op,[],Ind-Rel),
        put_assoc(R,RhoIn,Ind-Rel,Rho1),
        initSubst(Eqs,D,Op,Rho1,RhoOut).

%%% updateSubst(-Eqs,-Domain,-Op,-RhoIn,+RhoOut)
updateSubst([],_D,_O,Rho,Rho).
updateSubst([R=F|Eqs],D,Op,RhoIn,RhoOut) :-
        evalForm(R,D,RhoIn,ValR), evalForm(F,D,RhoIn,ValF),
        D:operation(Op,[ValR,ValF],Val),
        put_assoc(R,RhoIn,Val,Rho1),
        updateSubst(Eqs,D,Op,Rho1,RhoOut).

%%% testFixPoint(-Eqs,-Domain,-Test,-Rho1,-Rho2,+Reached)
testFixPoint([],_D,_T,_Rho1,_Rho2,1).
testFixPoint([R=_F|Eqs],D,T,Rho1,Rho2,Reached) :-
        evalForm(R,D,Rho1,Val1), evalForm(R,D,Rho2,Val2),
        D:test(T,Val1,Val2,Reached1),
        (Reached1=0 -> Reached=0 ; testFixPoint(Eqs,D,T,Rho1,Rho2,Reached)).

%%% evalForm(-Formula,-Domain,-Rho,+Ind-Rel)
evalForm(C,D,_R,Ind-Rel) :-
        D:constraint(C,_Ar,Ind,Rel).
evalForm(R,D,Rho,Ind-Rel) :-
        D:relation(R,_Ar,Ind),
        get_assoc(R,Rho,Ind-Rel).
evalForm(op(Op,Args),D,Rho,Val) :-
        evalArgs(Args,D,Rho,Vals),
        D:operation(Op,Vals,Val).

evalArgs([],_D,_Rho,[]).
evalArgs([A|As],D,Rho,[V|Vs]) :- evalForm(A,D,Rho,V), evalArgs(As,D,Rho,Vs).
```

À titre d'exemple, nous montrons comment mettre en œuvre la logique CTL (voir section 3) en définissant un module destiné à être utilisé conjointement au programme ci-dessus. Deux prédicats manquent pour que ce module soit complet : le prédicat `vertices(+L)` qui donne la liste des sommets du graphe considéré et le prédicat `predecessors(+Vs,-Ps)` qui donne la liste `Ps` des prédécesseurs des sommets de la liste `Vs`. La valeur d'une formule est représentée par la liste ordonnée des sommets la satisfaisant.

```
:-module(ctl,[]).
:-use_module(library(ordsets)).

%%% constraint(Name,Arity,Indexing,Value).
constraint(vertex(V),1,[1],[V]).

%%% relation(Symbol,Arity,Indexing).
relation(pair,1,[1]).
relation(precPair,1,[1]).
relation(infPair,1,[1]).

%%% test(Name,Indexing-Value1,Indexing-Value2,Vueal).
test(equal,[1]-Val1,[1]-Val2,Res):- (Val1=Val2 -> Res=1 ; Res=0).
```

```
%%% operation(Name,Args,Indexing-Value).
:- dynamic operation/3. % to make it possible to use clause().
operation(false,[],[1]-[]).
operation(true,[],[1]-V)              :- vertices(V).
operation(not,[[1]-V1],[1]-V)         :- vertices(Vs), ord_difference(Vs,V1,V).
operation(or,[[1]-V1,[1]-V2],[1]-V3)  :- ord_union(V1,V2,V3).
operation(and,[[1]-V1,[1]-V2],[1]-V3) :- ord_intersection(V1,V2,V3).
operation(ex,[[1]-L],[1]-V)           :- predecessors(L,V).
```

Références

[1] H.R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126 :3–30, 1994.

[2] A. Arnold. MEC : a System for Constructing and Analysing Transition Systems. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*. Springer Verlag, June 1989.

[3] A. Arnold, D. Bégay, and P. Crubillé. *Construction and analysis of transition systems with MEC*. World Scientific Publishers, 1994. ISBN 981-02-1922-9.

[4] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking : 10^{20} States and Beyond. *IEEE transactions on computers*, 1990.

[5] E.M. Clarke, E.A. Emmerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Trans. on Prog. Lang Syst*, 8 :244–263, 1986.

[6] A. Colmerauer. An introduction to Prolog III. *CACM*, 33(7) :70–90, July 1990.

[7] A. Colmerauer, H. Kanoui, and M. Van Caneghem. Prolog, bases théoriques et développements actuels. *TSI*, 2(4), 1983.

[8] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximations of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[9] P. Cousot and R. Cousot. Abstract interpretation and applications to logic programs. *J. Logic Programming*, 13 :103–180, 1992.

[10] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of the 5th ACM Symposium on Principles of Programming Languages*, pages 84–96, 1978.

[11] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publisher, 1993. ISBN 0-7923-9311-2.

[12] J. Jaffar and J.L. Lassez. Constraint logic programming. Technical Report 74, Monach University, Australia, 1986.

[13] J. Jaffar and M.J. Maher. Constraint logic programming : a survey. *J. Logic Programming*, pages 503–581, 1994.

[14] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27 :333–354, 1983.

[15] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993. ISBN 0-7923-9380-5.

[16] F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs. In *Proc. of JICSLP'96*, pages 7–21. MIT Press, 1996.

[17] D. Park. Finitiness is Mu-ineffable. *Theory of Computation*, 3, 1974.