Connexions entre Complexité Descriptive et Programmation Logique par Contraintes

Lucas Bordeaux
IRIN*
bordeaux@irin.univ-nantes.fr

Eric Monfroy
IRIN*
monfroy@irin.univ-nantes.fr

Résumé

La Complexité Descriptive (CD) étudie des langages logiques qui expriment exactement les problèmes de certaines classes de complexité. Ses enjeux sont principalement théoriques, mais il est légitime de se demander si ces logiques ont un intérêt pratique en tant que langages de programmation. Puisque ces langages permettent d'exprimer - de manière parfois extrêmement naturelle - des problèmes combinatoires (polynomiaux, NP, ...), il est envisageable de les considérer comme des langages de programmation logique par contraintes. Nous présentons les résultats de Complexité Descriptive qui nous paraissent significatifs pour la communauté PLC, et nous comparons en particulier l'approche PLC à un fragment de logique du second ordre discuté en CD. Nous montrons ainsi des connexions frappantes entre les deux domaines. Nous proposons un aperçu de la diversité des perspectives issues de la CD pour la résolution de problèmes, la conception de langages, et l'analyse de traitabilité.

1 Introduction

Si la théorie de la complexité permet de déterminer la *traitabilité des problèmes*, elle apporte également des outils pour classifier l'*expressivité des langages*. Cet aspect de la complexité a émergé avec un résultat de Ronald Fagin [15] identifiant NP (*c-à-d.* la classe des "problèmes de satisfaction de contraintes") avec l'ensemble des problèmes expressibles dans un certain fragment de logique à modèles finis. Des correspondances similaires ont par la suite été démontrées entre les principales classes de complexité et d'autres fragments de logique. Cette approche de la théorie de la complexité est appelée *Complexité Descriptive*[1] [21, 20, 12, 14, 19].

*Institut de Recherche en Informatique de Nantes - 2 rue de la Houssinière - BP. 92208 - 44322 Nantes cedex 3 - France

[1]Il semble toutefois que l'emploi de ce terme soit sujet à ambiguïté : certains auteurs [33] l'utilisent comme synonyme de *complexité de Kolmogorov*, et Fagin [14] lui donne un sens légèrement différent.

De manière orthogonale et indépendante, une longue tradition de recherche en Intelligence Artificielle a été consacrée à la conception d'outils pour la résolution de problèmes, avec un intérêt particulier pour les problèmes combinatoires de la classe NP. Un nombre important de propositions ont été faites pour définir des *langages* permettant d'exprimer ces problèmes, et des *systèmes* pour les résoudre. Les premières contributions remontant aux années 1970[2] avec REF-ARF [16] et ALICE [26]. Quand la programmation par contraintes est apparue, elle a tout naturellement trouvé son principal terrain d'application dans les problèmes combinatoires, et le succès des premiers résolveurs en Domaines Finis (*cf.* CHIP [35]) a ouvert la voie aux applications industrielles de cette technologie. Parmi les nombreux langages existants, on peut grossièrement distinguer ceux qui instancient ou étendent le cadre de la *Programmation Logique* [22, 35, 11, 38] de ceux qui ne sont pas explicitement fondés sur la logique mais plutôt sur la programmation mathématique [26, 17, 36, 39] ou les langages impératifs [16, 3, 34].

La Complexité Descriptive (CD) présente des liens evidents avec la Programmation Logique (PL) comme avec la Programmation par Contraintes en Domaines Finis (PC).

La CD et la PL ont en commun de relier logique et calcul suivant un paradigme "Formule = Programme" [2], par opposition à l'approche "Preuve = Programme" des langages fonctionnels. Alors que le langage Prolog permet d'exprimer l'ensemble des prédicats calculables, d'autres fragments de logique expriment des classes de problèmes plus restreints. Selon le cas, l'universalité de Prolog peut être vue comme un avantage ou un inconvénient puisqu'il y a finalement assez peu d'intérêt pratique à exprimer des problèmes non-élémentaires ou des calculs non-terminants.

La CD et la PC ont en commun de proposer des *langages formels* dans lesquels les problèmes combinatoires peuvent être formulés. La CD considère toutefois cette question d'un point de vue purement théorique, alors qu'en PC la tâche de formulation logique des problèmes est identifiée avec l'exercice même de programmation déclarative.

A notre connaissance, les connexions entre CD et PC sont encore mal connues et peu explorées. De manière significative, l'*expressivité* des différents langages de programmation par contraintes n'est jamais évoquée dans la littérature (une exception notable est cependant donnée par un rapport de recherche de Fages et al. [13], qui établit de nombreux résultats pour les langages concurrents à contraintes). Les langages de programmation par contraintes présentent en général une batterie d'outils syntaxiques pour définir des expressions numériques, des itérations sur des listes ou des matrices, une fonction-objectif, etc..., et il n'est pas évident de dégager un *noyau* suffisant de primitives capturant l'essence de ces langages. De plus, s'il est clair que ces constructions sont suffisantes pour exprimer n'importe-quel problème de programmation linéaire en nombres entiers, la modélisation d'un problème donné (portant sur des mots, des graphes, ou des bases de données) peut pour autant requérir un encodage sophistiqué.

L'exemple standard illustrant l'élégance des logiques à modèles finis est le problème de 3-colorabilité. Une modélisation *naturelle* de ce problème est donnée par la phrase

---

Nous nous référons ici à la signification donnée par le livre d'Immerman [21].

[2]On peut en fait probablement remonter plus loin, notamment jusqu'aux travaux de Newell, Shaw et Simon [32] - sans parler des travaux de programmation linéaire. Cependant les références mentionnées sont les premières, à notre connaissance, à proposer de véritables *langages* pour exprimer les problèmes.

logique suivante :

$$\begin{array}{l}\exists color : nodes \rightarrow \{red, yellow, blue\}, \\ \quad \forall x,\ \forall y,\ edge[x,y] \Rightarrow color[x] \neq color[y]\end{array} \tag{1}$$

Si la logique utilisée dans cette définition est très différente de celles généralement considérées en programmation logique classique, il s'agit néanmoins d'un langage formel parfaitement identifié – celui-là même défini par Fagin en 1974. Il se trouve également que les définitions écrites dans ce langage sont *exécutables* : on peut directement déduire un algorithme de reconnaissance de la propriété exprimée formellement dans ce langage. Une particularité importante des problèmes combinatoires est qu'ils portent sur des structures finies ((hyper)graphes, relations finies, nombres machine, matrices, ...), et la théorie logique adaptée à ces problèmes, ou *théorie des modèles finis*, présente de nombreuses différences avec la logique "classique" utilisée en PL (*cf.* [12, 14]). La logique de Fagin, ou *logique existentielle du second-ordre*, permet de définir tous les problèmes de la classe NP, quelle que soit la structure finie sur laquelle ceux-ci portent.

Finalement, nous voyons deux autres champs possibles d'application de la complexité descriptive à la PC. Le premier concerne la *traitabilité* des problèmes. La DC fournit des moyens syntaxiques de caractérisation des problèmes solubles en temps polynomial, puisque tous les problèmes de la classe P peuvent être exprimés dans un langage particulier. Le second concerne les problèmes d'une complexité supérieure à NP. Tous les problèmes d'Intelligence Artificielle ne sont pas dans NP, et certains ne peuvent donc être modélisés par les langages de PC classiques. Nous verrons brièvement que la DC fournit des extensions permettant de capturer un grand nombre de classes de complexité, y-compris au-delà de NP.

Le but de cet article est de convaincre le lecteur issu de l'une ou l'autre des communautés PL/PC de l'intérêt probable de la fertilisation croisée avec la DC. Notre argument principal est que la logique naturelle pour modéliser les problèmes combinatoires est une logique à modèles finis, différente de celle mise en œuvre dans Prolog par exemple. En particulier, nous argumentons les avantages de la logique existentielle de second-ordre comme langage de *modélisation* des problèmes combinatoires. Nous présentons un survol naïf des résultats de complexité descriptive qui nous paraissent significatifs pour la communauté PL/PC. Pour une présentation plus formelle et complète de ce domaine, nous renvoyons le lecteur aux ouvrages de référence [12, 21].

Travaux connexes

La CD a inspiré des résultats dans des domaines proches de la PLC. Dans le cadre de la programmation par règles, D. Mac Allester [27, 18] étend des résultats obtenus sur Datalog à un langage comportant des termes. Il montre que tous les problèmes de P sont expressibles dans certaines restrictions du langage de règles (critères de *superficialité* et de *localité*). A partir d'une extension de Datalog capturant exactement NP [8], l'équipe de M. Cadoli propose un "langage de spécification" des problèmes NP-complets, nommé NP-SPEC [7], qui est compilé vers SAT. Bien que ce langage soit moins proche des langages de modélisation usuels utilisés en PC, il présente de nombreuses idées communes avec notre approche.

Plan de l'article

Quelques rappels préliminaires sur les principales classes de complexité sont effectués en section 2, puis nous donnons une présentation informelle de quelques résultats de DC (section 3). Nous insistons sur une logique particulière appelée *logique existentielle de second ordre* (section 4), dont nous mettons en avant les similarités avec la PLC ; la section 5 discute des perspectives issues de la DC concernant la traitabilité et l'expressivité. Nous concluons alors en section 6.

2 Quelques notions de complexité

Nous commençons par quelques rappels concernant les classes de complexité [33], qui permettent de classer les problèmes rencontrés en PC et dans d'autres domaines de l'IA. Les ressources de base sont bien-sûr le *temps* et l'*espace*, et les problèmes sont classés en fonction de leurs besoins en termes de ces ressources. Pour ce qui nous concerne, les principales classes sont les suivantes :

$$LOGSPACE \subseteq P \subseteq PSPACE \subseteq EXP$$

représentant respectivement les problèmes solubles en espace logarithmique, en temps polynomial, en espace polynomial, et en temps exponentiel.

2.1 Quelques notions importantes

Le non-déterminisme repose sur l'intuition suivante : un problème appartient à la classe NP (*resp.* LOGSPACE, NEXP) si ses solutions peuvent être *vérifiées* en temps polynomial (*resp.* espace logarithmique, temps exponentiel). Par exemple, si une heuristique bien choisie exhibe un ordre de visite de sommets, ou un tuple d'entiers, il est facile de vérifier s'ils sont des solutions au problème posé (le problème de chemin hamiltonien et de programmation linéaire respectivement). Cette caractéristique explique le succès des méthodes locales pour résoudre les instances *solubles* des problèmes NP.

La complémentation exprime la négation de propriétés. Alors que les problèmes NP sont des problèmes d'existence d'une solution, les problèmes de co-NP expriment des propriétés universelles ($\forall$), telles la tautologie – il est donc facile de réfuter ces propriétés en exhibant un contre-exemple, et difficile de les prouver. La conjecture NP $\neq$ co-NP implique que les problèmes NP possèdent des instances insatisfaisables pour lesquels aucune preuve d'insatisfaisabilité de taille polynomiale n'existe (tout résolveur traitera donc ces instances en temps exponentiel).

Les problèmes de fonction permettent de donner un sens à la notion de *solution*, par opposition aux problèmes de décision où la réponse attendue est un simple booléen. Cette distinction est similaire à celle qui existe en PL : on n'attend pas seulement de Prolog qu'il réponde *oui* ou *non* à la requête posée, mais également qu'il retourne le résultat correspondant. Nous comprendrons par la suite les *problèmes de fonction* (*e.g.,* ceux de la classe FNP) comme les problèmes dans lequels le quantificateur $\exists$ est remplacé par le mot-clé `find`.

L'alternation est une généralisation du non-déterminisme qui autorise un usage arbitraire des quantificateurs. Entre NP et PSPACE, la *hiérarchie polynomiale* est un ensemble de classes construites par un nombre borné d'alternations de quantificateurs. Elle contient des classes telles F^{NP} ; intuitivement, certains problèmes demeurent P-complets en présence d'un oracle pour NP.

2.2 Problèmes

Différents types de problèmes sont en général considérés dans la littérature relative aux contraintes, et leur complexité n'est pas toujours donnée avec précision.

Le Problème de Satisfaction de Contraintes consiste à trouver des valeurs pour des variables soumises à des contraintes vérifiables en temps polynomial (formules logiques, equations linéaires, diséquations entre couleurs, etc...). Bien que peu d'auteurs le remarquent explicitement (une exception de taille étant [28]), les CSPs correspondent donc à la classe (F)NP. Dans certains articles, le CSP est défini comme le problème de "trouver une ou toutes les solutions, ou une solution optimale", mais ces trois problèmes sont en fait fort différents.

Les problèmes d'énumération sont évidemment très difficiles car de nombreux problèmes ont un nombre réellement exponentiel de solutions. Même le problème de *dénombrement* des solutions d'un CSP – ou le problème de décision associé consistant à déterminer si ce nombre est pair ou impair – est considéré comme beaucoup plus difficile que le problème de satisfaction de contraintes ($\sharp P$-complétude).

Les problèmes d'optimisation ne sont pas toujours dans NP, car la preuve de minimalité requiert une quantification universelle ($\forall s \in solutions,\ s \leq s^\star$). Les problèmes d'optimisation trouvent typiquement leur place dans les premiers niveaux de la hiérarchie polynomiale.

Pour finir, notons qu'il existe des problèmes d'Intelligence Artificielle dépassant la hiérarchie polynomiale, et que PSPACE possède notamment des problèmes complets naturels, comme des jeux (GO) ou les problèmes de contraintes quantifiées (QSAT est le problème PSPACE-complet canonique).

3 Un aperçu naïf des logiques à modèles finis

Nous présentons quelques logiques étudiées en complexité descriptive qui nous paraissent intéressantes pour la communauté PC. Les auteurs ne revendiquent aucune autorité en DC, et la présentation se veut introductive et destinée à un public PL/PC.

3.1 Structures de Données

Nous avons vu que les logiques considérées en CD sont des logiques *à modèles finis*. Par conséquent, elles fournissent des structures de données plus simples que les termes habituels en PL. Ces structures sont essentiellement des *tableaux*, même si du point du vue logique des subtilités techniques nécessitent d'encoder l'ensemble (fini, ordonné) d'indices avec la donnée. Celle-ci consiste en un ou plusieurs tableaux, qui peuvent être de dimension 2 ou plus, et prennent leurs valeurs dans un alphabet fini quelconque.

Autant que possible, nous tenterons de donner des noms explicites aux ensembles et de tableaux utilisés pour nos problèmes, et nous utiliserons une notation “à la Pascal”, avec des crochets, pour désigner les éléments de tableaux. Par exemple, un graphe sera codé à travers un ensemble fini de *sommets* (indices), ainsi qu'un tableau booléen à deux dimensions, noté $arc[x, y]$, où x et y prennent leurs valeurs parmi les *sommets*. D'autre données typiques sont les chaînes de caractères, les entiers (codés sur n bits), les tableaux ou matrices d'entiers ainsi codés, les bases de données, ou encore les contraintes n-aires en domaines finis.

3.2 Programmes

Les *quantificateurs de premier ordre* sont bornés à l'un des ensembles d'indices de la structure (*e.g.,* $\forall x \in nodes$). Une expression de premier ordre est construite grâce aux opérateurs logiques classiques ($\wedge, \vee \dots$) et aux quantificateurs de premier ordre. Par exemple, la propriété de consistance d'arc entre deux domaines d_x et d_y reliés par une contrainte binaire c_{xy} s'écrit comme suit :

$$\forall i \in d_x,\ \exists j \in d_y,\ c_{xy}[i, j] \tag{2}$$

Il est facile de se convaincre que les propriétés de premier ordre peuvent être vérifiées en temps polynomial – une solution naïve consiste à simuler les quantificateurs par des boucles “pour tout” imbriquées. En fait, les propriétés de premier ordre représentent une classe de complexité assez pauvre, qui est prouvée strictement incluse dans P (il s'agit d'ailleurs là d'un des rares résultats de séparation entre des classes de complexité qui n'en soit pas au seul stade de la conjecture), et qui est même incluse dans la classe NC des problèmes fortement parallélisables.

Les *quantificateurs de second ordre* introduisent de nouvelles relations, que nous pouvons considérer comme des sous-ensembles du jeu d'indices (ou de manière équivalente comme des tableaux de booléens indexés par ces indices, *e.g.,* $\exists r \subseteq nodes$). Par exemple, une expression naïve du filtrage obtenu par consistance d'arc est la suivante :

$$\exists c'_{xy} \subseteq c_{xy},\ \forall i \in d_x,\ \exists j \in d_y,\ c'_{xy}[i, j] \tag{3}$$

La notion de *vérité* dans de telles logiques possède une signification évidente. De manière formelle, la structure $\mathcal{A}$ sur laquelle on teste la propriété exprimée par le programme $\mathcal{P}$ est vue comme un *modèle* de ce programme, ce que l'on note $\mathcal{A} \models \mathcal{P}$.

3.3 La logique existentielle de second ordre

Fagin [15] a identifié un fragment de logique de second ordre appelé *logique existentielle de second-ordre* (SO$\exists$) ; les phrases de cette logique sont de forme

$$\exists p_1 \subseteq S_1, \dots, \exists p_n \subseteq S_n,\ \Phi$$

où Φ est de premier ordre. Un exemple de définition dans cette logique est donné par le filtrage d'arc-consistance ci-dessus. SO$\exists$ possède deux propriétés intéressantes :

Proposition 1. *Le problème de déterminer si un programme de SO$\exists$ est vrai pour une certaine donnée ($\mathcal{A} \models \mathcal{P}$) est dans NP.*

Démonstration. Pour prouver $\exists r, \Phi$, un algorithme non-déterministe peut exhiber une relation candidate r. Il peut être vérifié en temps polynomial si la condition de premier ordre Φ est vraie pour cette relation. □

Le fait qu'une logique de second-ordre soit non seulement décidable mais également d'une complexité réaliste ne doit pas surprendre le lecteur ; la finitude des structures est bien évidemment à mettre en cause. Fagin [15] a complété le résultat précédent en montrant que SO$\exists$ est une caractérisation syntaxique de NP :

Proposition 2. [Théorème de Fagin] *Tout problème de NP peut s'exprimer comme un programme logique $\mathcal{P}$ écrit en logique existentielle du second-ordre*[3].

Démonstration. (suggestion, *cf.* [15, 33, 21]). Techniquement, inspirée de la preuve du théorème de Cook. Intuitivement, à toute machine de calcul non-déterministe bornée polynomialement en temps il est possible d'associer un énoncé de SO$\exists$ signifiant "il existe un calcul acceptant de cette machine" (cette vision de la preuve est celle donnée par Immerman [21]). □

En d'autres termes : les problèmes exprimés en SO$\exists$ correspondent exactement aux problèmes de satisfaction de contraintes. Ce théorème justifie l'usage de SO$\exists$ comme langage de contraintes d'un point de vue théorique ; son intérêt pratique sera discuté dans la section suivante.

3.4 Logique de second-ordre

Le théorème de Fagin se généralise naturellement à la hiérarchie polynomiale. La logique universelle de second-ordre (SO$\forall$) correspond – sans surprise – à co-NP. Prise sans restriction, la logique du second-ordre correspond à la hiérarchie polynomiale.

4 La logique de Fagin comme langage de programmation logique par contraintes

Dans cette section, nous illustrons brièvement l'emploi de la logique SO$\exists$ comme outil de programmation. Nous concluons sur une comparaison avec des langages existants.

4.1 Exprimer des problèmes en SO$\exists$

Les langages logiques sont souvent trop simples pour être utilisés directement, et des extensions syntaxiques sont requises pour améliorer leur utilisation. Le langage logique est alors vu comme un *noyau* permettant de simplifier la conception du langage. Cette distinction prend un sens particulier dans le cas de SO$\exists$, puisque toute extension du langage est suffisamment riche pour exprimer NP. Naturellement, notre intérêt se portera sur les extensions d'expressivité raisonnable, dont le problème de décision reste dans NP.

[3] La distinction avec un résultat classique de NP-complétude est subtile. Les problèmes NP-complets, tels SAT ou la programmation linéaire en nombres entiers, peuvent servir à modéliser tous les autres problèmes NP au prix d'une réduction polynomiale qui peut être extrêmement peu naturelle (elle n'est en particulier pas nécessairement linéaire). D'après le théorème de Fagin, non seulement tous les problèmes NP peuvent-ils être réduits à un problème SO$\exists$, mais encore cette réduction est-elle *elle-même expressible dans ce langage.*

Un outil utile pour représenter les solutions de problèmes combinatoires est d'utiliser des applications (*mappings*) ; par exemple, une solution à un problème SAT est une instanciation qui associe à chaque variable un booléen ; un chemin hamiltonien est une permutation des sommets du graphe. Les applications s'expriment directement en SO$\exists$ en introduisant un symbole de relation pour chaque résultat possible, et en exprimant le fait que chaque élément possède une et une seule image. Par exemple, la phrase $\exists color : nodes \rightarrow \{red, blue, yellow\}$ peut être réécrite ainsi :

$$\begin{array}{l} \exists red \subseteq nodes,\ \exists yel \subseteq nodes,\ \exists blue \subseteq nodes, \\ \quad \forall x \in nodes,\ (red[x] \vee yel[x] \vee blue[x]) \wedge \\ \qquad \neg(red[x] \wedge yel[x]) \wedge \neg(yel[x] \wedge blue[x]) \wedge \neg(blue[x] \wedge red[x]) \end{array} \tag{4}$$

Une notion d'application plus souple est parfois requise, et nous admettrons toute sorte d'application construite sur un produit cartésien d'ensembles finis.

Les quantificateurs de premier ordre sont typiquement utilisés comme boucles sur les structures de données, comme illustré par la contrainte globale suivante ("alldiff") :

$$\forall x \in var,\ \forall y \in var,\ x \neq y \Rightarrow val[x] \neq val[y] \tag{5}$$

L'addition et la multiplication sont expressibles en logique du premier ordre (sous certaines conditions discutées dans [21]), mais il est également important que le langage fournisse des primitives arithmétiques. Notez que des sommes itérées de forme $\sum_i tab[i]$ ne posent pas de problème, puisqu'elles restent dans NP. Voici un problème-jouet (SEND + MORE = MONEY) qui illustre ces constructions du langage :

$$\begin{array}{l} find\ val : var \rightarrow [0..9], \\ \quad 1000val[s] + 100val[e] + 10val[n] + val[d] + \\ \quad 1000val[m] + 100val[o] + 10val[r] + val[e] = \\ \quad 10000val[m] + 1000val[o] + 100val[n] + 10val[e] + val[y] \\ \quad \wedge val[s] \neq 0 \wedge val[m] \neq 0 \wedge \\ \quad \forall x \in var,\ \forall y \in var,\ x \neq y \Rightarrow val[x] \neq val[y] \end{array} \tag{6}$$

Commer dernier exemple, ce modèle concis du problèmes des n reines sur un échiquier est tiré de [37] :

$$\begin{array}{l} find\ queen : 1..n \rightarrow [1..n], \\ \quad \forall i \in 1..n, \\ \qquad \forall j \in i+1..n, \\ \qquad\quad (queen[i] \neq queen[j]) \wedge \\ \qquad\quad (queen[i] + i \neq queen[j] + j) \wedge \\ \qquad\quad (queen[i] - i \neq queen[j] - j) \end{array} \tag{7}$$

Nous avons donné une idée de l'utilisation de SO$\exists$ par quelques problèmes académiques particulièrement simples. D'autres arguments en la faveur de son utilisation pratique seront donnés en section 4.2.

4.2 Une comparaison...

Il est intéressant de préciser la position du langage SO$\exists$ grâce à une comparaison avec les langages existants.

4.2.1 ... avec Prolog

La Programmation en Logique par Contraintes est considérée unanimement comme un grand succès, à la fois théorique et pratique ; elle combine les avantages d'une sématique claire et d'une utilité pratique pour des applications industrielles. Cependant, Prolog n'est plus considéré comme le langage-hôte universel de Programmation par contraintes. Le fait que des librairires C++ ou Java [34] soient plus faciles à intégrer dans un développement logiciel est en général l'explication avancée pour ce phénomène, mais ceci n'explique pourtant pas la récente préférence pour d'autres langages [3, 36], impératifs ou à base de programmation mathématique. Une autre explication possible réside dans le décalage entre le but original de la programmation logique et son utilisation réelle. Alors qu'il se voulait initialement un moyen de programmation *déclarative*, il est bien connu que Prolog est utilisé en PLC pour poser les contraintes de manière essentiellement *procédurale*. Un exemple simple est la définition originale du *alldiff* de [35], peu déclarative :

```
alldifferent([]).                 outof(X, []).
alldifferent([X|Y]) :-            outof(X, [F|T]) :-
  out_of(X,Y),                      X ≠ Y,
  alldifferent(Y).                  outof(X, T).
```

Si Prolog est un outil de programmation général permettant un travail de conception d'algorithmes de résolution ou de stratégies de recherche, il est clair qu'il ne constitue pas la bonne logique pour *modéliser* les problèmes de façon déclarative. Nous espérons avoir convaincu le lecteur que les logiques à modèles finis, et SO∃ en particulier, atteignent en grande partie l'objectif initial de déclarativité de la programmation logique.

4.2.2 ... avec les langages de modélisation

Pour faciliter la modélisation des problèmes, différents auteurs ont tenté récemment de proposer des langages simples d'utilisation, permettant aux utilisateurs d'exprimer leurs problèmes d'une manière la plus indépendante possible de l'algorithme de résolution utilisé. De tels langages sont appelés *langages de modélisation*[4]. On peut classer les langages OPL [36] et NCL [39] dans cette catégorie, mais également ALICE [26] ou un langage de programmation mathématique comme AMPL [17]. Voici comment Michel et Van Hentenryck [30] résument les mérites des langages de modélisation :

> Modeling languages provide high-level algebraic and set notations to express, in concise ways, mathematical problems that can be solved using state-of-the-art solvers. These languages do not require specific programming skills. Constraint programming languages [...] make it possible to program search procedures to specify how to explore the search space. However, these languages are mostly aimed at computer scientists.

Il est clair que SO∃ entre dans cette catégorie de langages. Les lecteurs familiers avec les langages de modélisation auront notamment remarqué de fortes similarités avec ALICE [26]. Une requête ALICE est typiquement définie comme une application quantifiée existentiellement devant satisfaire des contraintes exprimées avec d'éventuels quantificateurs de premier ordre – ceci correspond exactement à une formule SO∃.

[4]Une autre catégorie de langages, dont l'usage est cependant réservé à des spécialistes, a pour but de faciliter la programmation de stratégies de résolution, voire par exemple [25, 31, 29]

SO∃ présente de plus une séparation nette entre le *modèle* du problème (*e.g.,* la description logique d'un problème de flôts) et les *données* considérées (*e.g.,* des graphes). Cette caractéristique indispensable pour une bonne réutilisabilité des modèles le rapproche de langages comme OPL [36] ou AMPL [17].

5 Le compromis entre traitabilité et expressivité

Alors que la programmation par contraintes est généralement consacrée aux problèmes de satisfaction de contraintes ou d'optimisation, la complexité descriptive considère une grande variété de langages, depuis les plus pauvres jusqu'aux plus expressifs.

5.1 Traitabilité

Il existe plusieurs fragments logiques capturant la classe de complexité P – celle des problèmes "traitables". Nous les présentons brièvement et discutons leur intérêt pour la PL et la PC.

5.1.1 Premier Ordre + Points fixes

Puisque le non-déterminisme correspond à une quantification existentielle de second ordre, il est tentant de penser que le premier ordre correspond au temps polynomial – l'intuition étant que les quantificateurs de premier ordre correspondent à des boucles imbriquées. Il n'en est rien, et il est en fait démontré que le problème d'*atteignabilité* (le problème d'existence d'un chemin entre deux sommets d'un graphe) ne peut être exprimé par une formule de premier ordre. Remarquez que ce problème est complet pour LOGSPACE, et qu'il n'est donc même pas parmi les problèmes les plus difficiles de P.

Ce qui manque à la logique de premier ordre pour pouvoir exprimer l'atteignabilité est la possibilité de faire des définitions *inductives.* Considérons la séquence de définitions suivantes :

$$\begin{array}{lcl} path_0[x,y] & \equiv & x = y \\ path_1[x,y] & \equiv & path_0[x,y] \vee \ (\exists z,\ edge[x,z] \wedge path_0[z,y]) \\ path_2[x,y] & \equiv & path_1[x,y] \vee \ (\exists z,\ edge[x,z] \wedge path_1[z,y]) \\ \ldots & & \end{array} \tag{8}$$

Chaque prédicat $path_i$ exprime l'existence d'un chemin de longueur au plus i entre deux sommets. Pour généraliser cette définition à des chemins de longueur arbitraire, il nous faut une définition inductive spécifiée par une fonction qui, prenant une relation r en entrée, définit une nouvelle relation :

$$\phi(r) \equiv \{\langle x,y\rangle \mid\ r[x,y] \vee \exists z,\ edge[x,z] \wedge r[x,y]\} \tag{9}$$

Bien sûr, on retrouve alors $path_i \equiv \phi^i(Id)$, où Id est la relation d'indentité sur les sommets. Une propriété importante de la formule $r[x,y] \vee \exists z,\ edge[x,z] \wedge r[x,y]$ est que r présente des occurrences *positives*, *c-à-d.* qu'il apparait toujours sous un nombre pair de négations. De manière plus générale, toute formule positive définit un opérateur monotone dont le point-fixe peut être effectivement calculé.

L'extension de la logique de premier ordre avec un opérateur de point fixe est notée FO(LFP). Notez que les relations ne sont pas restreintes à être binaires, et que de telles définitions sont donc plus générales que des clôtures transitives. Les opérateurs de point-fixe sont bien connus en programmation logique, mais la finitude des modèles fait gagner la propriété suivante :

Proposition 3. *FO(LFP) correspond à P, plus précisément :*

1. *Le problème de décider si une définition FO(LFP) est vérifiée par une structure finie est dans P ;*
2. *Tout problème de P peut s'exprimer comme un programme logique écrit en FO(FLP).*

Démonstration. (1) Adaptation du théorème de Knaster-Tarski aux modèles finis. Le point-fixe peut être construit par un algorithme d' "itérations chaotiques" similaire à ceux utilisés en propagation de contraintes ou en analyse de programmes [1].

(2) Il existe un problème complet pour P *par des réductions de premier ordre* qui peut être exprimé en FO(LFP) [21]. □

5.1.2 Logique existentielle de Horn du second-ordre

Les langages utilisant un opérateur de point-fixe explicite sont "notoirement incompréhensibles" [6] ; malgré ses belles propriétés, il n'est donc pas réaliste de considérer FO(LFP) comme un langage *naturel* pour exprimer les problèmes de P. En fait, la proposition 3 est aussi vérifiée par le langage Datalog, *c-à-d.* la restriction de la logique existentielle de second ordre à des clauses de Horn (une précaution est nécessaire *cf.* [19, 20]). Par exemple, le problème d'atteignabilité s'exprime évidemment de manière naturelle en Datalog :

$$\begin{array}{l} path[x,y] \leftarrow x = y \\ path[x,y] \leftarrow edge[x,z] \wedge path[z,y] \end{array} \tag{10}$$

5.1.3 Intérêt pour la communauté PLC

La faculté des langages discutés précédemment de définir des modèles exécutables en temps polynomial est séduisante. Des perspectives d'application concernent, par exemple, des outils génériques de définition de contraintes globales [4].

En programmation par contraintes, une série d'articles a été consacrée à déterminer des jeux de contraintes primitives restreints permettant de garantir la traitabilité des problèmes [23, 24]. Une telle préoccupation relève évidemment de la complexité descriptive, bien que les auteurs ne fassent pas référence de manière explicite à la littérature de ce domaine. En particulier, certains résultats concernant l'utilisation de points fixes semblent proches de ceux concernant FO(LFP).

5.2 Plus d'expressivité

La théorie des modèles finis fournit des langages dont l'expressivité dépasse NP. Le langage obtenu en ajoutant à la logique de second ordre un *opérateur de clôture transitive* correspond à PSPACE. Peu de langages de programmation par contraintes permettent d'exprimer ces problèmes (l'un de ces langages est Alma [3]). Au niveau algorithmique, il est vrai que la résolution des problèmes PSPACE-complets demeure moins bien comprise que celle des problèmes de NP.

L'expression de problèmes d'optimisation ne requiert pas toute l'expressivité du second-ordre ; l'utilisation d'une couche d'alternation $(\exists, ..., \forall)$ est suffisante.

6 Conclusion

L'une des contributions les plus remarquables de la logique à la programmation a assurément été apportée par la théorie des bases de données, notamment avec les travaux de Codd [9]. Alors que les langages de requêtes sur les bases de données utilisent une logique à modèles finis, la plupart des autres langages de programmation logique ont choisi d'utiliser des constructions inductives comme les termes de Prolog. La complexité descriptive montre que les connexions entre les logiques à modèles finis et le calcul ne sont pas épuisées. Les langages de programmation en logique à modèles finis présentent des caractéristiques remarquables : la *terminaison* des programmes est garantie, et la complexité des requêtes dépend de critères purement *syntaxiques*. Les questions d'*expressivité* des langages peuvent également être exprimées formellement, puisque ces langages capturent des classes de complexité.

Puisque les problèmes combinatoires traitent en général de structures finies (telles les graphes) identiques à celles étudiées en théorie des modèles finis, la programmation logique à modèles finis trouve un champ d'applications potentielles comme langage de programmation par contraintes. Nous avons proposé un aperçu naïf des résultats de complexité descriptive d'intérêt pour la communauté PLC. Notre présentation doit beaucoup à Immerman [20], et nous espérons que notre approche facilitera la diffusion de ces résultats dans la communauté PLC. Pour résumer, notre argument principal était que la logique existentielle de second ordre est le langage logique de choix pour *modéliser* les problèmes combinatoires, alors que Prolog est un outil de programmation plus général, servant à des tâches moins "déclaratives". Nous avons également montré que SO$\exists$ peut être vu retrospectivement comme le noyau logique de différents langages de modélisation existants.

Les connexions entre logique et programmation mises en avant dans la littérature sont d'une diversité foisonnante, particulièrement dans les domaines du *model-checking* [6, 10] et (plus généralement) de la vérification de programmes [5]. L'un des résultats de la complexité descriptive est qu'il n'existe pas seulement une hiérarchie de classes de problèmes mais également une hiérarchie de fragments de logique leur correspondant. Dans cet article, nous avons discuté la position des langages de programmation logique par contraintes dans cette hiérarchie ; nous espérons avoir rendu compte des perspectives offertes par la complexité descriptive comme moyen de classification de toutes ces approches logiques de la programmation.

Remerciements

Cet article a bénéficié de discussions avec Gérard Ferrand et de remarques pertinentes des relecteurs, notamment Antoine Rauzy.

Références

[1] K. R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2), 1999.

[2] K. R. Apt and Bezem M. Formulas as programs. In K. R. Apt, Marek V., Truszczynski M., and D. S. Warren, editors, *The Logic Programming Paradigm : A 25 Years Perspective*, pages 75–107. Springer-Verlag, 1999.

[3] K. R. Apt and A. Schaerf. The Alma project, or how first-order logic can help us in imperative programming. In *Correct System Design*, pages 89–113. Springer, 1999.

[4] N. Beldiceanu. Global constraints as graph properties on a structured network of elementary constraints of the same type. In *Proc. of the 6th Int. Conf.on Constraint Programming*, pages 52–66, Singapore, 2000. LNCS.

[5] R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, 1979.

[6] J. C. Bradfield and C. Stirling. Modal logics and mu-calculi. In Bergstra, Ponse, and Smolka, editors, *Handbook of Process Algebra*, pages 293–330. Elsevier, 2001.

[7] M. Cadoli, I. Giovambattista, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC : an executable specification language for solving all problems in NP. *Computer Languages*, 26(2-4) :165–195, 2000.

[8] M. Cadoli and L. Palopoli. Circumscribing DATALOG : Expressive power and complexity. *Theoretical Computer Science*, 193(1-2) :215–244, 1998.

[9] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6) :377–387, 1970.

[10] M.-M. Corsini and A. Rauzy. Toupie : The mu-calculus over finite domains as a constraint language. *J. of Automated Reasoning*, 19(2) :143–171, 1997.

[11] D. Diaz and P. Codognet. Design and implementation of the GNU-Prolog system. *Journal of Functional and Logic Programming*, 6, 2001.

[12] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory.* Springer, 1999.

[13] F. Fages, S. Soliman, and V. Vianu. Expressiveness and complexity of concurrent constraint programming : a finite model theoretic approach. Technical Report 98-14, LIENS, 1998.

[14] R. Fagin. Finite-model theory - a personal perspective. *Theor. Comp. Sc.*, 116(1-2) :3–31, 1993.

[15] R. A. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In R. Karp, editor, *Complexity of Computation*, pages 43–73. SIAM-AMS, 1974.

[16] R. Fikes. REF-ARF : A system for solving problems stated as procedures. *Artificial Intelligence*, 1(1/2) :27–120, 1970.

[17] R. Fourer, D. Gay, and B. Kernighan. *AMPL : A Modeling Language for Mathematical Programming.* Duxbury Press, 1993.

[18] R. Givan and D. A. Mc Allester. New results on local inference relations. In *Proc. of the 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 403–412, Cambridge, MA, 1992.

[19] E. Gradel. Capturing complexity classes by fragments of second-order logic. *Theoretical Computer Science*, 101(1) :35–57, 1992.

[20] N. Immerman. Descriptive complexity : A logician's approach to computation. *Notices of the American Mathematical Society*, 42(10) :1127–1133, 1995.

[21] N. Immerman. *Descriptive Complexity.* Springer, 1999.

[22] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Conf. Record of the 14th Annual ACM Symp. on Principles of Programming Languages*, pages 111–119, Munich, Germany, 1987.

[23] P. Jeavons, D. Cohen, and M. C. Cooper. Constraints, consistency and closure. *Artificial Intelligence*, 101(1-2) :251–265, 1998.

[24] P. Jeavons, D. Cohen, and M. Gyssens. Closure properties of constraints. *Journal of the ACM*, 44(4) :527–548, 1997.

[25] F. Laburthe and Y. Caseau. SALSA : A language for search algorithms. In *Proc. of the 4th Int. Conf. on Constraint Programming*, pages 310–324, Pisa, Italy, 1998. LNCS.

[26] J.-L. Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1) :29–127, 1978.

[27] D. A. Mc Allester. Automatic recognition of tractability in inference relations. *J. of the ACM*, 40(2) :284–303, 1993.

[28] D. A. Mc Allester. Notes for course 6.824 on AI. Technical report, MIT, 1995. chapters on CSPs, Graph search, Games.

[29] L. Michel and P. Van Hentenryck. Localizer. *Constraints*, 5(1/2) :41–82, 2000.

[30] L. Michel and P. Van Hentenryck. Modeler++ : A modeling layer for constraint programming libraries. In *CP-AI-OR Workshop*, Ashford, Kent, UK, 2001.

[31] E. Monfroy. A solver collaboration in BALI. In *Proc. of the Int. Joint Conf. and Symp. on Logic Programming*, pages 349–350, Manchester, UK, 1998. The MIT Press.

[32] A. Newell, J. C. Shaw, and H. A. Simon. Report on a general problem-solving program. In *Proc. of the 1st IFIP World Computer Congress*, pages 256–264, Paris, France, 1960. UNESCO.

[33] C. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

[34] J.-F. Puget. A C++ implementation of CLP. Technical report, ILOG, 1994. ILOG Solver Collected Papers.

[35] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* the MIT Press, 1989.

[36] P. Van Hentenryck. *The OPL Optimization Programming Language.* the MIT press, 1999.

[37] P. Van Hentenryck, L. Perron, and J.-F. Puget. Search and strategies in opl. *ACM Transactions on Computational Logic*, 1(2) :285–320, 2000.

[38] P. Van Hentenryck, V. A. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). *Journal of Logic Programming*, 37(1-3) :139–164, 1998.

[39] J. Zhou. Introduction to the constraint language NCL. *Journal of Logic Programming*, 45(1-3) :71–103, 2000.