MASTER EN SCIENCES INFORMATIQUES

Equivalence algorithmique par transformations de programmes logiques avec contraintes

Yernaux, Gonzague

Award date:
2017

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2016–2017

Équivalence algorithmique par transformations de programmes logiques avec contraintes

Gonzague Yernaux

Maîtres de stage : Étienne Payet et Frédéric Mesnard

Promoteur : Wim Vanhoof

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

L'équivalence algorithmique est un sujet important en analyse de programmes. Elle est utile dans de nombreuses applications, dont la détection de plagiat et la reconnaissance automatique d'algorithmes. Nous proposons une approche nouvelle articulée autour des transformations de programmes logiques avec contraintes. Nous définissons un cadre de travail flexible dans lequel la comparaison de programmes logiques avec contraintes se fait par transformations syntaxiques successives. Nous introduisons des transformations de programmes logiques avec contraintes paramétrables et discutons leur impact sémantique sur les programmes. Nous évaluons la capacité de cette approche à préserver la structure des algorithmes capturés par les programmes, et proposons des extensions simples pour aiguiser le processus de comparaison algorithmique présenté.

Mots-clés : transformations de programmes, équivalence algorithmique, programmation logique avec contraintes

Préface

Ce mémoire est présenté en vue de l'obtention du grade de Master en sciences informatiques à l'Université de Namur. Depuis septembre 2016, le travail de recherche et de rédaction a été réalisé sous la supervision de Monsieur Wim Vanhoof, Professeur à l'Université de Namur. Une partie des recherches a été effectuée lors d'un stage entre septembre et décembre 2016 sous la direction de Messieurs Mesnard et Payet, Professeurs à l'Université de la Réunion.

Mes premiers remerciements vont à mon promoteur Wim Vanhoof pour son implication tout au long du projet. Ses conseils avisés guidés par son impressionnante expérience dans le domaine de la recherche m'ont permis de continuellement avancer dans mon travail. Je lui suis particulièrement reconnaissant pour le temps qu'il m'a consacré pendant tous ces mois. Ni ses nombreuses responsabilités, ni l'éloignement géographique ne l'ont empêché d'être d'une aide colossale dans la mise en place et la structure des résultats présentés dans ce mémoire.

Je tiens également à remercier mes maîtres de stage, Messieurs Mesnard et Payet. Non contents d'accepter la venue d'étudiants belges à l'Université de la Réunion, ils ont placé le stage sous le signe de la confiance et de l'autonomie. C'est donc une réelle expérience de la recherche, et de toutes les difficultés d'organisation auxquelles il faut faire face dans ces circonstances, que j'ai vécue lors de ce stage exceptionnel.

Le voyage jusqu'à la Réunion n'aurait pas été le même sans mes deux camarades de cours Gérôme Laffineur et Axel Devos. Tout au long du processus de recherche, puis de rédaction, nous nous sommes tirés vers le haut dans un esprit de travail commun.

Enfin, je remercie les membres de ma famille qui ont relu le présent mémoire avant sa remise. Je remercie également les membres du personnel de l'Université de Namur et de l'Université de la Réunion pour les infrastructures et les outils de travail mis à notre disposition.

Gonzague Yernaux
Mai 2017

Table des matières

Résumé **i**

Préface **iii**

1 Introduction **1**
1.1 Contexte . . . 1
1.2 Motivations . . . 1
1.3 Applications potentielles . . . 2
1.4 État de l'art . . . 3
1.5 Nouvelles contributions . . . 3
1.6 Objectifs . . . 4

2 Programmation logique avec contraintes **5**
2.1 Domaines et contraintes . . . 5
2.2 Foncteurs et termes imbriqués . . . 6
2.3 Structure des programmes logiques avec contraintes . . . 7
2.4 Sens des programmes logiques avec contraintes . . . 8
2.5 Exécution des programmes logiques avec contraintes . . . 9
2.6 Exemple de programme logique avec contraintes . . . 9

3 Cadre de travail pour l'équivalence de programmes logiques avec contraintes **11**
3.1 Transformations de programmes logiques avec contraintes . . . 11
3.2 Fonction sémantique des programmes CLP . . . 12
3.3 Équivalence et connexion algorithmiques . . . 14
3.4 Points d'entrée . . . 15
3.5 Ordre des clauses et des arguments . . . 17
3.6 Paramétrage et méta-paramétrage des transformations . . . 17
3.6.1 Paramètres . . . 17
3.6.2 Méta-paramètres . . . 17
3.7 Transformations sur le corps des clauses . . . 20
3.7.1 Remplacement de contraintes . . . 20
3.7.2 Agrégation de clauses . . . 22
3.7.3 Changement d'ordre d'appels . . . 23
3.7.4 Remplacement d'arguments . . . 23
3.8 Algorithme naïf de comparaison . . . 24
3.9 Exemples . . . 25
3.9.1 Exemple 1 . . . 25
3.9.2 Exemple 2 . . . 27

4 Transformations de programmes CLP avancées **29**
4.1 Problématiques récurrentes . . . 29
4.1.1 L'explicitation de calculs . . . 30
4.1.2 L'ajout d'intermédiaires . . . 30
4.1.3 Les calculs impossibles . . . 31
4.1.4 L'ajout de variables . . . 31
4.1.5 La duplication de code . . . 32
4.1.6 La réécriture et l'insertion de boucles . . . 32
4.2 Dépliage . . . 33
4.2.1 Dépliages simples et multiples . . . 33
4.2.2 Réponse à l'explicitation de calculs et à l'ajout d'intermédiaires . . . 36

4.2.3 Dépliages successifs . . . 37
4.3 Suppression de clauses inconsistantes . . . 40
4.4 Pliage . . . 42
4.5 Tranchage . . . 43
4.5.1 Tranchage d'appels . . . 44
4.5.2 Tranchage de tautologies . . . 44
4.5.3 Tranchage d'arguments . . . 45
4.5.4 Tranchage en deux passes . . . 47
4.5.5 Connexion de tranches . . . 49
4.6 Gestion des boucles . . . 50
4.6.1 Déroulage de boucles . . . 50
4.6.2 Boucles impératives et boucles logiques . . . 52
4.6.3 Boucles inutiles . . . 54
4.6.4 Syntaxe des boucles . . . 54
4.7 Stratégie typique de transformation de programmes à comparer . . . 55

5 Équivalence algorithmique **59**
5.1 Spécialisation de programmes CLP . . . 59
5.2 Dépliage et complexité . . . 61
5.3 Autres manifestations des algorithmes . . . 63
5.4 Équivalence algorithmique dans la littérature . . . 63
5.4.1 Classes d'équivalences et mathématiques . . . 64
5.4.2 Instance algorithmique . . . 65
5.4.3 Valeurs fondamentales . . . 65
5.4.4 Exécution symbolique et déviations . . . 66
5.4.5 Caractéristiques et concepts algorithmiques . . . 66
5.4.6 Impossibilité de l'équivalence algorithmique . . . 67

6 Conclusions et travaux futurs **69**
6.1 Un cadre de travail nouveau pour comparer des programmes logiques avec contraintes . . . 69
6.2 Des transformations puissantes et paramétrables . . . 70
6.3 Une approche indépendante des sémantiques . . . 70
6.4 Travaux futurs . . . 71
6.4.1 Gestion des boucles . . . 71
6.4.2 Gestion des sous-algorithmes . . . 71
6.4.3 Nouvelles transformations . . . 71
6.4.4 Gestion des contraintes . . . 72
6.4.5 Automatisation . . . 72
6.4.6 Extensions du cadre de travail . . . 72

Bibliographie **73**

Chapitre 1

Introduction

1.1 Contexte

En 2016, l'article [43] décrivait une approche centrée sur les transformations de programmes logiques pour comparer des algorithmes écrits en clauses de Horn. Cette approche devait encore être creusée afin d'aboutir à un cadre de travail pour la comparaison de programmes logiques avec contraintes. C'est l'objectif du présent mémoire.

Nous allons étudier l'équivalence algorithmique avec une approche transformationnelle centrée sur les programmes logiques avec contraintes. Deux algorithmes seront considérés comme équivalents si les programmes qui les capturent respectivement peuvent être transformés en un même programme tiers, à l'aide de transformations exclusivement syntaxiques et préservatrices (d'une partie) du sens des programmes. Nous fournissons pour cela un cadre de travail dans lequel ces transformations peuvent être paramétrées. L'idée est d'obtenir un cadre assez souple pour s'adapter aux différentes applications qui en feraient usage. Certaines transformations de programmes logiques, déjà étudiées en profondeur dans la communauté scientifique et considérées comme relativement puissantes, seront présentées et leur utilité sera illustrée.

1.2 Motivations

La comparaison d'algorithmes est un sujet controversé dans la communauté scientifique ([43]). Différentes approches ont déjà été creusées dans la littérature. Cet intérêt porté au sujet nous motive à l'étudier plus en profondeur. Ainsi ce mémoire essaiera-t-il d'apporter des nouvelles lumières à des questions encore ouvertes telles que « Est-il possible de définir l'équivalence algorithmique de façon univoque ? » ou encore « À quel moment deux programmes calculent-ils réellement la même chose ? ».

De plus, l'article [43] décrit comment il est possible de traduire du *bytecode* Dalvik en clauses de Horn. C'est dans le cadre de ce projet de recherche que d'autres étudiants ont réalisé le même travail pour d'autres formalismes de langage assembleur ([12] et [20]). Les clauses de Horn sont donc dérivables, automatiquement, de langages de haut niveau tels que le Java, puisqu'il suffit d'en décompiler le *bytecode* à l'aide de l'outil décrit dans [20]. De même à partir d'exécutables décompilés en langage assembleur x86 grâce à l'outil décrit dans [12].

Le problème de déterminer si deux programmes calculent la même fonction est indécidable en général, parce que le problème de l'arrêt est indécidable ([27]). Nous adoptons ici une approche alternative, orientée sur la syntaxe des programmes manipulés. Cette approche utilise une méthode transformationnelle, qui aura pour but de créer des séquences de programmes transformés à partir des programmes à comparer. Nous essayerons, de façon heuristique, de faire coïncider les programmes ainsi générés pour conclure à une équivalence sémantique entre les programmes initiaux.

Les clauses de Horn nous semblent présenter un formalisme puissant, relativement universel et dans lequel il est aisé d'effectuer des transformations de programmes. Intuitivement, un programme logique sera plus facile à manipuler dans ce but qu'un programme impératif ou orienté objet, grâce à sa structure linéaire et répétitive, sans structures imbriquées complexes.

1.3 Applications potentielles

Le fait de disposer d'une définition (flexible) de l'équivalence algorithmique, voire d'un algorithme de comparaison d'algorithmes fondé sur une telle définition, peut avoir diverses applications.

De deux programmes, pourtant différents à première vue dans leur structure et leur contenu (code), l'équivalence algorithmique permet d'extraire un même « cœur algorithmique ». Une des applications évidentes de ce processus est la **détection de plagiat**. Comme mentionné dans [47], le plagiat algorithmique est plus complexe que le plagiat logiciel puisqu'un même algorithme peut être développé différemment par différents développeurs, de différentes façons et dans différents langages de programmation. Il ne s'agit pas seulement de détecter des portions de codes identiques (ou clones), mais bien de comparer l'essence de deux programmes ([43]). Comme expliqué ci-dessus, nous nous concentrerons sur la comparaison de programmes écrits en clauses de Horn, un formalisme assez universel dans le sens où il nous semble que de nombreux langages impératifs, orientés objet, mais aussi assembleurs, peuvent aisément être traduits en clauses de Horn.

Au fil de son cycle de vie, un programme a tendance à évoluer et il peut être intéressant d'en comparer les différentes versions. Cela permet notamment de s'assurer que l'algorithme capturé dans le programme est bien inchangé malgré certaines modifications dans le code. Il s'agit alors d'une étude de régression. Cet aspect est particulièrement utile en **déboguage** et pour vérifier la conformité d'un programme par rapport à des spécifications données ([2]).

Au-delà de la comparaison d'algorithmes, nous pourrions parler de **reconnaissance** d'algorithmes, un sujet qui est l'objet de nombreux travaux existants. On parle de reconnaissance d'algorithmes lorsqu'il s'agit de vérifier si un programme implémente un algorithme donné ([45]). Par exemple, si un programme est supposé capturer un certain algorithme $\mathcal{A}$ mais que cet algorithme n'est pas reconnu dans le programme après application de la reconnaissance d'algorithmes, c'est qu'il peut y avoir une erreur dans le programme examiné ([2]). Dans la même veine, la reconnaissance d'algorithmes dans le code d'un programme participe à la **compréhension** de ce programme. Cela peut alors servir de base pour de la rétro-ingénierie ([2]). Il est aussi possible d'optimiser un programme en remplaçant son code capturant un algorithme connu par une autre implémentation, plus efficace, du même algorithme ([27]).

Comme décrit dans [11], nous pourrions également appliquer une comparaison algorithmique à la vérification de propriétés des programmes. En particulier, l'élégance des clauses de Horn est un atout pour l'expression des contraintes que des programmes, même impératifs, doivent vérifier.

Notre approche se focalisant sur les transformations de programmes logiques, ces transformations ont été implémentées dans un outil Prolog et peuvent être utilisées à d'autres fins que la seule comparaison d'algorithmes. À nouveau, il existe de nombreuses utilisations possibles de ces transformations.

D'abord, comme nous le verrons dans les chapitres suivants, la transformation de dépliage, pour un programme logique avec contraintes, n'est jamais qu'un pas dans l'exécution de ce programme. Le simple fait de disposer de cette transformation peut donc permettre d'exécuter un programme logique étape par étape et ainsi d'en favoriser le déboguage. En outre, le dépliage est un outil simple et élégant pour spécialiser des programmes logiques, comme nous le verrons dans un chapitre ultérieur.

Ensuite, le tranchage est une transformation qui extrait d'un programme une « tranche » de ce programme. Cette transformation, également utilisée dans d'autres paradigmes de programmation, permet notamment d'extraire d'un programme uniquement le code correspondant à l'évolution d'une certaine variable d'intérêt, par exemple pour effectuer des tests ou pour déboguer ([33]).

Par ailleurs, nous exposons des transformations typiques au paradigme de la programmation logique avec contraintes, telles que l'agrégation de clauses et le remplacement de contraintes. Ces transformations ont des utilités directes dans l'étude et l'optimisation de programmes logiques avec contraintes.

Dans [3], sont citées comme applications concrètes d'un tel arsenal de transformations l'amélioration de l'efficacité de programmes par des transformations successives et la dérivation automatique de programmes à partir de spécifications.

1.4 État de l'art

Dans ce mémoire, nous abordons la comparaison d'algorithmes. Ce thème a déjà éveillé une certaine curiosité scientifique. La question « Quand deux algorithmes sont-ils les mêmes ? » est le titre et l'objet principal de l'article [6], qui questionne la faisabilité de définir la notion d'algorithme. Dans [2], la reconnaissance d'algorithmes est amenée à l'aide d'équations de récurrence sur un programme impératif initial que l'on essaye de faire correspondre à un bout de code donné. Une autre approche est celle de l'article [41] dont le but est d'améliorer la correction automatique de travaux de programmation. Il y est question d'examiner certaines caractéristiques quantitatives et descriptives des programmes soumis par les étudiants pour y déceler un algorithme précis (par exemple, le tri à bulles). L'algorithme en question est reconnu grâce à un arbre de décision dont les arcs sont étiquetés par les caractéristiques observables (par exemple, un arbre permettant de distinguer les différents algorithmes de tri). Quant à l'ouvrage [27], il cherche une façon automatique de remplacer des algorithmes, soit par une implémentation plus efficace, soit par le même algorithme en un autre langage. Cela peut par exemple servir à faciliter le changement de technologie d'un système d'information en automatisant une partie du processus.

La plupart des sources mentionnées ci-dessus s'intéressent à la reconnaissance algorithmique dans le cas de programmes impératifs. Nous nous concentrons dans le présent mémoire sur l'équivalence d'algorithmes capturés dans des *programmes logiques avec contraintes*. Dans [24], l'équivalence de programmes logiques est abordée dans un sens très général et théorique. On y envisage la comparaison de programmes sous les angles des sémantiques logique, opérationnelle et fonctionnelle. Dans [9], les clones sémantiques dans les programmes logiques, et leur détection, sont étudiés. Notre approche, en contraste, utilise des transformations de programmes pour définir une notion d'équivalence algorithmique, paramétrée par un ensemble de transformations et une fonction sémantique. Si la transformation d'un programme en un autre programme (alors équivalent) a déjà été étudiée, il n'y a pas eu à notre connaissance de recherche sur la transformation parallèle de deux programmes dans le but d'arriver à un même programme tiers, sauf dans [43].

Les références ci-dessus ne sont pas exhaustives, mais donnent une idée des différentes théories étudiées dans le secteur de la comparaison algorithmique au fil des dernières décennies. Dans un chapitre ultérieur, nous examinons certaines de ces recherches plus en détail dans le but de cerner au mieux leurs définitions d'algorithme et, par là même, d'équivalence algorithmique. Nous verrons que le sujet est relativement controversé.

Les transformations de programmes logiques en elles-mêmes ont également été abordées dans de nombreux articles. Le tranchage est notamment étudié en détail dans [33], [23], [38] et [44]. De très nombreux auteurs explicitent ou font usage du pliage et du dépliage, deux autres transformations très puissantes, notamment pour construire des stratégies de transformations. Nous citerons [29], [10] ou encore [11]. Ce dernier article construit une technique à l'aide du dépliage et du pliage, appelée appariement de prédicats, dans le but de vérifier des propriétés de programmes. Nous citerons encore [40], un article où tranchage et dépliage sont combinés dans une approche originale.

En 1996, dans [29], des transformations sont présentées afin d'améliorer, par leur application successive, un programme logique initial potentiellement inefficace. En 1997, l'article [3] donne des transformations applicables sur des programmes logiques avec contraintes tout en examinant la sémantique de ces programmes qui doit être conservée au fil des transformations.

Puis, en 2013, dans [10], l'idée est de transformer des programmes logiques avec contraintes pour vérifier des propriétés de programmes impératifs. En 2016, les mêmes auteurs proposent dans [11] une formalisation cette fois centrée sur les clauses de Horn contraintes, toujours dans le but de vérifier des propriétés de programmes impératifs. D'autres articles se sont penchés sur la question au fil des années, mais force est de constater qu'il y eut un élan important dans le domaine des transformations de programmes logiques dans les années 1990. Un nouveau souffle semble avoir été donné dans ce domaine dans les années 2010, notamment avec l'apparition de termes tels que « clauses de Horn contraintes ».

Remarquons que dans les deux derniers articles cités ci-dessus, l'on retrouve une phase de traduction de programme depuis un langage impératif vers un formalisme logique. C'est également ce qui est élaboré dans [43] en 2016 et dans [12] et [20] en 2017 ; trois documents qui se situent dans le même projet de recherche que le présent mémoire et qui en donnent le point de départ.

1.5 Nouvelles contributions

La nouveauté présentée dans ce travail tient aux points suivants :

— l'utilisation de transformations de programmes logiques avec contraintes dans le but de comparer le cœur algorithmique de deux programmes ;
— la déclinaison flexible des transformations de programmes logiques avec contraintes en différentes formes, chacune avec son degré de sensibilité ;
— la définition d'un cadre de travail capable de s'assouplir en fonction des différentes sémantiques accordées aux programmes logiques avec contraintes ;
— la formalisation d'un algorithme général, et d'une stratégie heuristique bâtie sur cet algorithme, capables de comparer deux programmes logiques avec contraintes.

De plus, un prototype de comparateur d'algorithmes a été implémenté en Prolog. Ce prototype prend en entrée des clauses de Horn contraintes et cherche à construire des suites de transformations de programmes applicables à ces programmes, dans le but de transformer les deux programmes en entrée en un même programme tiers.

1.6 Objectifs

Pour comparer des algorithmes, nous allons étudier les programmes qui les incarnent. Avant cela, il s'agit de représenter ces programmes en un seul et même formalisme. Dans le cadre de ce mémoire, le formalisme choisi est celui des clauses de Horn contraintes. Nous allons donc considérer des programmes écrits en programmation logique avec contraintes (abrégé CLP dans la suite). Dans un premier temps, nous définirons ce paradigme en nous appuyant sur les fondements de la programmation logique. Ensuite, nous définirons un cadre de travail pour la comparaison algorithmique à l'aide de transformations de programmes CLP. Nous introduirons des transformations élémentaires envisagées dans ce cadre. Puis nous présenterons quelques transformations de programmes CLP plus puissantes et discuterons leur utilité pratique à l'aide d'exemples. Enfin, nous étayerons quelques réflexions que soulève cette approche transformationnelle en nous recentrant sur le sens de l'équivalence algorithmique, avant de conclure et de donner des pistes de travaux futurs.

Chapitre 2

Programmation logique avec contraintes

Les notions explicitées dans ce chapitre posent les fondations du paradigme de programmation logique avec contraintes. Les termes employés pour décrire la programmation logique sont ceux du langage de programmation Prolog. La façon d'introduire les concepts est une synthèse de [42], [7], [25] et [17]. Nous ne développerons que les notions qui nous seront utiles dans le cadre de ce mémoire.

Les notions développées présupposent quelques connaissances basiques en programmation logique. Les sources mentionnées ci-dessus comportent un complément d'information conseillé pour les lecteurs ne connaissant pas les bases de la programmation logique.

2.1 Domaines et contraintes

Avant de parler des programmes logiques en tant que tels, il convient d'aborder la notion de **contraintes**. L'idée des contraintes est de capturer les opérations du programme logique ayant rapport avec un certain **domaine de calcul**. L'on pourrait ainsi imaginer de se limiter aux nombres entiers. Le domaine de calcul est alors équivalent à $\mathbb{Z}$.

Une fois le domaine de calcul choisi, il est possible d'exprimer des contraintes qui correspondent à des relations parmi les valeurs appartenant au domaine de calcul et qui s'expriment grâce à l'utilisation de **symboles fonctionnels** et de **symboles prédicatifs**.

Pour formaliser, nous allons suivre la démarche de [17] qui nous semble assez générale pour rendre nos réflexions prochaines sur CLP également générales.

Définition 2.1 (Domaine de contraintes)**.** *Soit un ensemble appelé domaine de calcul et noté $\mathcal{D}$, défini sur un alphabet Σ. Soit l'ensemble de contraintes, noté $\mathcal{C}$, une classe de formules formées par les symboles de Σ. Le domaine de contraintes est la paire $(\mathcal{D}, \mathcal{C})$. Par abus de notation, nous nous contenterons d'écrire $\mathcal{D}$ pour désigner le domaine de contraintes.*

Usuellement, pour indiquer que l'on a affaire à du CLP par rapport à un domaine particulier, on note $CLP(Dom)$ où Dom correspond au domaine choisi (par exemple $\mathbb{Z}$ ou $\{1\} \cup \{5\}$). Ainsi lorsque l'on écrit $CLP(\mathbb{Z})$, nous envisageons un cas où $\mathcal{D} = \mathbb{Z}$ et $\mathcal{C}$ est l'ensemble des contraintes que l'on peut écrire sur les nombres entiers avec les symboles de Σ. Dans le cas où l'on parle du paradigme en général par rapport à un domaine quelconque, on écrit $CLP(X)$ ([17]).

L'alphabet Σ comprend d'une part les constantes du domaine. D'autre part, il englobe les symboles fonctionnels et les symboles prédicatifs du domaine. Les symboles fonctionnels seront des foncteurs d'**opérations**. Ainsi, le symbole fonctionnel $+$ est un foncteur qui représente traditionnellement l'opération d'addition pour la plupart des domaines arithmétiques usuels. Quant aux symboles prédicatifs, ils représentent des prédicats, et donc des **relations**. Il peut ainsi s'agir des symboles $=, \neq, \leq$ *ou* $\geq$ dans l'arithmétique traditionnelle.

Par simplicité, et pour suivre la démarche utilisée dans [17], nous partirons du principe que $\mathcal{C}$ contient des termes issus d'un langage de logique de premier ordre. Il sera donc possible d'utiliser des quantificateurs $(\exists, \forall)$ et des connecteurs logiques $(\vee, \wedge, \neg, \Rightarrow, \Leftrightarrow)$.

Dans la même idée, des raccourcis seront tolérés dans la suite pour peu qu'ils soient évidents au regard du domaine considéré. Par exemple, même si -4 ne fait pas partie d'un alphabet Σ qui ne comprendrait que 0 et 1 comme constantes, ce sera une notation utilisée fréquemment comme raccourci pour l'expression $0-1-1-1-1$, où nous faisons appel au symbole fonctionnel binaire $-$ et aux symboles constants 0 et 1. Nous partirons également du principe que Σ comprendra toujours un nombre infini de symboles pouvant servir pour représenter des **variables**.

Notons qu'afin d'éviter toute confusion, les symboles prédicatifs, les symboles fonctionnels et les constantes sont tous des suites de symboles distinctes. Tous auront un nom commençant par une minuscule ou par un symbole non-alphabétique, par opposition aux variables dont les noms commenceront nécessairement par une lettre majuscule. Toutes ces données sont en réalité des termes.

Définition 2.2 (Terme)**.** *Un terme est un des éléments suivants :*
- *un symbole fonctionnel, suivi de ses arguments (des termes) entre parenthèses ;*
- *une constante du domaine ;*
- *une variable.*

Un autre élément présent dans les programmes logiques est l'**atome**.

Définition 2.3 (Atome)**.** *Un atome est un élément de programme logique, constitué d'un symbole prédicatif et de termes. Un atome s'écrit $p(t_1, t_2, ..., t_n)$ où p est un symbole prédicatif et $\forall i \in 1..n : t_i$ est un terme.*

Un appel de prédicat est donc un cas particulier d'atome.

Nous avons déjà évoqué la notion de contrainte. En voici la définition formelle.

Définition 2.4 (Contrainte)**.** *Une contrainte sur un domaine de contraintes $\mathcal{D}$ est une formule logique du premier ordre dont les termes constants, symboles prédicatifs et symboles fonctionnels appartiennent à $\mathcal{D}$.*

Les notations admises pour les contraintes ont déjà été abordées plus haut. Voici un exemple de contrainte écrite en suivant ces règles :

Exemple 2.5. $= (X, 1) \vee = (X, 2) \vee \neg(\exists Y : = (mod(X, Y), 0) \wedge < (Y, \sqrt{X}))$

En pratique, l'on se permettra d'écrire les foncteurs et prédicats binaires entre leurs arguments plutôt que de placer ceux-ci entre parenthèses, par souci de lisibilité. Voici la contrainte précédente remaniée en ce sens :

Exemple 2.6. $X = 1 \vee X = 2 \vee \neg(\exists Y : X \bmod Y = 0 \wedge Y < \sqrt{X})$

Comme décrit en détail dans [17], la programmation logique classique est en réalité une forme de $CLP(\mathcal{T})$ où $\mathcal{T}$ est le domaine de contraintes défini sur les **arbres finis**. Ce concept dépasse le cadre de ce mémoire mais permet d'appuyer le fait que la programmation logique n'est jamais qu'une forme particulière du paradigme CLP. Notre approche étant axée sur la programmation logique avec contraintes, elle sera donc assez générale pour prendre en compte les programmes logiques dans leur ensemble.

2.2 Foncteurs et termes imbriqués

Les foncteurs peuvent servir à imbriquer des structures de données en leur sein. Par exemple, $p(q(X), constante, r(X, q(Y)))$ est un terme valide. Un foncteur est avant tout un sucre syntaxique ; mis à part les symboles fonctionnels courants (tels que $+, -, /, *$ pour l'arithmétique par exemple), que nous considérerons comme implémentés nativement dans le système, il est tout à fait possible d'utiliser n'importe quel nom de foncteur avec n'importe quels arguments. Syntaxiquement, un tel foncteur est en tout point similaire à un nom de prédicat, à la différence que le prédicat devra être défini à l'aide de clauses listées ailleurs dans le programme.

Dans le cadre de ce mémoire, nous utiliserons fréquemment un foncteur particulier. Il s'agit de celui qui permet de définir des **listes**. Les listes sont l'exemple typique des structures de données imbriquées. En pratique, nous n'utiliserons pas de foncteur pur avec parenthèses, mais bien un raccourci de notation admis en Prolog. Ainsi, [ ] est une notation permettant de définir la liste vide. $[H|T]$ est une notation permettant de définir la liste constituée du terme H et ayant, comme autres éléments, ceux de la liste T. La liste T sera donc elle-même soit la liste vide ([ ]), soit constituée d'un terme $T1$ suivi des éléments d'une autre liste Ts ($[T1|Ts]$). Une liste peut être constituée de n'importe quel nombre de termes. On note $[elem_1, elem_2, ..., elem_n]$ la liste constituée des n termes $elem_1, elem_2, ..., elem_n$. La liste $[elem_1, elem_2, ..., elem_n|Ls]$ est la concaténation de la liste $[elem_1, elem_2, ..., elem_n]$ et d'une liste Ls.

Exemple 2.7. $[1, [nom_1, nom_2], 10|[4, 5]]$ *est une liste constituée des termes* $1, [nom_1, nom_2], 10, 4$ *et* 5*. Il s'agit de la concaténation de deux listes, équivalente à une seule liste qui pourrait s'écrire* $[1, [nom_1, nom_2], 10, 4, 5]$*. Remarquons que la liste compte parmi ses éléments une autre liste (*$[nom_1, nom_2]$*).*

Les listes sont utilisées pour aborder des problèmes où la taille du jeu de données en entrée est variable et inconnue au départ. Conceptuellement, elles permettent de raisonner sur des problèmes de façon générale, ce qui est souvent réalisé à l'aide d'une définition récursive des prédicats. La récursivité se fait alors sur la taille de la liste : on se trouve soit devant une liste vide, soit devant une liste ayant au moins un élément. Nous verrons des exemples d'utilisation de listes dans les chapitres suivants.

2.3 Structure des programmes logiques avec contraintes

Un programme logique est constitué d'une **base de connaissances** qui contient des *faits* et des *règles*. Ces deux types de données sont des *clauses* et forment des *prédicats*. Nous avons déjà évoqué la notion de prédicat ci-dessus ; nous allons à présent la formaliser pour définir la structure des programmes logiques avec contraintes.

Définition 2.8 (Prédicat)**.** *Un prédicat est une relation logique. Il porte un nom et est le regroupement de plusieurs clauses. Tout prédicat a zéro, un ou plusieurs arguments ; le nombre d'arguments d'un prédicat est son* ***arité****.*

Un prédicat peut être identifié de manière unique en considérant son nom et son arité. Traditionnellement, on utilise une barre oblique pour séparer ces deux constituants de l'identifiant.

Exemple 2.9. $p/3$ *fait référence au prédicat* p *d'arité* 3*.*

Dans la définition de prédicat, nous mentionnons la notion de clause. La clause est en effet l'unité constitutrice du prédicat. Elle représente une ligne de la base de connaissances et se termine toujours par un point. Voici sa définition formelle.

Définition 2.10 (Clause)**.** *Une clause est l'association d'une tête et d'un corps. Si le corps est vide, on parle de* ***fait****. Sinon, on parle de* ***règle*** *; la tête est alors séparée du corps par le symbole* $:-$*. Une tête est un atome. S'il y a un corps, il est constitué d'un(e) ou plusieurs contrainte(s) et/ou atome(s) séparés par des virgules.*

Exemple 2.11. $p(X, Y, Z) :- \geq (X, Y), \geq (Y, Z), q.$ *est une clause définissant le prédicat* $p/3$ *et se servant de variables pour définir la relation qui existe entre ses arguments. Le dernier atome est l'appel au prédicat* $q/0$*.*

Exemple 2.12. $r(X).$ *est un fait.*

Par souci de clarté, dans la suite nous suivrons les règles suivantes pour écrire un programme logique avec contraintes.

1. Le corps des clauses est, par pure lisibilité, séparé en deux « parties » :
   - La première partie représente les **contraintes** exprimées par des formules de $\mathcal{C}$. La virgule fait office de conjonction entre deux contraintes et est donc équivalente au symbole $\wedge$.
   - La seconde partie représente les **appels à d'autres prédicats** définis dans la base de connaissances. La virgule est ici une notation qui permet de séparer les différents appels.
2. Toujours par souci de lisibilité, la première partie (les contraintes) sera encadrée d'accolades. S'il n'y a pas de contrainte, on l'indiquera explicitement en écrivant une paire d'accolades vide.
3. S'il y a des appels dans la seconde partie, alors ils seront séparés des contraintes par une virgule écrite après l'accolade fermante de celles-ci.

L'on peut considérer qu'en toute généralité, une clause a *plusieurs contraintes* (reliées par des conjonctions) ou, de façon équivalente, *une seule contrainte* (éventuellement conjonction de plusieurs contraintes). Dans la suite, nous utiliserons les deux terminologies sans distinction de sens.

En guise d'exemple, le prédicat suivant détermine si oui ou non son argument est un nombre premier. Notons qu'il n'y a pas d'appels à d'autres prédicats : l'on écrit alors directement un point après l'accolade fermante des contraintes.

Exemple 2.13. $premier(X) :- \{X = 1 \vee X = 2 \vee \neg(\exists Y : X \bmod Y = 0 \wedge Y < \sqrt{X})\}.$

Remarquons que le langage CLP présenté ici est très général et conceptuel : il n'est pas un langage de programmation exécutable en tant que tel. Néanmoins, en parallèle du présent mémoire, nous avons réalisé un outil écrit en Prolog capable d'interpréter un sous-ensemble de ce langage.

2.4 Sens des programmes logiques avec contraintes

Pour lire (et comprendre) un programme logique, il faut voir chaque clause comme une implication de droite à gauche. Ainsi, $chien(X) : -\{\}, poilu(X).$ se lit « Si X est poilu, alors X est un chien ». Les clauses Prolog sont en réalité des clauses de Horn.

Définition 2.14 (Clause de Horn)**.** *Une clause de Horn est une disjonction logique d'atomes, ayant au plus un atome positif.*

Toute clause en programmation logique est une clause de Horn. En effet, notons d'abord que toute implication peut s'écrire comme une disjonction. Or, dans le cas des clauses logiques, nous avons une tête (le côté gauche de l'implication) et un corps (le côté droit, qui est une conjonction). Par exemple, $p : -q, r, s.$ [1] peut s'écrire $q \wedge r \wedge s \Rightarrow p$, ce qui aura la même table de vérité que $\neg q \vee \neg r \vee \neg s \vee p$. Intuitivement, on a que, si q, r et s sont vrais, alors pour que l'implication soit valide, il faudra que p soit aussi vrai.

Il faut donc comprendre une base de connaissances logique comme une liste d'implications. Dans le cas des faits, le corps est vide ; il est alors équivalent à $true$ qui, en Prolog, est un prédicat natif qui réussit toujours. $r(X).$ est ainsi équivalent à $r(X) : -true.$ et donc, conceptuellement, à $true \Rightarrow r(X)$: une tautologie. C'est cela que nous avons appelé un *fait* plus haut.

S'il y a plusieurs clauses pour un même prédicat, c'est l'ensemble de ces clauses qui déterminera le sens du prédicat. Le prédicat est alors défini par plusieurs implications. Dans l'exemple suivant écrit en $CLP(\mathbb{R})$, le prédicat $non\_nul/1$ est défini comme vrai dans deux cas qui correspondent à deux implications : $X > 0 \Rightarrow non\_nul(X)$ et $X < 0 \Rightarrow non\_nul(X)$.

Exemple 2.15.
```
non_nul(X):- {X > 0}.
non_nul(X):- {X < 0}.
```

Envisageons à présent un programme complet écrit en $CLP(W)$, avec W un domaine admettant comme constantes $broussaille$ et $tartuffe$ et n'admettant aucun symbole fonctionnel.

```
poilu(broussaille).
aboie(broussaille).
parent(tartuffe, broussaille).
chien(X) :- {}, poilu(X), aboie(X).
chien(X) :- {}, parent(X,Y), chien(Y).
```

Dans ce programme, il y a deux règles définissant le prédicat *chien* : il faut soit être poilu et aboyer, soit avoir un autre chien comme parent. Il y a également deux faits : *broussaille* (une valeur constante) est poilu et aboie.

Remarquons que les arguments des prédicats peuvent avoir un certain rôle, ou **mode**, qui est soit d'entrée, soit de sortie. Un programme capable de calculer une factorielle par exemple, pourrait s'écrire comme suit : $facto(X, FactoX).$ où le premier argument est le nombre dont il faut calculer la factorielle et le deuxième argument est le résultat du calcul de la factorielle. Si on imagine facilement de « fournir » le premier argument (c'est-à-dire d'y placer une constante) pour avoir le second (une variable) en retour, cela ne les empêche pas d'être utilisés dans un autre sens. Par exemple, l'on pourrait chercher, connaissant $FactoX$, à connaître la valeur de X. Une autre utilisation serait de ne fournir aucun des deux arguments (laisser deux variables) dans la requête et d'ainsi demander au système de lister les couples $(nombre, factorielle(nombre))$. Enfin, si l'on instancie les deux arguments, par exemple avec la requête $facto(3, 6)$, il s'agit d'une *vérification* : le programme renverra $true$ si 6 est bien la factorielle de 3, $false$ sinon. Nous verrons le mécanisme d'exécution d'une requête logique plus en détail dans la section suivante.

Le formalisme des clauses de Horn a donc l'avantage d'être plus général que les autres paradigmes usuels : dans la suite, l'on pourra chercher à comprendre le sens d'un programme non pas pour un seul mode des variables (entrée/sortie), mais bien pour tous les modes possibles. C'est aussi cela qui nous motive à utiliser les clauses de Horn comme formalisme dans lequel nous pourrons entreprendre la comparaison d'algorithmes.

Notons enfin qu'en programmation logique, un prédicat matérialise une *relation* et il n'y a donc aucun sens à chercher la « valeur » d'un prédicat comme on le ferait pour une fonction en programmation non-logique. Le

1. Les accolades n'ont pas d'importance dans cet exemple. Comme nous l'avons dit plus tôt, séparer les appels de prédicats des contraintes n'est qu'une convention de lisibilité que nous suivons.

prédicat décrit une relation entre ses arguments, qui tient sous certaines conditions, matérialisées dans le corps de ses clauses (voir [7] pour plus d'informations à ce sujet).

Techniquement, nous pourrions également parler de **clauses de Horn contraintes** (CHC) plutôt que de CLP. Comme expliqué dans [11], ce terme n'a pas d'implication opérationnelle. Par opposition, la CLP, en principe, contient l'idée d'une tentative de preuve d'insatisfiabilité sur base des formules (clauses) données. Dans la suite de ce mémoire, nous utiliserons les deux termes sans distinction.

2.5 Exécution des programmes logiques avec contraintes

Dans les grandes lignes, la méthode d'exécution d'un programme Prolog [2] est la suivante ([7]) :

1. Une requête est posée à la base de connaissances. Par exemple, *chien(tartuffe)* pour savoir si *tartuffe* est un chien ; ou *chien(X)* pour recevoir la séquence des noms des chiens, qui s'unifieront avec l'argument X laissé en variable dans ce but.
2. Le système utilise sa base de connaissances (implications) pour démontrer le théorème contenu dans la requête ou prouver qu'il est faux.

L'ordre des clauses n'a, conceptuellement, pas d'influence sur le sens déclaratif du programme. Toutefois, en pratique, la plupart des moteurs d'inférences, dont celui de Prolog, essaient d'unifier les clauses dans l'ordre dans lequel elles sont écrites dans la base de connaissances. De même pour l'ordre des atomes à unifier quand une requête ou le corps d'une clause en présentent plusieurs. Le fonctionnement du moteur d'inférences de Prolog, ainsi que les notions d'arbre de dérivation, d'unification et de *backtracking* sont expliquées en détail dans [7].

Dans le cas des programmes logiques avec contraintes, certaines étapes sont ajoutées au processus décrit ci-dessus. Elles permettent d'optimiser l'exécution en présence de contraintes. Pour cela, à l'exécution, le système va alimenter un **sac de contraintes**, disons SC, qui vérifie $SC \in \mathcal{P}(\mathcal{C})$ où $\mathcal{C}$ est l'ensemble des contraintes que l'on peut écrire sur le domaine et $\mathcal{P}(\mathcal{C})$ est l'ensemble des parties de $\mathcal{C}$. Le système va étoffer SC au fil des clauses qu'il rencontre et éliminer les chemins de l'arbre de dérivation logique qui rendent SC insoluble. Le but de ce processus est de pouvoir prouver la non-satisfaisabilité d'une requête plus facilement.

Définition 2.16 (Sac de contraintes insoluble)**.** *Soit $\mathcal{D}$ un domaine de contraintes sur l'alphabet Σ. Un sac de contraintes SC est dit insoluble dans $\mathcal{D}$ s'il est impossible d'instancier toutes les variables apparaissant dans ses contraintes avec des valeurs de $\mathcal{D}$ de façon à ce que les contraintes soient toutes satisfaites.*

Considérons un domaine de contraintes quelconque $\mathcal{D}$. Nous pouvons à présent introduire les étapes propres à l'exécution des programmes écrits en $CLP(\mathcal{D})$.

1. SC est initialisé à $\emptyset$ avant l'exécution de la requête.
2. Après l'unification d'un appel du prédicat ϕ/n du type $\phi(B_1, B_2, ..., B_n)$ à la tête d'une clause C du type $\phi(A_1, A_2, ..., A_n)$ (les variables de la clause C en question ayant été renommées de façon unique avant l'unification), SC est mis à jour selon la règle $SC \leftarrow SC \cup Equ \cup Contr$ où
   — Equ est un ensemble de n contraintes du type $A_i = B_i$ avec $i \in \{1..n\}$;
   — $Contr$ est l'ensemble de contraintes de la clause C renommée.
3. Si, à ce moment, SC est insoluble dans $\mathcal{D}$, alors la branche actuelle de l'arbre de dérivation logique se termine sur un échec et le mécanisme de *backtracking* est activé. Dans ce cas, le *backtracking* implique également la restitution de SC dans l'état dans lequel il était au point du programme auquel renvoie le *backtracking*.

2.6 Exemple de programme logique avec contraintes

Nous allons écrire un programme en $CLP(\mathbb{N}_0)$.

```
premier(X) :- {X = 1}.
premier(X) :- {X = 2}.
premier(X) :- {X > 2}, divisible_par_un_seul_nombre(X).

divisible_par_un_seul_nombre(X):- {}, pas_divisible_par(X, 2).
```

2. Il s'agit même de la méthode d'exécution généralement admise pour tous les programmes logiques, indépendamment du langage qui les exprime ([42]).

```
pas_divisible_par(X, Y):-  {Y < X, X mod Y = Z, Z > 0}, pas_divisible_par(X, Y + 1).
pas_divisible_par(X, X) :- {}.
```

Plusieurs remarques s'imposent devant ce programme :
- Les quatrième et sixième clauses n'ont pas de contraintes.
- Dans la définition du prédicat *premier*/1 (défini comme vrai si et seulement si son argument est un nombre premier), les contraintes servent à distinguer les cas. On aura ainsi :
  - Si $X = 1$ alors le prédicat *premier*/1 est vrai.
  - Si $X = 2$ alors le prédicat *premier*/1 est vrai.
  - Si $X > 2$ alors le prédicat *premier*/1 sera vrai si *divisible_par_un_seul_nombre*(X) est vrai.
  - Sinon, c'est-à-dire si $X > 2$ alors que *divisible_par_un_seul_nombre*(X) est faux, alors le prédicat *premier*/1 est faux. Cela n'est pas écrit tel quel dans le programme ; il s'agit en réalité de l'**hypothèse du monde clos** souvent admise dans la programmation logique [3]. Cette hypothèse dit que les seules conditions pour qu'un prédicat soit vrai sont celles que l'on retrouve dans les clauses de Horn qui le définissent dans le programme. Autrement dit, tout ce qui n'est pas déclaré dans un programme logique est considéré comme faux ([7]).
- Dans l'avant-dernière clause, nous pouvons remarquer que la priorité des opérateurs fonctionnels (comme *mod*) est plus importante que celle des symboles prédicatifs (comme =).
- Dans la dernière clause, un raccourci syntaxique a été utilisé. Il s'agit plus précisément de *pattern matching*. Les deux arguments de *pas_divisible_par*/2 sont une seule et même variable (X), ce qui est équivalent à utiliser deux variables nommées différemment en les déclarant égales dans les contraintes.

Comparons le prédicat *premier*/1 avec celui de l'exemple 2.13. Dans l'exemple 2.13 l'on a défini le prédicat *premier*/1 en une seule clause. Notons que cette clause ne présente que des contraintes, et pas d'appel vers d'autres prédicats. La concision de cette écriture tient à l'utilisation de connecteurs logiques et de formules mathématiques logiques. L'écriture d'un programme logique avec contraintes laisse, en général, beaucoup de liberté sur les notations.

Par opposition, l'exemple ci-dessus est plus verbeux et tend à utiliser une notation plus proche du Prolog, c'est-à-dire sans utilisation des connecteurs logiques et des opérateurs de haut niveau (à moins qu'ils ne soient définis nativement). En pratique, c'est à ce type de programmes que nous aurons affaire pour plusieurs raisons. D'une part, ils sont interprétables par l'outil Prolog développé en parallèle du présent mémoire. D'autre part, les outils existants de génération de programmes CLP à partir de code assembleur ([43], [12] et [20]) génèrent à l'heure actuelle typiquement des programmes de cet acabit. À titre personnel, nous estimons également ces programmes plus faciles à lire et plus intuitifs à la compréhension. La concision de l'exemple 2.13, pour d'autres exemples plus compliqués, serait difficile à appréhender. Pour résumer, notre style restera proche du Prolog décrit dans [42] mais certains raccourcis de notations (toujours acceptés par le formalisme de CLP) seront parfois utilisés dans les exemples.

L'outil que nous avons développé pour supporter la recherche présentée dans ce mémoire est spécifique au $CLP(\mathbb{N})$. C'est pourquoi, bien que notre approche soit assez générale pour prendre en compte n'importe quel domaine de calcul, nous utiliserons souvent du code $CLP(\mathbb{N})$ dans les exemples.

3. C'est le cas en Prolog par exemple.

Chapitre 3

Cadre de travail pour l'équivalence de programmes logiques avec contraintes

Dans ce chapitre, nous établissons un cadre de travail dans lequel il nous sera possible de déterminer si deux programmes représentés sous forme de clauses de Horn réalisent le même algorithme ou non. Par « réalisent le même algorithme », nous entendons « effectuent le même calcul en suivant le même algorithme ». La difficulté réside dans le fait que des différences syntaxiques et structurelles peuvent apparaître dans les programmes à comparer. Ce qui nous intéresse, c'est de comparer la sémantique de ces programmes, en nous permettant de faire abstraction de leurs différences syntaxiques superficielles, tout en nous interdisant de faire abstraction d'écarts structurels plus importants, potentiels témoins de différences au niveau algorithmique. Notre cadre de travail sera amené dans cette optique, mais restera suffisamment large pour englober des comparaisons tolérant certains écarts sémantiques entre les programmes à comparer. Inversement, il peut être intéressant de ne pas se limiter à l'équivalence sémantique, et d'exiger que les programmes à comparer présentent d'autres propriétés pour que leur comparaison résulte en une réponse positive. Nos idées seront également paramétrables dans ce sens. En un mot, la souplesse du cadre de travail sera ajustable en fonction des applications. Différents degrés de souplesse seront discutés dans le chapitre 5.

Après avoir présenté ce cadre de travail, nous introduirons quelques transformations de programmes logiques qui nous semblent élémentaires pour une comparaison entre deux programmes CLP. Nous expliquerons comment paramétrer ces transformations pour leur donner plus ou moins de latitude. Enfin, nous présenterons un algorithme de comparaison de programmes naïf fondé sur notre cadre de travail, et conclurons avec des exemples d'utilisation des idées présentées.

3.1 Transformations de programmes logiques avec contraintes

Avant toute chose, il convient d'expliciter ce que nous entendons par l'opérateur d'égalité entre deux programmes CLP.

Définition 3.1 (Égalité de programmes)**.** *Soient P et Q deux programmes logiques avec contraintes écrits en $CLP(X)$. Soit $texte(P)$ le texte, sans espace, sans tabulation, sans indentation, et sans retour à la ligne, de P, et $texte(Q)$ l'équivalent pour Q. Alors P et Q sont égaux, ce que l'on note $P = Q$, si et seulement si $texte(P)$ est identique en tous points à $texte(Q)$. L'on dira que P et Q sont strictement égaux, ce que l'on notera $P \overset{strict}{=} Q$, si cette relation tient en conservant les espaces, tabulations, indentations et retours à la ligne.*

En d'autres mots, nous ne considérons deux programmes comme égaux que si, mis à part tous les artifices d'espacement, leurs codes sont rigoureusement identiques.

Voyons à présent ce que l'on entend par transformation de programme.

Définition 3.2 (Transformation de programme)**.** *Soit $\mathcal{L}$ un langage de programmation. Une transformation de programme y pour le langage $\mathcal{L}$ est une fonction qui, à tout programme P_1 écrit en $\mathcal{L}$ fourni en entrée, associe un programme $y(P_1)$ écrit en $\mathcal{L}$.*

Les transformations de programmes que nous utilisons dans la suite vont avoir la particularité de pouvoir être appliquées à des programmes logiques avec contraintes. Dans nos exemples, la famille de langages de programmation considérée sera donc toujours $CLP(X)$.

Exemple 3.3. *Imaginons une transformation de programmes CLP qui renomme toutes les variables d'un programme. Appliquer cette transformation au code présenté à gauche ci-dessous pourrait résulter en l'obtention du code présenté à droite.*

```
p(K,L,M,N) :- {N > a, K > b}, q(L,M).
p(K,L,M,N) :- {N =< a}, q(L,M).
q(W,X) :- {W > X}.
```

```
p(A,B,C,D) :- {D > a, A > b}, q(B,C).
p(A,B,C,D) :- {D =< a}, q(B,C).
q(A,B) :- {A > B}.
```

Notons que, dans la définition ci-dessus, l'on parle de fonction en toute généralité. Une transformation de programme ne doit être ni injective, ni surjective. Le seul critère important pour une telle procédure est qu'elle puisse être appliquée à tout programme écrit dans le langage considéré. Néanmoins, rien n'empêche une transformation de ne pas modifier certaines de ses entrées, c'est-à-dire que l'on pourrait imaginer une transformation y telle que $y(P_1) = P_1$ pour un certain programme P_1.

3.2 Fonction sémantique des programmes CLP

Vu la définition très large de transformation de programme que nous avons donnée ci-dessus, une infinité d'autres programmes pourraient être générés par le renommage de variables dans l'exemple 3.3, dépendant de son fonctionnement interne. L'on peut en effet toujours trouver de nouveaux noms de variables qui serviraient au renommage. En fait, quand nous parlons d'une transformation qui « renomme toutes les variables », un flou est volontairement laissé. Cette transformation va-t-elle associer un nom de variable donné à un autre (par exemple, tous les K du programme de départ sont remplacés par A) ? Va-t-elle faire cette association clause par clause seulement ? Ou peut-on même imaginer de remplacer chaque variable par un nouveau nom arbitraire, quitte à ce qu'une même variable apparaissant deux fois dans une clause soit renommée de deux façons différentes ?

Dans le cadre de ce mémoire, nous chercherons des transformations qui tendent à **préserver le sens des programmes**. C'est en effet en préservant le sens des programmes manipulés que l'on pourra affirmer que deux de ces programmes sont équivalents algorithmiquement. Dans l'exemple du renommage de variables, la version qui préserve au mieux le sens des programmes, intuitivement, ne pourra se permettre de renommer une même variable, dans une même clause, en deux nouveaux noms de variables différents. Clairement, le sens du programme logique en serait potentiellement altéré. De même, il serait préférable que cette transformation n'associe pas le même nouveau nom de variable à deux variables différentes. Avant de définir le renommage de variables, nous allons nous pencher sur le sens à donner aux programmes logiques avec contraintes.

Le sens des programmes s'exprime au travers d'une *fonction sémantique*, notée *sem*, qui paramétrise notre cadre de travail. Une fonction sémantique universelle n'existe ni pour les programmes CLP, ni pour les programmes logiques sans contraintes. En Prolog, par exemple, l'ordre des clauses a un impact sur le sens d'un programme ; ce qui n'est pas le cas pour d'autres sémantiques purement déclaratives. Dans la suite de ce chapitre, nous utiliserons la sémantique algébrique de programmes logiques avec contraintes décrite dans l'article [18]. Nous noterons cette sémantique sem_0. Il s'agit d'une extension de la sémantique algébrique admise pour les programmes logiques sans contraintes. Cette dernière considère que le plus petit modèle donnant une interprétation du programme logique définit sa sémantique [1]. Dans le cas des programmes CLP, nous sommes en présence de contraintes : les modèles devront donc respecter le sens qui est donné à ces contraintes par leur interprétation dans le domaine de calcul dont elles sont issues. Le plus petit modèle d'un programme en $CLP(X)$ est donc un modèle qui, en interprétant correctement les contraintes du programme, ne pourra considérer un prédicat comme « faux » si le corps d'au moins une des clauses qui le composent est considéré comme « vrai ». Lorsque nous parlons d'interprétation correcte, cela signifie que la sémantique du domaine X est conservée. Dans le cas où X est numérique, cela signifiera par exemple que le symbole prédicatif « $\leq$ » sera bien interprété comme « plus petit ou égal ».

Cette sémantique à la croisée de celles des programmes logiques et des domaines de contraintes est liée dans l'article [18] à une sémantique opérationnelle. La sémantique opérationnelle en question est responsable de déterminer quand un atome est « considéré comme vrai » ou « considéré comme faux ». Nous avons ainsi déjà mentionné le moteur d'inférences de Prolog. Celui-ci prend en considération l'ordre des clauses d'un programme, puisqu'il tente d'unifier les atomes avec les têtes des clauses du programme dans l'ordre dans lequel elles sont définies. Ce comportement peut entraîner des situations où une dérivation infinie est lancée alors que le simple fait de commencer les tentatives d'unifications par une autre clause aurait amené à une réponse (« vrai » ou « faux »). D'autres propriétés peuvent être exigées au niveau de la sémantique opérationnelle. Suivant la

1. Il s'agit, plus précisément, du plus petit modèle *de Herbrand*. Par simplicité, nous parlons ici de modèle logique au sens commun du terme. Les modèles de Herbrand sont discutés dans [24] et [16].

terminologie amenée dans [18], nous considérerons que notre moteur de résolution CLP se comporte « bien » (*well-behaved*) ; de même, nous chercherons une stratégie de sélection d'atomes à unifier dite « juste » (*fair*). La sémantique opérationnelle cachée derrière sem_0 est, au final, telle que

— l'ordre des clauses n'a pas d'impact sur le sens du programme ;
— l'ordre des atomes dans le corps d'une clause n'a pas d'impact sur le sens du programme ;
— l'exécution du programme CLP est, du reste, fidèle à la description que nous en avons faite dans la section 2.5. En particulier, l'hypothèse du monde clos est d'application.

La sémantique sem_0 nous semble appropriée dans notre approche car elle est purement déclarative. Elle dénote le sens d'un programme CLP par la conjonction de ses clauses logiques ([18]). Les différences introduites, souvent pour des raisons pratiques d'apport de déterminisme dans l'exécution du flux du programme, par un moteur de résolution en conditions réelles (telle que celui de Prolog) cassent en partie cet aspect déclaratif. Pour résumer, sem_0 est suffisamment générale et nous paraît intuitive eu égard à la description du paradigme CLP faite dans le chapitre précédent. Nous n'avons ici présenté que succinctement cette sémantique, qui est décrite plus en avant dans l'article [18]. Les auteurs sont ceux mêmes qui ont défini le paradigme CLP dans l'article [16], ce qui achève d'en faire à nos yeux une tendance à suivre.

La définition suivante caractérise les transformations dont l'application préserve une fonction sémantique.

Définition 3.4 (Transformation de programme préservatrice de sémantique)**.** *Soit un langage de programmation $\mathcal{L}$ et une fonction sémantique sur ce langage sem. Une transformation de programme y est préservatrice de sem pour les programmes écrits en $\mathcal{L}$ si, pour tout programme P écrit en $\mathcal{L}$, $sem(y(P)) = sem(P)$.*

Comme nous le disons dans la section suivante, les transformations admises dans notre cadre de travail dépendront du contexte. Toutes ne devront pas nécessairement préserver une sémantique du programme de départ, dépendant des applications. Néanmoins, les principales transformations introduites dans ce chapitre et dans le chapitre suivant respectent cette propriété. En l'occurrence, nous centrerons ce chapitre et le chapitre suivant sur la fonction sémantique sem_0, quand ce sera possible [2].

Une transformation trivialement préservatrice de sem_0 est la transformation identité.

Définition 3.5 (Transformation : identité)**.** *La transformation identité est une transformation de programme y telle que pour tout programme P, $y(P) \overset{strict}{=} P$. La transformation identité est également notée* id*.*

Il est évident qu'une transformation de renommage de variables, pour peu qu'elle soit définie correctement, sera également préservatrice de sem_0.

Définition 3.6 (Transformation : renommage de variables)**.** *Le renommage de variables est une transformation de programme y telle que pour tout programme P_1, $y(P_1) = P_2$ où P_2 est identique à P_1, à la seule différence près que les variables des clauses $i_1, i_2, ..., i_k$, avec $k \in 1..n$ où n est le nombre de clauses du programme, et $\forall j \in 1..k : i_j \in 1..n$, ont été renommées en suivant une fonction de renommage $f : Var \to Var$ où Var est l'ensemble des noms de variables admis dans le formalisme considéré.*
La transformation de renommage de variables est notée renommage_vars*.*

Le numéro des clauses à renommer, et la fonction f mentionnée dans la définition, sont en réalité des *paramètres* de cette transformation. Nous revenons sur ce concept plus loin dans ce chapitre ; pour l'instant, nous considérons la transformation dans sa forme la plus générale.

Le renommage peut également s'appliquer aux prédicats.

Définition 3.7 (Transformation : renommage de prédicats)**.** *Le renommage de prédicats est une transformation de programme y telle que, pour tout programme P_1, $y(P_1) = P_2$ où P_2 est identique à P_1, à la seule différence près que, pour tout prédicat p/a, dans toutes les occurrences de déclarations et d'appels à p/a, le nom p peut avoir été remplacé par un autre symbole prédicatif.*
La transformation de renommage de prédicats est notée renommage_preds*.*

Notons que les deux transformations présentées ci-dessus sont non-déterministes. Une application du renommage de prédicats pourrait, par exemple, renommer un prédicat $p/3$ en $q/3$, mais aussi en $r/3$. Les noms des prédicats n'ayant, opérationnellement, aucun impact sur le sens des programmes, ce non-déterminisme sera accepté par notre cadre de travail. Il en va de même pour le renommage de variables.

2. Plus loin dans ce chapitre, nous définissons sem_1, une variante de sem_0. Dans le chapitre suivant, nous introduisons encore deux nouvelles fonctions sémantiques, sem_2 et sem_3.

En tant que telles, dans leur définition, ces transformations ne préservent pas nécessairement sem_0. Par exemple, il suffirait qu'un prédicat p/a soit renommé avec un nom de prédicat q/a existant déjà dans le programme de départ, pour que *renommage_preds* ne préserve pas le sens déclaratif des programmes qu'elle manipule. Or il peut être intéressant, dans notre démarche, de définir un maximum de transformations préservatrices de sem_0, comme cette propriété peut avoir un grand intérêt dans l'équivalence algorithmique. Ce flou sur le comportement exact des transformations de renommage est laissé intentionnellement, et sera levé lorsque nous introduirons les paramètres et méta-paramètres dans la section 3.6.

3.3 Équivalence et connexion algorithmiques

Dans la suite, nous allons considérer que nous disposons d'un ensemble de transformations de programmes logiques avec contraintes noté $\mathcal{Y}$. Avec un tel ensemble, il est possible de chaîner les transformations sur un programme de départ ([43]).

Définition 3.8 (Série de $\mathcal{Y}$-transformations)**.** *Soient $\mathcal{D}$ un domaine de calcul, $\mathcal{Y}$ un ensemble de transformations de programmes $CLP(\mathcal{D})$ et P_1 un programme écrit en $CLP(\mathcal{D})$. Alors une série de $\mathcal{Y}$-transformations à partir de P_1 est une suite finie de programmes écrits en $CLP(\mathcal{D})$, notée $\langle P_1, P_2, ..., P_n\rangle$ où $\forall i \in \{2..n\} : P_i$ est obtenu en appliquant une transformation $y \in \mathcal{Y}$ à P_{i-1}.*

Par exemple, si nous désirons effectuer un renommage de variables sur toutes les clauses du programme, mais avec une fonction de renommage différente pour chaque clause à renommer, alors il s'agira d'utiliser une série de transformations, constituée exclusivement de renommages de variables.

Une série de transformations peut avoir la particularité de conserver le sens des programmes.

Définition 3.9 (Série de $\mathcal{Y}$-transformations préservatrice de sémantique)**.** *Soit $\langle P_1, P_2, ..., P_n\rangle$ une série de $\mathcal{Y}$-transformations à partir d'un programme P_1 écrit en $\mathcal{L}$. Cette série de $\mathcal{Y}$-transformations est dite préservatrice de la sémantique sem si $\forall i \in \{2..n\} : sem(P_i) = sem(P_{i-1})$.*

En particulier, si toutes les transformations de $\mathcal{Y}$ sont préservatrices d'une sémantique *sem*, alors toute série de $\mathcal{Y}$-transformations sera préservatrice de *sem*. De plus, la série de transformations vide (sans effet sur le programme initial) est préservatrice de toutes les sémantiques.

Le chaînage de transformations va nous permettre de découvrir une propriété intéressante pouvant lier deux programmes : la $\mathcal{Y}$-connexion.

Définition 3.10 ($\mathcal{Y}$-connexion)**.** *Soient deux programmes P_1 et Q_1 écrits en $CLP(\mathcal{D})$. S'il existe une série de $\mathcal{Y}$-transformations $\langle P_1, P_2, ..., P_{n-1}, R\rangle$ et une série de $\mathcal{Y}$-transformations $\langle Q_1, Q_2, ..., Q_{k-1}, R\rangle$, alors on dira que les algorithmes capturés par P_1 et Q_1 sont* ***$\mathcal{Y}$-connectés****.*
Si la même relation tient entre les programmes P_1 et Q_1, mais qu'en plus, les deux séries de transformations menant à R sont préservatrices d'une sémantique sem, on dira que P_1 et Q_1 sont ***$\mathcal{Y}$-équivalents*** *par rapport à sem.*

Intuitivement, cette définition signifie que, pour que deux programmes soient connectés par un ensemble de transformations $\mathcal{Y}$, il suffit de trouver deux séries de transformations puisées dans $\mathcal{Y}$, chacune appliquée à l'un des deux programmes, telles que les derniers programmes générés par les deux séries soient égaux.

Il est en réalité question de connexion (ou d'équivalence) entre deux **algorithmes**. Par abus de langage, nous nous permettons de parler de connexion (ou d'équivalence) entre *programmes*.

Afin de rester le plus général possible, nous étudierons dans la suite la propriété qu'ont des programmes à être $\mathcal{Y}$-connectés. Tous les résultats et observations que nous ferons resteront valides dans le cas où l'on cherche une $\mathcal{Y}$-équivalence (donc dans le cas où la connexion par $\mathcal{Y}$ se fait en préservant une certaine fonction sémantique des programmes). Cela dit, notre approche visera à définir un maximum de transformations préservatrices de sem_0, vu l'enjeu que cette propriété peut avoir sur la comparaison algorithmique. En particulier, $\mathcal{Y}$-équivalence et équivalence sémantique sont reliées de la façon suivante.

Théorème 3.11. *Soient deux programmes P_1 et Q_1 $\mathcal{Y}$-équivalents par rapport à une sémantique sem. Si sem ne considère pas les différences d'espacement dans le texte des programmes comme des différences de sens, alors $sem(P_1) = sem(P_2)$.*

Démonstration. P_1 et Q_1 étant $\mathcal{Y}$-équivalents par rapport à *sem*, il existe une série de $\mathcal{Y}$-transformations préservatrice de *sem* $\langle P_1, P_2, ..., P_k\rangle$ et une série de $\mathcal{Y}$-transformations préservatrice de *sem* $\langle Q_1, Q_2, ..., Q_l\rangle$ telles que

$P_k = Q_l$, et donc $sem(P_k) = sem(Q_l)$, puisque les programmes sont identiques aux symboles d'espacement près. Ces deux séries de transformations étant préservatrices de sem, on a $sem(P_1) = sem(P_2) = ... = sem(P_k) = sem(Q_l) = sem(Q_{l-1}) = ... = sem(Q_1)$. ■

3.4 Points d'entrée

Jusqu'ici, nous avons considéré que nous pourrions effectuer n'importe quelle requête aux programmes logiques manipulés. Dans les faits, les programmes CLP que nous allons considérer dans la suite contiennent tous un **point d'entrée**. Il s'agit du premier prédicat du programme, qui doit être appelé pour exécuter celui-ci.

Le besoin des points d'entrée est motivé par deux observations. D'abord, les programmes CLP que nous manipulons proviennent essentiellement d'outils de traduction de code binaire ([20], [12], [43]) en CLP. Ce code binaire est le résultat d'une compilation depuis du code Java ou C. Le code que nous manipulons est donc issu d'un langage impératif, et a dès lors systématiquement un point d'entrée puisque la notion d'ordre des instructions existe dans ce type de langage de programmation. Cela n'aurait pas de sens de vouloir comparer ces programmes sur n'importe quelle requête de n'importe quel prédicat en leur sein. En effet, la structure interne des programmes peut varier ; c'est leur cœur algorithmique qui nous intéresse.

Deuxièmement, certaines transformations se retrouveraient impactées d'une façon non souhaitable si nous ne prenions pas en compte un point d'entrée. En effet, certaines transformations, comme le dépliage (voir chapitre suivant), peuvent nous amener à une situation où une portion de code est totalement isolée du reste du code, dans le sens où aucun appel de prédicat ne lie ces deux sous-parties. Dans ce cas, nous arrivons à une situation où chaque programme contient en réalité plusieurs sous-programmes [3]. Il n'y aurait pas d'intérêt à comparer ces programmes sur chacune de leurs clauses. Nous illustrons ce point à l'aide de l'exemple suivant qui présente deux programmes sans point d'entrée :

Exemple 3.12.

```
a(X,Y) :- {X > Y}, c(Y).
a(X,Y) :- {Y >= X, X > 20}, c(Y).
b(X,Y) :- {X > 20}, c(Y).
c(X) :- {X < 40}.
```

```
a(L,M) :- {L > M}, c(M).
a(L,M) :- {M >= L, L > 20}, c(M).
c(L) :- {L > 40}.
```

Dans cet exemple, le programme de gauche contient une clause supplémentaire par rapport à celui de droite. Il est trivial de prouver que mis à part cela, les deux programmes sont identiques à un renommage de variables près.

La troisième clause du programme de gauche est inaccessible par les autres clauses, dans le sens où aucune autre clause ne contient d'appel vers $b/2$. De même pour le prédicat $a/2$, non appelé par un autre prédicat. À présent, comment déterminer si oui ou non les deux programmes encapsulent le même algorithme ? Deux possibilités s'offrent à nous :

— Soit nous considérons que, pour chaque prédicat du premier programme, doit exister un prédicat dans le second programme réalisant le même calcul.
— Soit nous considérons qu'il existe un point d'entrée dans chaque programme. Fixons le point d'entrée du programme de droite à $a/2$. Si le point d'entrée du programme de gauche est également $a/2$, alors nous pouvons faire abstraction de $b/2$ et les programmes sont clairement équivalents. En revanche, si le point d'entrée du programme de gauche est $b/2$, alors les programmes ne sont pas équivalents.

En un certain sens, la première des deux options est trop contraignante, puisque l'équivalence de programmes est alors déterminée par leur structure syntaxique. La deuxième option, en contraste, peut paraître réductrice puisqu'elle ne considère que des « blocs » de prédicats de chaque programme. Si un même programme contient plusieurs algorithmes séparés dans le code, un seul de ceux-ci (celui défini par le point d'entrée) sera pris en compte. Cette option est donc adaptée à notre cas puisque nous étudions l'équivalence de deux algorithmes et non de deux programmes complets. Du reste, elle ne nous semble pas trop contraignante dans notre approche puisque, comme déjà dit plus haut, les programmes manipulés, de par leur paradigme initial, contiennent un point d'entrée (qui correspond à la première instruction d'une méthode `main`, par exemple).

Il nous semble donc préférable de suivre la seconde des deux options exposées ci-dessus. Définissons sem_1 comme étant la version de sem_0 qui considère que chaque programme a un point d'entrée, et que le sens d'un programme est entièrement défini par les clauses accessibles depuis ce point d'entrée. Le résultat suivant est intuitif ; sa réciproque n'est pas vraie.

3. Avec le dépliage, en particulier, cette situation ne fait que s'empirer puisque le nombre de sous-programmes a tendance à grandir à chaque dépliage effectué.

Proposition 3.13. *Toute transformation préservatrice de sem_0 est préservatrice de sem_1.*

La sémantique sem_1 est plus générale que sem_0 dans le sens où de plus nombreux programmes seront $\mathcal{Y}$-équivalents par rapport à sem_1 que par rapport à sem_0. Cela correspond au fait que sem_1 ne s'intéresse qu'à la sémantique des clauses accessibles depuis le point d'entrée, là où sem_0 considère également toutes les autres clauses du programme. Nous dirons que sem_1, pour sa prise en compte d'un point d'entrée par programme, est une sémantique *orientée point d'entrée.*

Dans la suite, le point d'entrée des programmes que nous considérons portera, par convention, le nom *p_start*, à moins que nous ne spécifions explicitement un autre nom.

Une transformation intéressante en présence de points d'entrée est le nettoyage. Cette transformation prend appui sur la notion d'accessibilité entre prédicats.

Définition 3.14 (Prédicat accessible)**.** *Soit P un programme logique avec contraintes, et p/n un prédicat défini dans ce programme. Le prédicat p/n est dit accessible depuis un prédicat q_1/m_1 s'il existe un chemin $q_1/m_1 \rightarrow q_2/m_2 \rightarrow ... \rightarrow q_l/m_l$ où*
— *$\forall i \in 1..l : q_i/m_i$ est défini dans P ;*
— *$q_l = p$ et $m_l = n$;*
— *$\forall i \in 2..l : q_i/m_i$ est appelé dans le corps d'une des clauses définissant le prédicat q_{i-1}/m_{i-1}.*
Un prédicat est, en outre, toujours accessible depuis lui-même.

Définition 3.15 (Transformation : nettoyage)**.** *Soit P_1 un programme logique avec contraintes. La transformation de nettoyage est une transformation de programme y, paramétrée par un certain prédicat q/m défini dans P_1 et telle que $y(P_1)$ renvoie un programme P_2 où :*
— *tout prédicat de P_1 accessible depuis q/m se retrouve identiquement défini dans P_2 ;*
— *tout autre prédicat ne se retrouve pas dans P_2.*
Cette transformation se note nettoyage*.*

Notons que cette transformation est préservatrice de sem_1 **si** q/m **est le point d'entrée du programme**. Dans le cas contraire, le nettoyage pourrait supprimer des clauses pourtant constitutives du programme P_1. Nous revenons sur ce point dans la section 3.6.

Dans l'exemple 3.12, en considérant que le point d'entrée des deux programmes est le prédicat $a/2$, il existe deux séries de transformations applicables aux programmes qui permettraient de conclure à une connexion entre les deux algorithmes capturés. Cette série de transformations se fait en considérant l'ensemble de transformations $\mathcal{Y}$ suivant :

$$y_1 = nettoyage$$
$$y_2 = renommage\_vars$$
$$y_3 = id$$

En effet, l'application de y_1 au programme de gauche, avec comme prédicat de départ $a/2$, générera un nouveau programme identique, à l'exception du prédicat $b/2$ qui aura disparu. L'application de y_2 à chacune des clauses du programme ainsi généré, à chaque fois en suivant la fonction de renommage $\{X \rightarrow L, Y \rightarrow M\}$, générera un nouveau programme identique en tout point au programme de droite. Nous obtenons ainsi une série de transformations applicable au programme de gauche et menant à un nouveau programme.

Appliquons à présent la transformation y_3 au programme de droite : par définition, cela nous génère une copie conforme du programme de droite. Nous obtenons donc bien deux séries de transformations qui, chacune appliquée à un des programmes initiaux, nous amènent à deux programmes égaux. Les programmes de gauche et de droite sont donc $\mathcal{Y}$-connectés.

Remarquons que cette application de transformations n'est pas la seule solution dans cet exemple. Nous aurions pu nous passer de la transformation identité et donc effectuer une série de transformations vide sur le programme de droite. Nous aurions également pu effectuer un nettoyage sur le programme de gauche, et un renommage de variables sur celui de droite. Il aurait aussi été possible de définir une nouvelle transformation qui insérerait une clause inaccessible depuis le point d'entrée du programme [4]. L'appliquer au programme de droite nous dispenserait d'effectuer un nettoyage sur celui de gauche.

Nous pouvons constater qu'une comparaison d'algorithmes comme celle de l'exemple 3.12 peut être plus ou moins complexe, en fonction des programmes considérés et des transformations admises dans l'ensemble $\mathcal{Y}$.

4. Une telle transformation est préservatrice de sem_1.

3.5 Ordre des clauses et des arguments

Comme nous l'avons déjà dit, l'ordre des clauses dans les programmes CLP n'a pas d'incidence sur sem_0. Or, dans le programme de l'exemple 3.12, un simple changement d'ordre des clauses bernerait notre technique de comparaison. Il est donc naturel d'introduire des transformations qui changent l'ordre des clauses et des arguments des prédicats dans notre cadre de travail.

Définition 3.16 (Transformation : changement d'ordre de clauses)**.** *Le changement d'ordre de clauses est une transformation de programme y telle que, pour tout programme P_1, $y(P_1) = P_2$ où P_2 est identique à P_1, à la seule différence près que l'ordre de ses clauses peut avoir été modifié. Cette transformation est notée* ordre_clauses.

Définition 3.17 (Transformation : changement d'ordre d'arguments)**.** *Le changement d'ordre d'arguments est une transformation de programme y telle que, pour tout programme P_1, $y(P_1) = P_1$ où P_2 est identique à P_1, à la seule différence près que l'ordre des arguments des têtes de ses clauses peut avoir été modifié. Pour chaque modification ainsi effectuée, la même permutation dans l'ordre des arguments est effectuée dans tous les appels vers les prédicats que ces clauses définissent. Cette transformation est notée* ordre_args.

3.6 Paramétrage et méta-paramétrage des transformations

3.6.1 Paramètres

Nous avons déjà rencontré des transformations qui pouvaient être paramétrées. Le renommage de variables, par exemple, existe en plusieurs déclinaisons : l'on peut vouloir effectuer le renommage sur certaines clauses en particulier, ou encore décider que le renommage doit générer des nouveaux noms de variables commençant par « B ». La plupart des transformations que nous allons rencontrer dans la suite présentent cette même particularité de n'être pas gravées dans le marbre.

Il pourrait, dans ce contexte, être intéressant de disposer d'un formalisme qui permette d'exprimer ces paramétrages. Jusqu'ici, nous avons écrit $renommage\_vars$, sans aucune précision. Nous avons délibérément laissé dans le flou les clauses à prendre en compte, ainsi que la fonction de renommage que sous-entend cette transformation.

La définition suivante permettra d'exprimer des choses telles que « renommage des variables commençant par X uniquement », ou « renommage des variables en des noms de variables commençant par X uniquement », par exemple.

Définition 3.18 (Paramétrage de transformations)**.** *Soit y une transformation de programmes logiques, et soient $u_1, u_2, ..., u_n$ ses n paramètres, avec comme valeurs par défaut $v_1, v_2, ..., v_n$ respectivement. Alors écrire $y(u_1 = w_1, u_2 = w_2, ..., u_n = w_n)$ revient à invoquer la transformation y avec comme valeurs de paramètres respectifs $w_1, ..., w_n$. Omettre l'une des assignations $u_i = w_i$ ($i \in 1..n$) revient à écrire $u_i = v_i$.*

Les paramètres se placent entre parenthèses après le nom de la transformation. Il ne faut pas confondre cette notation avec celle de l'application de la transformation à un programme. Dans la suite, lorsque l'on écrira $y(P_1)$, avec P_1 un programme, y sera une incarnation particulière de transformation avec des paramètres instanciés, non une transformation générale. Ainsi, y pourrait être l'incarnation de la transformation de renommage de variables suivante : $renommage\_vars(clauses = \{1\})$.

Le tableau 3.1 reprend les transformations élémentaires introduites jusqu'à présent qui disposent de paramètres.

Pour exprimer que les variables dont le nom commence par « A » doivent être renommées avec le même nom où le « A » initial est remplacé par « B » dans les deuxième et troisième clauses du programme, l'on notera [5] :

$$renommage\_vars(clauses = \{2, 3\}, f(var) = \textbf{if}(var == "A" + nom)\ \textbf{return}\ "B" + nom\ \textbf{else return}\ var)$$

3.6.2 Méta-paramètres

Les paramètres rencontrés jusqu'à présent sont instanciés lors de l'application des transformations, donnant naissance à des incarnations particulières de ces transformations. Par exemple, le renommage de prédicats (en toute généralité) peut faire partie des transformations admises, et c'est au moment de son application

5. Nous admettrons une grande liberté dans les notations de fonctions, pour peu que ces notations soient compréhensibles.

Tableau 3.1 – Transformations paramétrables du chapitre 3

Transformation	Paramètres	Type de valeur	Valeurs par défaut	Exemples d'utilisation
renommage_vars	*clauses*	Ensemble d'indices correspondant aux indices des clauses dont les variables doivent être renommées	∞ (= toutes les clauses)	*renommage_vars*($clauses = \{1\}$, $f = \{"X" \to "A", "Y" \to "B"\}$)
	f	Fonction : $X \to X$ où X est l'ensemble des noms de variables du domaine. Les variables dont le nom n'est pas précisé dans la définition de la fonction gardent leur nom.	Fonction associant chaque nom de variable apparaissant dans les clauses du premier paramètre à un nouveau nom de variable aléatoire	
renommage_preds	f	Fonction : $W \to W$ où W est l'ensemble des couples « nom de prédicat/arité de prédicat) ».	Fonction associant chaque nom de prédicat du programme à un nom de prédicat aléatoire	*renommage_preds*($f = \{p/3 \to q/3, g/3 \to p/3\}$)
nettoyage	p	Nom et arité du prédicat de référence à partir duquel nettoyer	(doit être fourni)	*nettoyage*($p = p_{start}$)
ordre_clauses	*perm*	Bijection : $1..n \Rightarrow 1..n$ (où n est le nombre de clauses du programme) de permutation déterminant, pour chaque identifiant de clause, sa nouvelle position.	Permutation aléatoire.	*ordre_clauses*($perm = \{2 \to 10, 4 \to -3\}$)
ordre_args	f	Fonction associant à chaque identifiant (nom/arité) de prédicat, une bijection de permutation $1..n \to 1..n$ (où n est le nombre d'arguments du prédicat).	Fonction aléatoire associant, à chaque prédicat, une bijection de permutation aléatoire.	*ordre_args*($f = \{p/3 \to \{1 \to 3, 2 \to 1, 3 \to 2\}, q/1 \to \{1 \to 1\}\}$)

Tableau 3.2 – Transformations méta-paramétrables du chapitre 3

Transformation	Méta-paramètre	Explication	Valeur par défaut
renommage_vars	*toutes_variables*	Si mis à *true*, force le paramètre f à — être une bijection ; — associer un nom de variable (éventuellement le même) à **chaque variable** présente dans les clauses du paramètre *clauses*.	*true*
renommage_preds	*tous_preds*	Si mis à *true*, force le paramètre f à — être une bijection ; — associer un nom de prédicat (éventuellement le même) à **chaque prédicat** du programme.	*true*
nettoyage	*pe*	Si mis à *true*, force le paramètre p à être le point d'entrée du programme	*true*

que n'importe quel prédicat pourra être renommé, avec n'importe quel nom [6]. En d'autres mots, le fait d'inclure la transformation de renommage de prédicats dans l'ensemble $\mathcal{Y}$ permet de déployer ses effets avec toutes valeurs de paramètres (c'est-à-dire que l'on peut renommer n'importe quel nombre de prédicats du programme).

Les paramètres n'ont donc pas d'effet avant l'exécution concrète de la transformation. Or certaines transformations, si elles étaient acceptées pour toutes valeurs de leurs paramètres, seraient trop générales pour être préservatrices des sémantiques présentées [7]. Par exemple, le nettoyage, qui consiste pour rappel à supprimer les prédicats inaccessibles depuis un prédicat de référence, a comme paramètre le prédicat de référence en question. Mais si ce prédicat n'est pas le point d'entrée du programme, la transformation de nettoyage pourrait supprimer des clauses pourtant essentielles pour l'algorithme capturé dans le programme. Cela peut, dans certaines applications, être intéressant. mais dans notre optique de recherche de transformations préservatrices de sem_1, la transformation de nettoyage risquerait d'être trop puissante si l'on lui laissait cette liberté d'être instanciée avec n'importe quel prédicat comme paramètre. Pour cette raison, nous introduisons les **méta-paramètres**.

Définition 3.19 (Méta-paramétrage des transformations)**.** *Une transformation y disposant de n méta-paramètres $m_1, m_2, ..., m_n$ est dite* méta-paramétrée *par les valeurs w_i ($i \in 1..n$) associées aux méta-paramètres m_i respectifs. On note alors $y[m_1 = w_1, m_2 = w_2, ..., m_n = w_n]$. Dans le cas où une assignation $m_j = w_j (j \in 1..n)$ est omise, le méta-paramètre m_j prendra sa valeur par défaut. Une transformation méta-paramétrée dont au moins une valeur de méta-paramètre est différente de la valeur par défaut, est dite* altérée*.*

Un méta-paramètre du nettoyage pourrait être le fait de forcer, ou non, le paramètre du nettoyage d'être un point d'entrée. Si nous forçons cela, alors le nettoyage sera une transformation préservatrice de sem_1. Un raisonnement similaire peut être conduit pour les autres transformations. Nous listons dans le tableau 3.2 les méta-paramètres des transformations introduites dans ce chapitre, et leurs valeurs par défaut.

Les transformations *id*, *ordre_clauses* et *ordre_args* ayant un comportement univoque, elles ne disposent pas de méta-paramètres.

6. Sauf si l'on interdit le renommage à générer des collisions de noms de prédicats dans le programme, c'est-à-dire si l'on force cette transformation à être préservatrice de sem_0 ; ce que nous ferons plus loin, lorsque nous aurons introduit les méta-paramètres.
7. Ou pour toute autre raison, dépendant de l'application.

Notons que l'on pourrait imaginer des méta-paramètres additionnels pour toutes les transformations, poussant plus loin encore leur configuration. Notre but étant ici de définir des transformations relativement génériques et préservatrices des sémantiques, et non de définir toutes les déclinaisons possibles de chaque transformation, nous nous contenterons des seuls méta-paramètres définis dans le tableau 3.2.

Nous avons intentionnellement défini des valeurs par défaut de méta-paramètres permettant à toutes les transformations introduites jusqu'à présent d'être préservatrices de sem_0, sauf le nettoyage qui n'est préservateur que de sem_1. Ainsi, le renommage de variables n'entraînera pas de doublons dans les noms de variables, puisqu'avec *toutes_variables* mis à true, f est une bijection définie sur toutes les variables des clauses considérées (l'apparition de doublons serait possible avec *toutes_variables* mis à $false$). De même pour *renommage_pred* : une fois la valeur *true* assignée au méta-paramètre, il est clair qu'aucun nom de prédicat ne servira pour deux prédicats différents du programme initial. Les transformations de renommage ont donc le comportement que l'on peut attendre d'elles : elles renomment, de façon inoffensive au niveau déclaratif, des éléments du programme.

Le résultat suivant découle de ces observations.

Proposition 3.20. *Les transformations* renommage_vars*,* renommage_preds*,* nettoyage*,* ordre_clauses*,* ordre_args *et* id *sont préservatrices de* sem_1*. De plus, toutes ces transformations sont préservatrices de* sem_0*, à l'exception de* nettoyage*.*

3.7 Transformations sur le corps des clauses

3.7.1 Remplacement de contraintes

Lorsque nous comparons des programmes logiques, l'expressivité des contraintes peut être un frein pour découvrir des programmes équivalents. En effet, considérons les deux clauses suivantes, les seules clauses composant respectivement deux programmes P_1 et P_2 écrits en $CLP(\mathbb{R}^+)$:

$$p_1(X) : -\{X < 10\}.$$
$$p_2(X) : -\{10 > X\}.$$

Clairement, les prédicats $p_1/1$ et $p_2/1$ vérifient la même propriété sur leur argument : celle d'être strictement inférieur à 10. Néanmoins, les transformations introduites jusqu'à présent ne permettront pas de conclure à l'équivalence des deux programmes, à cause de la forme différente de ces contraintes, alors qu'une telle conclusion serait naturellement souhaitée.

Nous avons, jusqu'ici, découvert des transformations jouant sur la structure des programmes logiques, notamment sur le nom de certains éléments ou leur ordre d'apparition. Nous n'avons pas encore traité des contraintes et de leur forme. Pour cela, nous allons définir ce que cela signifie pour deux contraintes d'être équivalentes. Cette notion demande d'abord l'introduction de deux autres définitions : l'ensemble de variables d'une contrainte, et le domaine d'une variable contrainte.

Un ensemble de variables d'une contrainte est, naturellement, l'ensemble des variables apparaissant dans cette contrainte :

Définition 3.21 (Ensemble de variables d'une contrainte)**.** *Soit C une contrainte. L'ensemble de variables de C, noté $Var(C)$, est l'ensemble constitué de chaque variable apparaissant dans la contrainte C.*

Quant au domaine d'une variable contrainte, il s'agit d'une restriction du domaine initial de cette variable (en $CLP(X)$, le domaine initial est X), cette restriction lui étant infligée par une ou plusieurs contrainte(s).

Définition 3.22 (Domaine d'une variable contrainte)**.** *Soit v une variable apparaissant dans une contrainte C exprimée sur le domaine de calcul $\mathcal{D}$. Alors $dom(v, C)$ est l'ensemble de valeurs de $\mathcal{D}$ que peut prendre la variable v une fois la contrainte C appliquée à $Var(C)$.*

Nous pouvons à présent définir l'équivalence de contraintes.

Définition 3.23 (Contraintes équivalentes)**.** *Deux contraintes C_1 et C_2 sont équivalentes si*
- $Var(C_1) = Var(C_2)$
- $\forall v \in Var(C_1) : dom(v, C_1) = dom(v, C_2)$

Les deux contraintes citées en début de section sont bien équivalentes car dans les deux cas, le domaine de la variable X, après application de la contrainte, est $[0, 10[$.

Dans la définition ci-dessus, nous exigeons que deux contraintes contiennent les exactes mêmes variables pour être équivalentes. Mais il est aisé de vérifier que les deux contraintes des clauses suivantes, quoique présentant des variables différentes, sont logiquement équivalentes :

$$p_1(X) : -\{Dix = 10, X < Dix\}.$$
$$p_2(X) : -\{X < 10\}.$$

Pour résoudre ce problème, il faut se placer à l'échelle des clauses et non plus des contraintes. Les variables qui ont de l'importance pour une clause sont en effet distinctes de celles qui composent ses contraintes.

Définition 3.24 (Ensemble de variables d'une clause)**.** *Soit $h(V_1, V_2, ..., V_n) : -\{C\}, B$ une clause notée c, telle que B est une séquence de k appels $b_k(W_1^1, W_2^1, ..., W_{a_1}^1)), ..., b_k(W_1^k, W_2^k, ..., W_{a_k}^k)$. Alors l'ensemble de variables de c, noté $Var(c)$, est l'ensemble $\{V_1, V_2, ..., V_n, W_1^1, W_2^1, ..., W_{a_1}^1, ..., W_1^k, W_2^k, ..., W_{a_k}^k\}$.*

Dans le contexte d'un programme CLP, deux contraintes seront réellement équivalentes si elles ne modifient pas le sens de la clause dans laquelle elles sont définies.

Définition 3.25 (Contraintes équivalentes dans une clause)**.** *Deux contraintes C_1 et C_2 sont équivalentes dans la clause c si $\forall v \in Var(c) : dom(v, C_1) = dom(v, C_2)$.*

Nous pouvons à présent introduire le remplacement de contrainte, une importante transformation pour la comparaison de clauses de Horn contraintes.

Définition 3.26 (Transformation : remplacement de contrainte)**.** *Le remplacement de contrainte est une transformation de programmes CLP y telle que, pour tout programme P_1, $y(P_1) = P_2$ où P_2 est identique au programme P_1, sauf éventuellement pour certaines clauses dont les contraintes peuvent avoir été remplacées par des contraintes équivalentes (dans la clause considérée, si le méta-paramètre* vars_clauses *est mis à true). Le remplacement de contrainte est noté* rempl_c.
Cette transformation est paramétrée par une fonction f *associant, à des identifiants de clauses, leurs nouvelles contraintes qui doivent être équivalentes. Il est méta-paramétré par* vars_clauses*. Si* vars_clauses *est mis à true (ce qui est le comportement par défaut), alors les contraintes devront être équivalentes* ***dans les clauses****.*

Il s'agit donc d'une réécriture des contraintes, telle que les nouvelles contraintes contraignent les mêmes variables de la même façon. Étant donné que la transformation présentée ci-dessus est inoffensive à tous les autres niveaux, le résultat suivant est trivial.

Proposition 3.27. *La transformation de remplacement de contraintes est préservatrice de sem_0, peu importe la valeur de son méta-paramètre vars_clauses.*

Tout comme les transformations de renommage et de changement d'ordre, le remplacement de contraintes nous semble essentiel pour la comparaison de programmes, étant donné que nous serions, sans cette transformation, rapidement « bernés » par une simple réécriture de contraintes d'un programme à l'autre. De plus, sans nous ouvrir à l'équivalence des contraintes *dans les clauses*, des variables sans impact sémantique sur les clauses, comme la variable Dix dans l'exemple ci-dessus, nous berneraient également.

Il nous serait utile de disposer d'une transformation qui simplifie les contraintes de façon clairement définie, et telle que la nouvelle contrainte générée soit sous une forme universelle facilement comparable syntaxiquement à toute autre contrainte. Une telle transformation permettrait de lever le non-déterminisme du remplacement de contraintes, et faciliterait la comparaison automatique de programmes, étant donné que les contraintes seraient réduites, une fois pour toutes, sous une forme simplifiée générique. Deux contraintes ayant une forme simplifiée différente ne pourraient alors être équivalentes. C'est une question abordée dans l'ouvrage [25], où sont discutés les « simplificateurs en forme canonique » (*canonical form simplifiers*). Ces simplificateurs sont des algorithmes qui permettent d'amener des contraintes dans une forme normale définie de façon déterministe. De tels processus ne sont pas triviaux, puisqu'il faut, entre autres, que les questions du nommage des variables, et d'ordre d'apparition de celles-ci, soient traitées. Ainsi, en $CLP(\mathbb{N})$, $\{X = Y\}$ et $\{Y = X\}$ sont des contraintes dont l'une devrait, idéalement, être simplifiée en l'autre ([25]). Il n'y a, à notre connaissance, pas d'algorithme existant permettant, pour n'importe quel domaine X, de simplifier les contraintes de $CLP(X)$ en forme canonique (ni de définition d'une forme canonique universelle). En revanche, il existe des algorithmes ponctuels qui effectuent une telle simplification pour des domaines de contraintes précis. Un exemple est le simplificateur en forme canonique de Gauss-Jordan pour les équations linéaires, c'est-à-dire une version de $CLP(\mathbb{R})$ où seules les conjonctions de contraintes d'égalité entre expressions sont admises comme contraintes. Cet algorithme est présenté dans [25]. Il utilise une relation d'ordre entre les noms de variables pour assurer une simplification et un ordonnancement des contraintes univoque. L'article [21] construit un algorithme de complexité polynomiale qui génère une forme canonique de contraintes linéaires, en présence d'inégalités et d'inéquations. L'intérêt d'un

simplificateur se manifeste en particulier lorsqu'il s'agit de comparer deux contraintes, et que le processus de simplification, couplé à une comparaison syntaxique, est moins coûteux que celui de comparaison dite « sémantique » entre les mêmes contraintes. Le fait que les simplificateurs existants se concentrent sur les contraintes d'équations linéaires n'est pas étonnant puisqu'il s'agit d'une des manifestations du paradigme CLP les plus populaires et pour laquelle des optimisations sont activement recherchées.

Un autre pas vers la normalisation de contraintes est réalisé dans [36], où il est question d'éliminer toutes les *variables existentielles* des contraintes à l'aide de la transformation de pliage [8]. Les variables existentielles d'une clause sont celles qui apparaissent dans son corps mais pas dans sa tête. La règle de transformation exposée vise à les supprimer pour pouvoir ensuite exprimer les contraintes en fonction des seules variables réellement importantes pour le programme. Il s'agit d'un cas particulier de remplacement de contraintes.

3.7.2 Agrégation de clauses

Observons une autre situation qui pourrait survenir au niveau du corps des clauses. Sémantiquement, nous le savons, le fait d'avoir deux (ou plus) clauses définissant un même prédicat cache en réalité une disjonction par rapport à la valeur de vérité de ce prédicat. En présence de contraintes, cette observation reste valide. Mais si, dans la plupart des cas, l'utilisation de différentes clauses pour définir un même prédicat est justifiée, dans d'autres, il s'agit d'une alternative syntaxique à l'utilisation d'une disjonction au sein des contraintes. Prenons les parties de programmes de l'exemple suivant :

Exemple 3.28.

```
p(X,Y) :- {X > Y}, q(X).
p(X,Y) :- {X < Y}.
```

```
p(X,Y) :- {X > Y}, q(X).
p(X,Y) :- {X < Y}, q(X).
```

Dans la portion de programme de gauche, les deux clauses du prédicat $p/2$ se distinguent dans leurs contraintes et dans leurs appels. Il serait faux, en toute généralité, de vouloir les agréger de la façon suivante :

$$p(X,Y) : -\{X > Y \vee X < Y\}, q(X).$$

puisque la deuxième clause ne demande pas que son premier argument X vérifie $q(X)$. De même, il serait faux de se débarrasser de l'appel à $q/1$ en les agrégeant de la façon suivante :

$$p(X,Y) : -\{X > Y \vee X < Y\}.$$

étant donné que l'atome $q(X)$ de la première clause serait alors perdu.

Il semble donc impossible d'agréger ces deux clauses : elles sont constitutrices d'un même prédicat (on a l'implication $(X > Y \wedge q(X)) \vee (X < Y) \Rightarrow p(X,Y)$), mais elles restent bien distinctes dans leur structure et il est nécessaire de les séparer, à cause de la présence de l'atome $q(X)$ dans une seule des deux clauses.

Les clauses de droite présentent la même structure : l'ordre et le nom des arguments est le même, et les appels sont identiques. La seule différence, dans ces clauses, se situe au niveau de leurs contraintes. En l'état, ces clauses incarnent l'implication $(X > Y \wedge q(X)) \vee (X < Y \wedge q(X)) \Rightarrow p(X,Y)$. Or cette implication, après simplification des formules logiques qui la composent, est équivalente à $(X > Y \vee X < Y) \wedge q(X) \Rightarrow p(X,Y)$. Cette même formule peut être capturée en une seule clause :

$$p(X,Y) : -\{X > Y \vee X < Y\}, q(X).$$

Définition 3.29 (Clauses agrégeables)**.** *Les clauses $c_1, c_2, ..., c_n$ ($n \geq 2$) sont dites agrégeables entre elles si*
- *elles définissent le même prédicat p/a ;*
- *elles présentent dans leur tête les mêmes arguments, dans le même ordre ;*
- *elles présentent les mêmes appels, dans le même ordre.*

Les deux premières conditions dans la définition ci-dessus signifient que les clauses doivent avoir la même tête ; la troisième condition signifie qu'elles doivent avoir le même corps, au niveau des appels. En d'autres mots, seules les contraintes peuvent varier.

Nos observations nous poussent à définir une nouvelle transformation, dont le but est de pouvoir agréger des clauses.

Définition 3.30 (Transformation : agrégation de clauses)**.** *L'agrégation de clauses est une transformation de programmes CLP y telle que, pour tout programme P_1, $y(P_1) = P_2$ où P_2 est identique à P_1 sauf que des clauses agrégeables entre elles peuvent avoir été agrégées entre elles. L'agrégation se fait, pour deux clauses* $\mathrm{h} : -\{\mathrm{C}_1\}, \mathrm{B}$

8. Nous présentons cette transformation dans le prochain chapitre.

et $\mathrm{h} :- \{\mathrm{C}_2\}, \mathrm{B}$*, en supprimant ces clauses et en les remplaçant par une nouvelle clause* $\mathrm{h} :- \{\mathrm{C}_1 \vee \mathrm{C}_2\}, \mathrm{B}$*. Pour n clauses, l'agrégation se fait en agrégeant la n-ième clause au résultat de l'agrégation des* $n-1$ *autres clauses. La transformation d'agrégation de clauses se note* agregation*.*
Cette transformation admet un paramètre : la fonction f *associant à chaque prédicat du programme un ensemble d'ensembles de clauses agrégeables entre elles (qui seront agrégées). Elle n'admet pas de méta-paramètre.*

Qu'arriverait-il si nous étions tentés, pour « faire correspondre » deux programmes, de plutôt *désagréger* des clauses ? Nous pourrions, à cet effet, définir une nouvelle transformation, qui aurait l'effet inverse de l'agrégation de clauses. Cependant, nous éviterons cela afin de garder un ensemble de transformations le plus réduit possible. En réalité, à tout moment où la « désagrégation » pourrait nous servir pour rapprocher un programme P_1 d'un programme P_2, effectuer une agrégation sur P_2 est une alternative équivalente. Nous ne perdons donc aucune puissance dans notre cadre de travail en n'y acceptant pas la désagrégation. Nous aurions toutefois pu choisir d'inclure uniquement la désagrégation, au lieu de l'agrégation, dans notre approche. Nous avons fait le choix de l'agrégation pour sa capacité à diminuer le nombre de clauses du programme, et donc la complexité d'autres transformations [9], là où la désagrégation augmente le nombre de clauses.

Nous avions, par ailleurs, déjà rencontré un cas où deux transformations se répondaient de façon réciproque (du moins dans certaines conditions). Il s'agit du nettoyage et de l'introduction de clauses, où nous avons fait le choix du nettoyage pour les mêmes raisons [10].

Le résultat suivant est intuitif et sera proposé sans preuve.

Proposition 3.31. *L'agrégation de clauses est une transformation préservatrice de* sem_0*.*

3.7.3 Changement d'ordre d'appels

Le corps des clauses est, pour rappel, séparé en deux parties : les contraintes, et les appels vers d'autres prédicats. Nous avons déjà traité du cas des contraintes. En ce qui concerne les appels, il nous faudrait simplement pouvoir les permuter.

Définition 3.32 (Transformation : changement d'ordre d'appels)**.** *Le changement d'ordre d'appels est une transformation de programmes logiques y telle que, pour tout programme* P_1*,* $y(P_1) = P_2$ *où* P_2 *est identique à* P_1*, à une permutation de l'ordre des appels de chaque clause près. Cette transformation est notée* ordre_appels*. Elle admet un paramètre :* f*, une fonction associant, à chaque clause du programme, une permutation (bijection* $1..n \to 1..n$ *avec n le nombre d'appels de la clause) qui représente les nouvelles positions des appels. Elle n'admet pas de méta-paramètre.*

Le changement d'ordre d'appels ne préserve pas la sémantique opérationnelle de Prolog. Il préserve toutefois l'aspect déclaratif des programmes, et donc sem_0.

Proposition 3.33. *Le changement d'ordre d'appels est une transformation préservatrice de* sem_0*.*

3.7.4 Remplacement d'arguments

Les appels de prédicats et les têtes de clauses étant des atomes, leurs arguments sont par définition des termes. Or les termes peuvent être de différentes formes. Il peut dès lors être utile, dans la comparaison de programmes, de disposer d'une transformation qui ramène les arguments sous une forme générique, et d'en transférer l'éventuelle complexité aux contraintes. Cette complexité pourra alors être gérée par le remplacement de contraintes.

Définition 3.34 (Remplacement d'argument dans sa clause)**.** *Soit c une clause, expr un argument apparaissant dans l'un de ses atomes (y compris sa tête), et C la contrainte de c. Alors remplacer expr dans sa clause par V revient à modifier c de façon à ce que :*
- *expr soit remplacé par* V*, avec* $V \notin Var(c)$ *;*
- C *soit remplacée par* $C \wedge (V = expr)$ *;*
- *tous les autres éléments de c soient inchangés.*

Définition 3.35 (Transformation : remplacement d'arguments)**.** *Le remplacement d'arguments est une transformation de programmes logiques avec contraintes y telle que pour tout programme* P_1*,* $y(P_1) = P_2$ *où* P_2 *est identique à* P_1*, à la seule différence que des arguments peuvent avoir été remplacés par des variables dans leurs clauses. Cette transformation est notée* rempl_args*.*

9. Dont le dépliage, introduit au chapitre suivant.
10. Cela dit, une variante de cette « introduction de clauses » nous sera utile dans le chapitre suivant pour supporter la transformation de pliage.

Elle admet un paramètre : f*, une fonction associant à des indices de clauses les arguments qui doivent être remplacés par des variables dans les clauses ciblées, et le nom des variables en question. Le remplacement d'arguments n'admet pas de méta-paramètre.*

Exemple 3.36. *Les deux programmes suivants, en présence du remplacement d'arguments, sont $\mathcal{Y}$-connectés. Appliquer $rempl\_args(f = \{1 \to \{(X+1) \to Y\}\})$ au programme de gauche génère en effet celui de droite.*

```
p(X) :- {}, q(X + 1).
q(X) :- {X > 10}.
```

```
p(X) :- {Y = X + 1}, q(Y).
q(X) :- {X > 10}.
```

Le remplacement d'arguments est la dernière transformation élémentaire introduite dans ce chapitre. Comme la plupart des autres transformations de ce chapitre, elle est préservatrice des sémantiques présentées.

Proposition 3.37. *Le remplacement d'arguments est une transformation préservatrice de sem_0.*

3.8 Algorithme naïf de comparaison

Nous disposons à présent d'assez d'éléments pour nous intéresser à la comparaison de programmes en tant que telle. Nous avons déjà défini à l'aide du théorème 3.11 une condition sous laquelle deux programmes calculent deux algorithmes connectés. Dans cette section, nous tentons de définir un algorithme permettant de déterminer de façon automatique si deux programmes remplissent cette condition. Il s'agit de l'algorithme 1. Nous y utilisons la notation $v \leftarrow choisir\_un\_au\_hasard(E)$ pour décrire l'assignation à la variable v d'un élément tiré au hasard dans l'ensemble E.

Algorithme 1 Algorithme naïf de comparaison

function ComparaisonNaive(P_1 : Programme, Q_1 : Programme, $\mathcal{Y}$: ensemble de transformations)
  $SP \leftarrow \{P_1\}$
  $SQ \leftarrow \{P_2\}$
  while $SP \bigcap SQ = \emptyset$ && Intéressant de continuer **do**
    $P \leftarrow choisir\_un\_au\_hasard(SP)$
    $Q \leftarrow choisir\_un\_au\_hasard(SQ)$
    $y_P \leftarrow choisir\_un\_au\_hasard(\mathcal{Y})$
    $y_Q \leftarrow choisir\_un\_au\_hasard(\mathcal{Y})$
    $P' \leftarrow y_P(P)$
    $Q' \leftarrow y_Q(Q)$
    $SP \leftarrow SP \bigcup \{P'\}$
    $SQ \leftarrow SP \bigcup \{Q'\}$
  if $SP \bigcap SQ \neq \emptyset$ **then return** true
  return $\perp$

Cet algorithme est simple et intuitif : il se contente de mettre à jour deux ensembles de programmes correspondant à des listes de versions transformées des deux programmes initiaux à comparer. Pour cela, l'algorithme va sélectionner des transformations de programmes au hasard, et les appliquer à un programme au hasard de chaque ensemble, jusqu'à ce que soit il ne soit plus intéressant de continuer, soit un programme commun ait été trouvé dans les deux ensembles. Dans le cas où un programme commun a été trouvé, les deux programmes sont $\mathcal{Y}$-connectés. Dans le cas où il n'est plus intéressant de continuer, on ne peut conclure : peut-être les deux programmes sont-ils en réalité $\mathcal{Y}$-connectés, mais l'algorithme naïf n'a pas pu conclure en ce sens. Cette indécision de l'algorithme est symbolisée par la valeur $\perp$.

L'algorithme naïf soulève de nombreuses questions. D'abord, il est évident qu'il n'est pas performant en présence de transformations non-déterministes. Par exemple, avec le renommage de variables, une infinité de versions transformées d'un même programme peut être générée, alors qu'un choix judicieux pour la fonction de renommage (comme celui que nous avons fait dans l'exemple 3.12) permettrait d'éviter ce genre d'explosion combinatoire. Il en va de même pour le renommage de prédicats, par exemple. Cette limite de l'algorithme naïf est encore plus flagrante en présence de la transformation identité : dans le pire des cas, cette transformation est choisie par hasard à chaque tour de boucle, et une infinité de programmes identiques est alors générée, ce qui ne permettra pas de comparer judicieusement deux programmes initiaux différents. Clairement, il serait intéressant de disposer d'une version plus intelligente de l'algorithme, où les transformations ne sont pas exécutées dans n'importe quel ordre, avec n'importe quels paramètres, ou sur n'importe quels programmes des deux séquences.

En fonction des transformations admises, des optimisations sont possibles, en particulier sur l'ordre des transformations à appliquer. Dans le chapitre suivant, nous tenterons d'identifier de telles optimisations. Nous pouvons déjà remarquer, avec les quelques transformations dont nous disposons pour l'instant, que certaines séries de transformations seraient inefficaces. Nous avons déjà cité l'inutilité de la transformation identité. Dans la même veine, il est inutile d'effectuer plusieurs fois la transformation de nettoyage, puisqu'après un premier nettoyage, plus aucun prédicat ne pourra être « nettoyé » étant donné que tous les prédicats inaccessibles depuis le point d'entrée auront déjà été supprimés du programme. Dans la même optique, le renommage n'est réellement utile qu'une fois par clause et par prédicat. Il est inutile de tenter tous les renommages possibles : seuls ceux qui amèneront une correspondance entre les prédicats et variables d'un programme vers l'autre (comme dans l'exemple 3.12) seront intéressants. De même, effectuer un renommage alors qu'il est encore possible de nettoyer des clauses ou de remplacer des arguments, semble sous-optimal. Le renommage est typiquement une transformation qui doit être appliquée en dernier lieu sur un programme, quand celui-ci est structurellement déjà très proche d'un des programmes de la séquence de programmes avec laquelle il sera comparé. Un raisonnement similaire peut être tenu pour d'autres transformations, comme $rempl\_c, ordre\_clauses, ordre\_args$ ou $ordre\_appels$.

Une autre question que soulève l'algorithme naïf est dans la condition d'arrêt de la boucle. À quel moment peut-on dire qu'il est « intéressant de continuer » ? Cette question dépasse le cadre de ce mémoire et ne sera plus adressée dans la suite, mais il est nécessaire de souligner qu'elle est très importante pour la définition d'un algorithme réalisant une comparaison algorithmique automatisée.

3.9 Exemples

Pour clôturer ce chapitre, nous présentons deux exemples mettant en lumière l'utilisation des transformations et concepts présentés.

3.9.1 Exemple 1

Nous introduisons deux programmes, P_1 et P_2, écrits en $CLP(\mathbb{N})$, dont les codes respectifs sont repris dans les portions de code 3.1 et 3.2.

Portion de code 3.1 – Programme P_1

```
facto(X, F) :- {X = 0, F = 1}.
facto(X, F) :- {X = 1, F = 1}.
facto(X, F) :- {X > 1, X2 = X - 1}, facto_acc(X, X2, F).
facto_acc(A, B, F):- {B > 1, A2 = A * B, B2 = B - 1}, facto_acc(A2, B2, F).
facto_acc(A, B, F):- {B = 1, F = A}.
```

Portion de code 3.2 – Programme P_2

```
f(X, F) :- {X =< 1, F = 1}.
f(X, F) :- {X > 1, X2 = X - 1}, f_2(X2, X, F).
f_2(Decompte, Acc, F):- {Decompte = 1, F = Acc}.
f_2(Decompte, Acc, F):- {Decompte > 1, NAcc = Acc * Decompte,
        NDecompte = Decompte - 1}, f_2(NDecompte, NAcc, F).
```

Nous allons nous poser la question : est-ce que, dans le programme P_1, le prédicat $facto/2$ (point d'entrée) réalise le même calcul que le prédicat $f/2$ (point d'entrée) du programme P_2 ?

Pour y répondre, nous allons nous munir de l'ensemble $\mathcal{Y}$ de transformations présentées dans ce chapitre, qui sont toutes, avec les valeurs par défaut de méta-paramètres, préservatrices de sem_1. Notre but va donc être de déterminer si P_1 et P_2 sont $\mathcal{Y}$-équivalents modulo la sémantique sem_1. Nous allons maintenant prouver, par applications successives des transformations, que P_1 et P_2 peuvent être transformés en un même programme tiers.

D'abord, nous pouvons remarquer une différence au niveau des points d'entrée : $facto/2$ et $f/2$ ne présentent pas le même nombre de clauses (3 et 2 respectivement). Nous pouvons être tentés d'agréger les deux premières clauses de P_1, qui sont justement agrégeables entre elles. Le résultat d'effectuer $agregation(f = \{facto/2 \rightarrow$

$\{\{1,2\}\}, facto\_acc \rightarrow \{\}\}$) sur P_1 est un nouveau programme identique à P_1 sauf que les deux premières clauses ont été agrégées. Ce programme est $agregation(P_1)$, représenté dans la portion de code 3.3.

Portion de code 3.3 – $agregation(P_1)$

```
facto(X, F) :- {X = 0 \/ X = 1, F = 1}.
facto(X, F) :- {X > 1, X2 = X - 1}, facto_acc(X, X2, F).
facto_acc(A, B, F):- {B > 1, A2 = A * B, B2 = B - 1}, facto_acc(A2, B2, F).
facto_acc(A, B, F):- {B = 1, F = A}.
```

La première clause de $agregation(P_1)$ diffère de la première clause de P_2 au niveau de l'expression de la contrainte, mais clairement, puisque nous travaillons en $CLP(\mathbb{N})$, les deux contraintes sont en réalité équivalentes dans le sens où elles restreignent le domaine de X et de F de la même façon. Un remplacement de contraintes effectué sur le programme $agregation(P_1)$, avec comme paramètre la fonction $\{1 \rightarrow \{X \leq 1, F = 1\}\}$, amène le programme $rempl\_c(P_1)$ présenté dans la portion de code 3.4.

Portion de code 3.4 – $rempl\_c(P_1)$

```
facto(X, F) :- {X =< 1, F = 1}.
facto(X, F) :- {X > 1, X2 = X - 1}, facto_acc(X, X2, F).
facto_acc(A, B, F):- {B > 1, A2 = A * B, B2 = B - 1}, facto_acc(A2, B2, F).
facto_acc(A, B, F):- {B = 1, F = A}.
```

À première vue, $facto/2$ devrait d'une façon ou d'une autre correspondre au point d'entrée $f/2$ de P_2. Dès lors, $facto\_acc/3$ devrait trouver son équivalent dans $f\_2/3$. Un renommage de prédicat du programme P_2 semble judicieux (nous aurions pu renommer $rempl\_c(P_1)$, mais l'expressivité des noms de prédicats y semble plus intéressante). Effectuer $renommage\_preds(f = \{f/2 \rightarrow facto/2, f\_2/3 \rightarrow facto\_acc/3\})$ sur P_2 génère le programme $r\_p(P_2)$ présenté dans la portion de code 3.5.

Portion de code 3.5 – $r\_p(P_2)$

```
facto(X, F) :- {X =< 1, F = 1}.
facto(X, F) :- {X > 1, X2 = X - 1}, facto_acc(X2, X, F).
facto_acc(Decompte, Acc, F):- {Decompte = 1, F = Acc}.
facto_acc(Decompte, Acc, F):- {Decompte > 1, NAcc = Acc * Decompte,
        NDecompte = Decompte - 1}, facto_acc(NDecompte, NAcc, F).
```

Les deux versions de $facto\_acc$ dont nous disposons ne semblent pas donner le même nom à leurs arguments, ni les présenter dans le même ordre. Une application à $rempl\_c(P_1)$ de $renommage\_vars(clauses = \{3,4\}, f = \{A \rightarrow Acc, B \rightarrow Decompte, F \rightarrow F, A2 \rightarrow NAcc, B2 \rightarrow NDecompte\})$ et à $r\_p(P_2)$ de $ordre\_args(f = \{facto/2 \rightarrow \{1 \rightarrow 1, 2 \rightarrow 2\}, facto\_acc/3 \rightarrow \{1 \rightarrow 2, 2 \rightarrow 1, 3 \rightarrow 3\}\})$ nous amène aux deux programmes $r\_v(P_1)$ et $o\_a(P_2)$ présentés respectivement dans les portions de code 3.6 et 3.7.

Portion de code 3.6 – $r\_v(P_1)$

```
facto(X, F) :- {X =< 1, F = 1}.
facto(X, F) :- {X > 1, X2 = X - 1}, facto_acc(X, X2, F).
facto_acc(Acc, Decompte, F):- {Decompte > 1, NAcc = Acc * Decompte,
        NDecompte = Decompte - 1}, facto_acc(NAcc, NDecompte, F).
facto_acc(Acc, Decompte, F):- {Decompte = 1, F = Acc}.
```

Portion de code 3.7 – $o\_a(P_2)$

```
facto(X, F) :- {X =< 1, F = 1}.
facto(X, F) :- {X > 1, X2 = X - 1}, facto_acc(X, X2, F).
facto_acc(Acc, Decompte, F):- {Decompte = 1, F = Acc}.
facto_acc(Acc, Decompte, F):- {Decompte > 1, NAcc = Acc * Decompte,
        NDecompte = Decompte - 1}, facto_acc(NAcc, NDecompte, F).
```

Un dernier obstacle semble séparer les deux programmes auxquels nous sommes arrivés : l'ordre des clauses de $facto\_acc/3$. Une simple application de la transformation $ordre\_clauses(perm = \{1 \to 1, 2 \to 2, 3 \to 4, 4 \to 3\})$ à $o\_a(P_2)$ génère le programme auquel la série de transformations appliquée à P_1 a mené, c'est-à-dire le programme de la portion de code 3.6.

En conclusion, P_1 et P_2 sont $\mathcal{Y}$-équivalents par rapport à sem_1 [11], étant donné qu'il existe deux séries de $\mathcal{Y}$-transformations préservatrices de sem_1 qui, chacune appliquée à l'un des programmes de départ, génèrent un même programme P_3 (ici, $r\_v(P_1)$).

Notons bien l'importance de chacune des transformations utilisées. Sans l'agrégation de clauses, par exemple, nous n'aurions pas pu arriver à la bonne conclusion. C'est aussi cela qui rend notre approche modulaire : si l'on estime, par exemple, que l'agrégation de clauses est une transformation trop puissante [12], et qu'elle ne devrait pas faire partie des transformations autorisées pour la comparaison de programmes, il suffit de la retirer de l'ensemble $\mathcal{Y}$, le reste du cadre de travail étant inchangé.

3.9.2 Exemple 2

À nouveau, nous allons introduire deux programmes, Q_1 et Q_2, et tenter de déterminer s'ils capturent l'essence d'un même algorithme. Le point d'entrée, dans l'un et l'autre programme, est le prédicat $fibo/1$.

Portion de code 3.8 – Programme Q_1

```
fibo(Xs) :- {}, fibo(Xs, 0, 1).
fibo([], N1, N2) :- {}.
fibo([X|Xs], N1, N2):- {X = N1 + N2}, fibo(Xs, N2, X).
```

Portion de code 3.9 – Programme Q_2

```
fibo(Xs) :- {}, fibo(Xs, 0, 1).
fibo([], N1, N2) :- {}.
fibo([X], N1, N2) :- {X = N1 + N2}.
fibo([X, Y|Xs], N1, N2):- {X = N1 + N2}, fibo([Y|Xs], N2, X).
```

Q_1 et Q_2 ne semblent pas présenter de différences au niveau de l'ordre des clauses, de leurs noms, du nom ou de l'ordre des arguments. Le deuxième programme présente une clause supplémentaire par rapport au premier, mais son prédicat $fibo/3$ ne présente aucune paire de clauses agrégeables : l'agrégation de clauses n'est donc pas une piste. Une autre transformation pourrait réduire le nombre de clauses : il s'agit du nettoyage. Mais dans notre cas, il ne modifierait aucun des deux programmes.

En nous limitant à notre ensemble $\mathcal{Y}$, il sera en réalité impossible de conclure à l'équivalence des deux programmes. Or, si l'on s'intéresse à leur sémantique, ils réalisent bien le même algorithme. En l'occurrence, dans les deux cas, le prédicat $fibo/1$ sera vrai si et seulement si son argument est une liste représentant une suite de Fibonacci [13]. Et dans les deux cas, ce prédicat fait appel à un prédicat $fibo/3$ pour réaliser le calcul d'une suite de Fibonacci de même taille que la liste en argument. Si l'on observe de plus près les clauses définissant $fibo/3$ dans le deuxième programme, nous pouvons remarquer qu'elles correspondent au cas où le premier argument est vide, constitué d'un seul élément, et constitué d'au moins deux éléments, respectivement. Dans Q_1, en revanche, le même prédicat est défini en deux clauses : l'une pour le cas où le premier argument est vide, et l'autre pour le cas où il est constitué d'**au moins un** élément. Cette dernière clause est donc une version plus générale que la dernière clause de Q_2, puisqu'à elle seule elle englobe le cas où la liste ne contient qu'un seul élément et le cas où elle en contient plus. Tout se passe comme si l'auteur du deuxième programme avait voulu modifier le premier programme en explicitant, d'une façon non nécessaire, un cas particulier de $fibo/3$.

Les algorithmes sont bien identiques, et nous aimerions donc conclure positivement quant à l'équivalence des programmes. Il nous faudrait une transformation capable de détecter ce genre de modifications de programme. Dans l'exemple précédent, nous étions bien équipés pour détecter des petites différences syntaxiques entre deux programmes. Mais face à ce deuxième exemple, avec les seules transformations élémentaires introduites dans

11. En l'occurrence, tous deux calculent une factorielle.
12. Ou toute autre raison qui, dans le contexte où la comparaison algorithmique est effectuée, pousserait à refuser une transformation.
13. Notre implémentation de cet algorithme est inspirée de [15].

notre cadre de travail, qui ne traitent que de problématiques de bas niveau, nous sommes impuissants.

Dans le chapitre suivant, nous découvrirons la clé pour résoudre ce problème : il s'agit des transformations de *pliage* et de *dépliage*. Nous découvrirons également d'autres situations concrètes où des transformations plus complexes seront nécessaires, et nous introduirons les transformations en question.

Chapitre 4

Transformations de programmes CLP avancées

Dans le deuxième exemple introduit en fin du chapitre précédent, les limites des seules transformations syntaxiques de base que nous avons déjà découvertes ont été mises en lumière. C'est ce qui nous motive, dans ce chapitre, à définir de nouvelles transformations, plus puissantes mais aussi répondant à des situations récurrentes dans la comparaison de programmes.

Dans un premier temps, nous listons ces situations problématiques afin de voir plus clairement de quels types de transformations nous aurons besoin. Ensuite, nous introduisons chacune de ces transformations. Le tout sera motivé par des exemples concrets où l'utilisation de ces transformations est nécessaire.

Dans ce chapitre, nous considérons que l'ensemble $\mathcal{Y}$ comprendra toujours toutes les transformations élémentaires définies dans le chapitre précédent. Pour rappel, toutes ces transformations sont préservatrices de sem_1. Vu que nous nous sommes déjà convaincus, dans le chapitre précédent, de l'utilité de ces transformations que nous avons illustrée à plus d'une reprise, nous nous permettrons, dans la suite, d'utiliser ces transformations moins verbeusement que précédemment. Ainsi, nous dirons « permuter les clauses c_i et c_j » plutôt que « appliquer la transformation $ordre\_clauses(perm = \{..., c_i \rightarrow c_j, ..., c_j \rightarrow c_i, ...\})$ ». Ces facilitations de notations nous permettront de nous concentrer sur les transformations plus avancées présentées dans ce chapitre.

Comme nous l'avons vu dans le chapitre précédent, l'utilisation des transformations rencontrées jusqu'à présent est relativement intuitive et simple : il est en fait aisé de la guider automatiquement, c'est-à-dire de réaliser de façon automatique ce que nous avons fait manuellement dans l'exemple du paragraphe 3.9.1. C'est la raison pour laquelle en général, nous devrons d'abord faire appel aux transformations plus puissantes présentées dans ce chapitre, puis, au moment de conclure si les deux programmes générés sont connectés, d'effectuer les opérations du chapitre précédent, de façon guidée et déterministe, afin d'essayer de faire correspondre les deux programmes. C'est aussi la façon dont fonctionne l'outil développé en parallèle des recherches du présent mémoire. Pour clôturer ce chapitre, nous détaillerons cette stratégie typique de transformations de programmes dans un but de comparaison et l'appliquerons sur un exemple concret.

4.1 Problématiques récurrentes

Dans cette section, nous essayons de cerner les cas typiques dans lesquels une comparaison algorithmique devrait déceler des similitudes entre deux programmes. Cela nous permettra de visualiser les transformations dont il pourrait être intéressant de disposer dans notre cadre de travail.

Avant toute chose, rappelons qu'un de nos buts est la détection de plagiat. Dès lors, il convient de se demander : « quels changements simples peuvent être effectués à un programme impératif dans le but de le plagier en passant inaperçu ? ».

Clairement, cette question peut amener de nombreuses réponses, en fonction de la complexité du plagiat en question. Nous nous contenterons ici d'identifier les cas les plus communs. Ces cas seront souvent exprimés dans des termes correspondant aux langages impératifs et non logiques, la traduction de l'un à l'autre pouvant être effectuée à l'aide d'un des outils décrits dans [20] et [12]. Nous discutons néanmoins les impacts techniques de chaque cas en termes de programmes CLP.

4.1.1 L'explicitation de calculs

Nous avons déjà rencontré un cas d'explicitation de calculs dans le paragraphe 3.9.2. Nous en présentons dans la portion de code 4.1 une variante en forme impérative, où des calculs déjà gérés par une instruction sont explicités. Nous parlons alors d'explicitation **redondante**.

Portion de code 4.1 – Fonction avec calculs explicités

```
function fibo(int[] xs) {
        int n1 = 0;
        int n2 = 1;

        for(x in xs) {
                if(x == dernierElement(xs)) {
                        if(x != n1 + n2) return false;
                }
                else {
                        if(x != n1 + n2) {
                                return false;
                                n1 = n2;
                                n2 = x;
                        }
                }
        }
        return true;
}
```

En effet, la première condition dans la boucle est inutile. En règle générale, ajouter des calculs déjà gérés par d'autres instructions est une façon intuitive de modifier discrètement le code d'un programme. Le nouveau programme semble plus long, plus complexe, et quoique le cas général soit le même, les nombreux cas particuliers peuvent semer une certaine confusion. Il serait intéressant de disposer d'une transformation qui permette, soit, à partir d'un cas général, d'extraire un cas particulier après l'autre, soit de faire l'opération inverse.

4.1.2 L'ajout d'intermédiaires

L'ajout d'intermédiaires est le fait d'encapsuler des calculs dans une partie de programme et d'en faire une nouvelle étape dans le programme. Voyons la différence entre les fonctions $f1$ et $f2$ dans la portion de code 4.2.

Portion de code 4.2 – Ajout d'intermédiaires

```
function f1() {
        int i = 0;
        int j = 2;
        int r = calc(i, j);

        return r;
}

function calc(int i, int j) {
        int a = i * j;
        return a + j;
}

function f2() {
        return 0 * 2 + 2;
}
```

En termes de programmes CLP, l'ajout d'intermédiaires engendre des programmes où un prédicat en appelle un autre, qui en appelle un autre, et ainsi de suite, sans que ces prédicats n'aient d'impact sémantique conséquent sur leurs arguments. En d'autres mots, on aura une succession de prédicats avec des contraintes de moindre importance qui s'appellent en chaîne plutôt que d'avoir un seul prédicat avec une contrainte plus conséquente. Il

nous serait utile de pouvoir créer une telle chaîne d'appels à partir d'un seul prédicat, ou à l'inverse de pouvoir simplifier une chaîne d'appels en l'encapsulant dans un seul prédicat.

4.1.3 Les calculs impossibles

Qu'en est-il si, pour distinguer son programme d'un autre, un auteur malveillant introduisait du code qui ne s'exécutera en pratique jamais, mais qui ajoute de la confusion syntaxique dans le programme ? Voyons un exemple dans la portion de code impératif 4.3 où le cœur algorithmique du programme est masqué derrière de fausses incertitudes quant à l'issue de celui-ci.

Portion de code 4.3 – Calculs impossibles

```
function f1(int x, int y) {
        int z1 = (x + y) mod 4;
        int z2 = (x * y) mod 4;

        bool cc = verifier_condition(x, y);

        // Toujours execute
        if(cc)
                return z1;
        // Jamais execute
        else
                return z2;
}

// Fonction qui renvoie toujours true
function verifier_condition(int a, int b) {
        if (a <= b && a > 0 && b > 0) {
                int c = a * b;
                if(c < 0) // Condition impossible
                        return false;
                else
                        return true;
        }
        else {
                return true;
        }
}
```

En termes de programmes CLP, l'on assistera à l'apparition de clauses présentant des contradictions dans leurs contraintes. Il serait de bon ton de pouvoir s'en débarrasser. Mais les contradictions peuvent aussi être « cachées » par des appels successifs à d'autres prédicats et seule une (simulation d') exécution du programme permettra de découvrir de telles contradictions.

4.1.4 L'ajout de variables

Une autre arme pour semer la confusion dans un code est d'y introduire des nouvelles variables, fondamentalement inutiles. Dans la portion de code 4.4, par exemple, l'on retrouve deux variables ajoutées n'ayant pas d'utilité sur le fonctionnement du programme.

Portion de code 4.4 – Ajout de variables

```
function variables_inutiles() {
        int h = 5;
        int j = h;
        int d = 2;
        int a = 0;

        if(a == 1 || a != 1) {
                for(int i = d; i > 0; i--) {
```

```
                        h += i;
                }
        }

        return h;
}
```

En termes de programmes $CLP(X)$, l'ajout de variables amène des clauses dont les arguments contiennent des variables en réalité inutiles au programme. Cela se matérialise par des variables qui soit ne sont jamais contraintes (dont le domaine est toujours égal à X) à travers le programme, soit sont contraintes mais sans que leur valeur n'ait d'impact sur les autres variables du programme. Ces variables n'ayant pas d'influence sur la sortie du programme, une transformation qui les élague serait la bienvenue.

4.1.5 La duplication de code

Dupliquer du code donne également une illusion de complexité à un programme. Voyons la portion de code 4.5, écrite en $CLP(\mathbb{N})$.

Portion de code 4.5 – Duplication de code

```
p(X, Y) :- {X = 1, Y = 2, X = 1}, q(X), q(X).
p(X, Y) :- {X = 1, Y = 2, X = 1}, q(X), q(X).
q(X).

r(X) :- {}, q(X).
r(X) :- {}, r(X).
```

Comme nous le voyons, en termes de programmes $CLP(X)$, la duplication peut apparaître à divers niveaux.

- D'abord au niveau des **clauses** (les deux premières clauses dans le code 4.5). Mais nous savons déjà comment gérer le cas de deux clauses identiques : il suffit de les agréger, puis éventuellement de remplacer la contrainte de la clause résultante de l'agrégation par une contrainte équivalente, pour pallier ce problème.
- Ensuite au niveau des **contraintes** (les contraintes des deux premières clauses dans le code 4.5). Là encore, nous disposons déjà d'une arme pour assainir une contrainte contenant une répétition inutile : le remplacement de contrainte.
- Enfin, au niveau des **appels** :
  - si un même appel est répété (comme les appels des deux premières clauses dans le code 4.5), il serait judicieux de pouvoir supprimer l'un des deux appels, puisque (dans sem_0), cela n'altérera pas le sens du programme logique ;
  - si une clause ne présente aucune contrainte et appelle le prédicat qu'elle définit elle-même (dernière clause dans le code 4.5), cela n'a pas d'impact sémantique sur le programme. Dans l'exemple, l'implication $r(X) \Rightarrow r(X)$ est une tautologie et n'a donc pas d'utilité dans notre base de connaissances. Elle devrait pouvoir être supprimée.

4.1.6 La réécriture et l'insertion de boucles

Dans le code 4.6 sont présentées trois boucles. Les deux premières sont une réécriture l'une de l'autre ; la troisième est une boucle n'ayant pas d'impact sur le comportement du programme.

Portion de code 4.6 – Réécriture et insertion de boucles

```
function boucle_terrible1() {
        int val = 0;

        int i = 0;
        while(i <= 5) {
                val += i;
                i++;
        }

        return val;
```

```
}

function boucle_terrible2() {
        int val = 0;

        for(int j = 1; j <= 6; j++) {
                val += (j - 1);
        }

        return val;
}

function boucle_terrible3() {
        int val = 0;

        for(int k = 0; k <= 4; k++) {
                k = k;
        }

        return val;
}
```

En termes de programmation logique, les deux premières fonctions donneront lieu à deux programmes distincts ayant le même comportement opérationnel extérieur. Mais les algorithmes sous-jacents peuvent raisonnablement être considérés comme identiques, et nous aimerions pouvoir conclure en ce sens. La troisième fonction se traduira par un code où le passage par un prédicat (celui représentant la boucle) ne changera rien au fonctionnement de programme. La boucle pourrait donc être retirée du programme sans effet sur celui-ci.

4.2 Dépliage

L'explicitation de calculs est un cas qu'il serait intéressant de pouvoir gérer dans notre cadre de travail. Une transformation qui répond à la situation est le **dépliage**.

Le dépliage a fait l'objet de nombreuses recherches. Sujet éprouvé par la communauté scientifique, il s'agit d'une transformation fréquemment citée lorsque l'on traite de transformations de programmes logiques avec contraintes. Cette section prend en particulier appui sur [3], [29] et [40] pour la définition du dépliage.

4.2.1 Dépliages simples et multiples

Avant d'exposer la transformation de dépliage, nous donnons le concept de clause renommée et de dépliage de clause.

Définition 4.1 (Clause renommée)**.** *Soit P un programme contenant une clause c. Alors la version renommée de c, notée $r(c)$, est identique à la clause c, à la seule différence que toutes les variables apparaissant dans c ont été renommées en suivant une fonction de renommage bijective telle qu'il n'existe aucun $v \in Var(c')$ dont le nom apparaisse ailleurs dans P.*

Une clause renommée dans un programme présente donc des noms de variables arbitraires qui doivent vérifier la propriété de n'être repris nulle part ailleurs dans le programme.

Définition 4.2 (Dépliage de clause à un appel)**.** *Soit P un programme CLP, et c une clause $h : -\{C\}, B.$ de ce programme, vérifiant la propriété que B ne contient qu'un seul appel, en l'occurrence l'appel $q(V_1, ..., V_a)$ vers le prédicat q/a. Soient $c_1, ..., c_n$ les n clauses définissant q/a et respectivement de la forme :*

$$q(W_1^i, ..., W_a^i) : -\{C_i\}, B_i.\ (i \in 1..n)$$

Alors déplier *c revient à remplacer c par n nouvelles clauses $h : -\{\tilde{C}_i\}, \tilde{B}_i.$ telles que $\forall i \in 1..n$:*

- $\tilde{B}_i = B'_i$
- $\tilde{C}_i = C \wedge C'_i \wedge C_i^{equ}$

où :

- $r(c_i) = q(W_1^{i'}, ..., W_a^{i'}) : -\{C'_i\}, B'_i.$
- $C_i^{equ} = \{V_1 = W_1^{i'}, ..., V_a = W_a^{i'}\}$

En d'autres mots, déplier une clause qui n'a qu'un seul appel revient à expliciter chacun de ses cas de figure, correspondant à chaque possibilité que permet son appel. Ainsi, le dépliage permet d'omettre l'étape du passage par la clause dépliée : celle-ci n'existera plus en tant que telle. Elle sera remplacée par le résultat du calcul de tous les aboutissements possibles de ses appels.

Pour un appel vers le prédicat q/a, la contrainte de chaque nouvelle clause générée est une conjonction de différentes contraintes :

— la contrainte de la clause dépliée avant qu'elle ne soit dépliée ;
— la contrainte de la clause de q/a actuellement explorée, après un renommage de ses variables qui évitera tout conflit dans les noms de variables ;
— des nouvelles contraintes qui conservent le lien entre les variables passées en argument au prédicat q/a lors de l'appel et le nom que ces arguments portent dans la clause explorée, après renommage des variables.

Exemple 4.3. *Considérons la portion de code 4.7. Le résultat d'y déplier la première clause est présenté dans la portion de code 4.8.*

Portion de code 4.7 – Programme de l'exemple 4.3 avant dépliage

```
p(X, Y) :- {X > 10}, q(Y).

q(X) :- {X > 5}.
q(X) :- {X < 5}, r(X).

r(X) :- {X > 0}.
```

Portion de code 4.8 – Programme de l'exemple 4.3 après dépliage

```
p(X, Y) :- {X > 10, Y = A, A > 5}.
p(X, Y) :- {X > 10, Y = A, A < 5}, r(A).

q(X) :- {X > 5}.
q(X) :- {X < 5}, r(X).

r(X) :- {X > 0}.
```

Nous pouvons remarquer que, dans l'exemple 4.3, après dépliage, le prédicat $q/1$ est inaccessible depuis $p/2$. L'étape que représentait $q/1$ dans le programme logique a été assimilée dans le prédicat $p/2$ déplié. D'ailleurs, déplier la deuxième clause du programme généré aurait le même impact, cette fois sur le prédicat $r/1$. Au final, l'entièreté du programme serait résumée en deux clauses, correspondant à l'explicitation des cas d'exécution possibles du programme initial. En réalité, c'est l'effet du dépliage : l'exécution d'une étape du programme CLP, au sens où nous l'avons décrit dans la section 2.5. En effet, une clause dépliée ajoute les contraintes des clauses « suivantes » aux siennes, tout comme le système CLP maintient un sac de contraintes. Couplé à une étape de suppression des clauses dont les contraintes sont inconsistantes [1], chaque dépliage exécute une étape du programme CLP plutôt que de simplement le transformer syntaxiquement.

Il est également possible de déplier une clause contenant un nombre indéfini d'appels. Clairement, si ce nombre est 0, alors aucun remplacement ne sera effectué. Si ce nombre est 1, alors il suffit d'appliquer la procédure exposée ci-dessus. Si le nombre est supérieur à 1, nous allons devoir définir une nouvelle stratégie. Celle-ci est présentée dans l'algorithme 2. Nous y utilisons les notations suivantes :

— (H, C, B) est un raccourci de notation équivalent à la clause $H :- \{C\}, B$.
— Une collection est un ensemble ordonné avec de potentiels doublons. Les éléments d'une collection sont notés entre crochets. En particulier, pour une collection Col et une collection E :
  — $Col.enlever(E)$ modifie l'état de Col en supprimant une occurrence de chaque élément de E en son sein. S'il y a plusieurs occurrences, elles sont supprimées dans l'ordre dans lequel elles apparaissent dans la collection.
  — $Col \backslash E$ renvoie une copie de Col, appelée Cop, sur laquelle l'opération $Cop.enlever(E)$ a été effectuée ;
  — $Col.ajouter(E)$ modifie l'état de Col en ajoutant une occurrence de chaque élément de E en son sein ;

1. Nous définissons une telle transformation dans la section suivante.

— $Col \bigcup E$ renvoie une copie de Col, appelée Cop, sur laquelle l'opération $Cop.ajouter(E)$ a été effectuée ;
— $for\ e \in Col$ permet d'itérer sur les éléments de Col dans l'ordre, l'élément courant étant capturé dans e.

— Dans une clause (H, C, B), soient $b_1(...), ..., b_n(...)$ les appels de B dans l'ordre dans lequel ils apparaissent dans la clause. Alors B est une collection $[b_1(...), ..., b_n(...)]$.
— Un programme présentant, dans l'ordre, les clauses $c_1, c_2, ..., c_m$ est représenté par une collection de clauses $[c_1, c_2, ..., c_m]$.
— $depliage\_un\_appel(Prog, Clause)$ est une fonction qui renvoie, sous la forme d'une collection, le résultat du dépliage de la clause à un seul appel $Clause$ au sein du programme $Prog$.

Algorithme 2 Dépliage de clause à plus d'un appel

```
procedure DEPLIAGEMULTIPLE(P : Programme, c ≡ (H1, C1, B1) : clause ∈ P)
    Appels_a_examiner ← B1
    Clauses ← [c]

    while Appels_a_examiner ≠ [ ] do

        for Appel ∈ Appels_a_examiner do
            ClausesADeplier ← Clauses

            for (H, C, B) ∈ ClausesADeplier do
                Clauses.enlever([(H, C, B)])
                NB ← B\[Appel]
                ClauseADeplier ← (H, C, [Appel])
                ClausesDepliees ← depliage_un_appel(P, ClauseADeplier)

                for (H2, C2, B2) ∈ ClausesDepliees do
                    B2.ajouter(NB)
                Clauses ← Clauses ⋃ ClausesDepliees
    P.enlever(c)
    P.ajouter(Clauses)
```

La stratégie de l'algorithme est de déplier les appels un par un pour nous ramener au cas du dépliage de clause à un seul appel. Les autres appels déjà présents dans la clause avant ce dépliage doivent ensuite lui être à nouveau accolés. Les appels en question sont auparavant stockés dans la variable NB. C'est ce qui est capturé dans la dernière boucle for, où NB est ajouté à chacune des clauses résultantes du dépliage de la clause initiale par rapport à l'appel examiné.

Intuitivement, dans ce cas de dépliage dit « multiple », on dépliera une clause c en listant toutes les possibilités auxquelles elle mène (tout comme pour le dépliage dit « simple »). Or la clause c fait potentiellement plusieurs appels vers d'autres prédicats. Chacun de ces appels, lors du dépliage, doit tenir compte de toutes les possibilités de ce qu'exige le prédicat (c'est-à-dire de n possibilités si le prédicat est défini par n clauses). C'est pourquoi au final, l'algorithme 2 génère $n_1 \times n_2 \times ... \times n_k$ nouvelles clauses, s'il y a k appels et que les prédicats appelés par ces appels sont respectivement définis par $n_1, n_2, ...,$ et n_k clauses.

Nous avons à présent toutes les cartes en main pour définir la transformation de dépliage.

Définition 4.4 (Transformation : dépliage)**.** *Le dépliage, noté* depliage*, est une transformation y telle que pour tout programme P_1, $y(P_1) = P_2$ où P_2 est identique à P_1, sauf que des clauses peuvent y avoir été dépliées. Les paramètre et méta-paramètre du dépliage sont repris dans le tableau 4.1.*

De nombreuses sources effectuent la preuve que le dépliage est une transformation préservatrice de la sémantique déclarative des programmes logiques. En particulier, l'article [3] démontre que c'est le cas pour la sémantique sem_0 (c'est-à-dire la sémantique algébrique définie dans [18]). Nous admettrons ce résultat dans le présent mémoire. Observons que de par sa nature purement déclarative, sem_0 est également préservée lorsqu'une partie seulement des appels d'une clause sont dépliés.

Tableau 4.1 – Paramètres et méta-paramètres du dépliage

(Méta-)paramètre	Type	Explication	Valeur par défaut
clause	Paramètre	Indice de la clause devant être dépliée (sous forme d'entier)	1
limite	Méta-paramètre	Entier définissant le nombre maximal de dépliages autorisés. Une fois ce nombre atteint, la transformation de dépliage ne peut plus être utilisée.	∞

4.2.2 Réponse à l'explicitation de calculs et à l'ajout d'intermédiaires

Voyons à présent un exemple pratique où le dépliage est nécessaire pour conclure avec justesse quant à l'équivalence de deux programmes. Reprenons pour cela l'exemple du paragraphe 3.9.2. Si nous effectuons $depliage(clause = 3)$ sur Q_1, nous arrivons au programme Q_2, à un remplacement d'arguments, un remplacement de contraintes et un renommage de variables près. En effet, déplier la clause

$$fibo([X|Xs], N1, N2) : -\{X = N1 + N2\}, fibo(Xs, N2, X). \tag{4.1}$$

génère les clauses :

$$fibo([X|Xs], N1, N2) : -\{X = N1 + N2, Xs = [\,], N2 = A, X = B\}. \tag{4.2}$$

et

$$fibo([X|Xs], N1, N2) : -\{X = N1 + N2, Xs = [L|Ls], N2 = A, X = B, L = A + B\}, fibo(Ls, B, L). \tag{4.3}$$

ce qui correspond bien à l'explicitation des deux cas possibles auxquels amène la clause initiale. Soit la liste en argument ne contient qu'un élément, soit elle en contient au moins deux.

Les arguments de certaines clauses des deux programmes contiennent des termes plus complexes que des simples variables. Un remplacement d'arguments sur les deux programmes permettra de les aplanir à ce niveau. Il nous suffit alors d'effectuer un remplacement de contraintes et un renommage de variables judicieux sur un des deux programmes pour le transformer en l'autre. Ils sont donc bien $\mathcal{Y}$-connectés [2].

En ce qui concerne l'explicitation de calculs faite de façon redondante (c'est-à-dire où la véracité d'une clause est impliquée par celle d'une autre clause définissant le même prédicat), elle peut également être gérée par le dépliage. Pour nous en convaincre, nous allons utiliser le concept d'équivalence sémantique entre groupes de clauses.

Définition 4.5 (Groupes de clauses équivalents sémantiquement)**.** *Soit P un programme, $\mathcal{Y}$ un ensemble de transformations, sem une fonction sémantique orientée point d'entrée, et C_1 et C_2 deux groupes de clauses de P, tels que toutes les clauses de C_1 et de C_2 définissent un même prédicat p/a accessible depuis le point d'entrée du programme. Soit P_1 la version de P contenant C_1 et ne contenant pas C_2. Soit P_2 la version de P contenant C_2 et ne contenant pas C_1. Alors C_1 et C_2 sont des groupes de clauses équivalents sémantiquement par rapport à sem si et seulement si P_1 est $\mathcal{Y}$-équivalent à P_2 par rapport à sem.*

En particulier, si C_1 n'est constitué que d'une clause c_1 et C_2 d'une clause c_2, on dira que c_1 et c_2 sont sémantiquement équivalentes par rapport à sem.

Par définition, la clause 4.1 est sémantiquement équivalente (par rapport à sem_1 et aux transformations définies jusqu'ici) aux deux qui résultent de son dépliage. Dès lors, nous pouvons « ajouter » la clause 4.1 aux deux autres clauses sans modifier le sens du programme résultat. Avec le même raisonnement, il apparait que nous pouvons retirer la clause 4.3 du programme sans en modifier le sens, puisqu'il est déjà entièrement capturé dans la clause 4.1. Au final, nous obtenons les deux clauses suivantes :

$$fibo([X|Xs], N1, N2) : -\{X = N1 + N2, Xs = [\,], N2 = A, X = B\}. \text{ (clause 4.2)}$$
$$fibo([X|Xs], N1, N2) : -\{X = N1 + N2\}, fibo(Xs, N2, X). \text{ (clause 4.1)}$$

2. Nous n'avons pas retranscrit ici en détail l'application de ces transformations, l'exemple ayant pour seul but d'illustrer le rôle du dépliage dans l'explicitation de calculs.

sémantiquement équivalentes à la seule clause 4.1, ou au groupement des deux clauses 4.2 et 4.3.

Concrètement, le dépliage nous a ici permis d'expliciter les deux cas possibles du prédicat $fibo/3$. S'il le fait sans redondance dans le corps des clauses générées, le fait de jouer sur l'équivalence sémantique des clauses nous autorise à nous retrouver avec des cas redondants explicités. En présence de la clause 4.1, la clause 4.2 pourrait en effet être supprimée du programme sans impact sur son sens, puisqu'elle est redondante par rapport à la clause 4.1.

Le cas d'ajout d'intermédiaires exposé en début de chapitre peut également être géré par le dépliage. Le fait d'expliciter des calculs supprime les appels intermédiaires, qui ne sont jamais qu'une anti-explicitation. Ces deux problématiques sont duales et toutes deux sont résolues par l'utilisation du dépliage, qui amènera les clauses à un même niveau d'explicitation.

4.2.3 Dépliages successifs

Reprenons l'exemple précédent. Nous avons déjà vu que déplier une fois la clause définissant le cas général du prédicat $fibo/3$ nous amène à générer deux cas. Le premier cas est celui où la liste en argument ne contient qu'un seul élément (résultat du dépliage de l'appel à $fibo/3$ par rapport à sa première clause). Le second cas est celui où la liste en argument contient plus d'un élément (résultat du dépliage de l'appel à $fibo/3$ par rapport à sa deuxième clause, c'est-à-dire la clause à déplier elle-même). Le programme, à ce stade, est celui présenté dans la portion de code 3.9.

À présent, essayons de poursuivre le dépliage de $fibo/3$. Des trois clauses définissant ce prédicat, une seule contient encore un appel, et peut donc être dépliée. Déplions-la ; nous obtenons, après réécriture des contraintes et arguments, la portion de code 4.9 :

Portion de code 4.9 – Programme Q_2 après un dépliage de $fibo/3$

```
fibo(Xs) :- {}, fibo(Xs, 0, 1).
fibo([], N1, N2) :- {}.
fibo([X], N1, N2) :- {X = N1 + N2}.
fibo([X, Y], N1, N2) :- {X = N1 + N2, Y = N2 + X}.
fibo([X, Y, Z|Xs], N1, N2) :- {X = N1 + N2, Y = X + N2}, fibo([Z|Xs], X, Y).
```

À nouveau, seule la dernière clause contient un appel. Déplions-la encore, pour obtenir la portion de code 4.10.

Portion de code 4.10 – Programme Q_2 après deux dépliages de $fibo/3$

```
fibo(Xs) :- {}, fibo(Xs, 0, 1).
fibo([], N1, N2) :- {}.
fibo([X], N1, N2) :- {X = N1 + N2}.
fibo([X, Y], N1, N2) :- {X = N1 + N2, Y = N2 + X}.
fibo([X, Y, Z], N1, N2) :- {X = N1 + N2, Y = N2 + X, Z = X + Y}.
fibo([X, Y, Z, W], N1, N2) :- {X = N1 + N2, Y = N2 + X, Z = X + Y, W = Y + Z}.
fibo([X, Y, Z, W, V|Xs], N1, N2) :- {X = N1 + N2, Y = N2 + X, Z = X + Y,
        W = Y + Z}, fibo([V|Xs], Z, W).
```

Le nombre de cas générés croît avec les dépliages. Ce qui est logique, car un dépliage cherche à expliciter tous les cas des appels de la clause à déplier. Or l'appel étant, ici, récursif, le nombre de clauses définissant le prédicat $fibo/3$ grandit à chaque itération. Notre premier dépliage a généré 2 nouvelles clauses. Ensuite, nous en avons généré 3. Si nous itérions à nouveau, 5 nouvelles clauses viendraient remplacer la dernière. Après un nouveau dépliage, 9 nouvelles clauses seront générées. Ensuite, il y en aurait 17. En règle générale, un tel dépliage, sur un prédicat possédant un cas récursif et un cas de base, génère $2^n + 1$ clauses, où n est le nombre de dépliages effectués jusqu'à présent.

Clairement, dans notre cas, ce processus est potentiellement infini, puisque le cas général du prédicat $fibo/3$ n'est pas borné. Nous avons déjà signalé que le dépliage avait pour effet d'exécuter le programme CLP. Ici, il ne s'agit plus réellement d'exécution, mais plutôt d'explicitation. Pour que le programme soit réellement exécuté par les dépliages, il faudrait que nous disposions d'une liste instanciée, ayant donc une longueur bornée. Néanmoins, le fait de déplier le programme de cette façon un certain nombre de fois permettra typiquement

au programme résultat de répondre plus rapidement aux requêtes portant sur un cas de calcul déjà explicité [3]. Nous revenons sur ce point dans le chapitre suivant.

Dans un cas non-infini, il peut être intéressant de déplier une clause, voire un point d'entrée, « au maximum ». Nous illustrons ce concept sur le programme S de la portion de code 4.11. Une fois traduit en clauses logiques par l'outil décrit dans [43] (le résultat est visible dans la portion de code 4.12), ce programme est transformé par des dépliages successifs de $p\_start/1$ en la portion de code 4.13. Notons que, par souci de lisibilité, nous avons effectué un renommage de variables et un remplacement de contraintes dans cette portion de code. Sans le remplacement de contraintes, de nombreuses contraintes générées lors du processus de dépliage (typiquement des égalités entre les arguments effectifs et les nouveaux noms des arguments formels correspondants) seraient présentes.

Portion de code 4.11 – Programme S

```
int max(int x, int y, int z) {
        int max_xy;
        if (x>=y)
                max_xy = x;
        else
                max_xy = y;
        if (max_xy>=z)
                return max_xy;
        else
                return z;
}
```

Portion de code 4.12 – Programme S traduit en CLP par Rundroid[43]

```
% Initialisation des registres (variables locales)
p_start(IV0,IV1,IV2,IV3,IV4,IV5,IM,OM,R) :- {OV0 = 0, OV1 = IV1, OV2 = IV2,
        OV3 = IV3, OV4 = IV4, OV5 = IV5}, p2(OV0,OV1,OV2,OV3,OV4,OV5,IM,OM,R).

p2(IV0,IV1,IV2,IV3,IV4,IV5,IM,OM,R) :- {OV1 = 0, OV0 = IV0, OV2 = IV2,
        OV3 = IV3, OV4 = IV4, OV5 = IV5}, p3(OV0,OV1,OV2,OV3,OV4,OV5,IM,OM,R).

%% Premier "if"
% Si x < y on saute vers le premier "else"
p3(IV0,IV1,IV2,IV3,IV4,IV5,IM,OM,R) :- {IV3 < IV4, OV0 = IV0, OV1 = IV1,
        OV2 = IV2, OV3 = IV3, OV4 = IV4, OV5 = IV5},
        p9(OV0,OV1,OV2,OV3,OV4,OV5,IM,OM,R).

% Sinon, si x >= y on rentre dans le premier "if"
p3(IV0,IV1,IV2,IV3,IV4,IV5,IM,OM,R) :- {IV3 >= IV4, OV0 = IV0, OV1 = IV1,
        OV2 = IV2, OV3 = IV3, OV4 = IV4, OV5 = IV5},
        p4(OV0,OV1,OV2,OV3,OV4,OV5,IM,OM,R).
% max_xy := x
p4(IV0,IV1,IV2,IV3,IV4,IV5,IM,OM,R) :- {OV1 = IV3, OV0 = IV0, OV2 = IV2,
        OV3 = IV3, OV4 = IV4, OV5 = IV5}, p5(OV0,OV1,OV2,OV3,OV4,OV5,IM,OM,R).

%% Deuxième "if"
% Si max_xy < z on saute vers le deuxième "else"
p5(IV0,IV1,IV2,IV3,IV4,IV5,IM,OM,R) :- {IV1 < IV5, OV0 = IV0, OV1 = IV1,
        OV2 = IV2, OV3 = IV3, OV4 = IV4, OV5 = IV5},
        p11(OV0,OV1,OV2,OV3,OV4,OV5,IM,OM,R).

% Sinon si max_xy >= z on rentre dans le "if"
p5(IV0,IV1,IV2,IV3,IV4,IV5,IM,OM,R) :- {IV1 >= IV5, OV0 = IV0, OV1 = IV1,
        OV2 = IV2, OV3 = IV3, OV4 = IV4, OV5 = IV5},
        p7(OV0,OV1,OV2,OV3,OV4,OV5,IM,OM,R).
```

3. C'est, du moins, le cas pour les moteurs d'inférence tels que celui de Prolog.

```
% return max_xy
p7(IV0,IV1,IV2,IV3,IV4,IV5,IM,OM,R) :- {OV0 = IV1, OV1 = IV1, OV2 = IV2,
        OV3 = IV3, OV4 = IV4, OV5 = IV5}, p8(OV0,OV1,OV2,OV3,OV4,OV5,IM,OM,R).

%% Stocke dans le dernier argument (valeur de retour) la valeur du premier argument
p8(IV0,IV1,IV2,IV3,IV4,IV5,IM,IM,IV0) :- {}.

%% Premier "else"
% max_xy := y
p9(IV0,IV1,IV2,IV3,IV4,IV5,IM,OM,R) :- {OV1 = IV4, OV0 = IV0, OV2 = IV2,
        OV3 = IV3, OV4 = IV4, OV5 = IV5}, p10(OV0,OV1,OV2,OV3,OV4,OV5,IM,OM,R).

% Retour au deuxième "if"
p10(IV0,IV1,IV2,IV3,IV4,IV5,IM,OM,R) :- {OV0 = IV0, OV1 = IV1, OV2 = IV2,
        OV3 = IV3, OV4 = IV4, OV5 = IV5}, p5(OV0,OV1,OV2,OV3,OV4,OV5,IM,OM,R).

% return z
p11(IV0,IV1,IV2,IV3,IV4,IV5,IM,OM,R) :- {OV0 = IV5, OV1 = IV1, OV2 = IV2,
        OV3 = IV3, OV4 = IV4, OV5 = IV5}, p12(OV0,OV1,OV2,OV3,OV4,OV5,IM,OM,R).

p12(IV0,IV1,IV2,IV3,IV4,IV5,IM,OM,R) :- {OV0 = IV0, OV1 = IV1, OV2 = IV2,
        OV3 = IV3, OV4 = IV4, OV5 = IV5}, p8(OV0,OV1,OV2,OV3,OV4,OV5,IM,OM,R).
```

Portion de code 4.13 – Programme S après dépliage de $p\_start/9$

```
p_start(A,B,C,D,E,F,G,H,I):- {D>=F, D>=E, D=I, G=H}.
p_start(A,B,C,D,E,F,G,H,I):- {D<F, D>=E, F=I, G=H}.
p_start(A,B,C,D,E,F,G,H,I):- {D<E, E>=F, E=I, G=H}.
p_start(A,B,C,D,E,F,G,H,I):- {D<E, E<F, F=I, G=H}.
```

Si le programme de la portion de code 4.12 semble illisible, c'est parce qu'il est généré automatiquement. De fait, le nombre de variables qu'il présente est inutilement élevé pour calculer un maximum entre trois valeurs. Plusieurs variables ne voient d'ailleurs jamais leur domaine contraint au fil du programme. Cette limitation ne peut être réglée par le dépliage, et sera discutée dans la section 4.5.

Il n'en reste pas moins que le programme étudié calcule le maximum entre les variables $IV3$, $IV4$ et $IV5$. Le résultat de ce calcul est la variable R. Quant aux variables IM et OM, elles représentent la mémoire avant et après l'exécution du programme ; une particularité du logiciel de traduction présenté dans [43]. Après dépliages successifs de $p1/9$, nous obtenons quatre clauses seulement, sur lesquelles nous effectuons un renommage de variables. Toutes présentent la contrainte $G = H$ (correspondant à $IM = OM$ dans le programme initial). En effet, la mémoire est, dans cet exemple, inchangée dans tous les cas. Les autres contraintes permettent de scinder les cas du calcul : dans la première clause, c'est la variable D (correspondant à $IV3$) qui est le maximum. Dans la seconde et la dernière, c'est F (correspondant à $IV5$). Enfin, dans la troisième clause, c'est E ($IV4$) le maximum.

Comme nous pouvons le voir, le programme est réellement exécuté au maximum, et les cas résiduels correspondent aux différents points de choix qui apparaissent dans le programme : les disjonctions marquées par des clauses différentes pour un même prédicat. Le programme CLP initial était long et difficile à lire ; le dépliage l'a raccourci et, couplé à un remplacement de contraintes, il en a facilité la compréhension.

En réalité, l'exemple se prêtait bien à la situation, mais il y a un cas dans lequel un tel dépliage « complet » est rendu impossible. Dès lors qu'un cycle apparaît pour un appel vers un certain prédicat p_1/a_1 à déplier, c'est-à-dire un chemin d'appels $p_1/a_1 \rightarrow p_2/a_2 \rightarrow ... \rightarrow p_1/a_1$, le dépliage « au maximum » ne pourra se terminer, à cause des appels cycliques (ou, plus simplement, récursifs) qui reviendront à l'infini dans les différents dépliages [4]. Il est en revanche possible d'exécuter un dépliage au maximum, jusqu'à ce qu'un appel vers un prédicat tel que p_1/a_1 soit rencontré. En effectuant ce procédé sur tous les prédicats du programme, le programme sera déplié autant que possible, et présentera comme seuls appels résiduels ceux qui mènent à des potentiels

4. À moins, évidemment, que nous ne disposions d'une transformation permettant d'éliminer les clauses dont les contraintes sont inconsistantes, et que cette transformation soit suffisante pour assurer que le dépliage se termine.

cycles. Un tel programme sera dit *complètement déplié*.

En résumé, le dépliage est une transformation très puissante, dans sa capacité de simulation d'exécution des programmes logiques. Elle permet l'explicitation de calculs, et la suppression d'intermédiaires, redondants ou non. Cette transformation peut, dans certains cas, être exécutée successivement au maximum sur les points d'entrée des programmes à comparer, afin de réduire ces programmes à leur version la plus explicitée et donc d'annihiler toute différence du niveau d'explicitation des programmes.

4.3 Suppression de clauses inconsistantes

Nous avons vu que des cas intentionnels d'introduction d'inconsistances dans un programme pouvaient survenir à travers l'introduction de calculs impossibles. Des erreurs de programmation, ou certaines transformations de programmes, peuvent également avoir cette conséquence. Si jusqu'ici nos contraintes ne présentaient aucune contradiction, nous devrions pouvoir prendre en compte le cas où de telles contradictions apparaissent, intentionnellement ou non.

Prenons par exemple le programme S présenté dans la section précédente, pour lequel le dépliage complet a permis d'expliciter tous les cas d'exécution possibles de façon générale, puisque le programme lui-même calculait le maximum entre trois nombres de façon générale. Ce n'est pas toujours le cas : des variables pourraient se voir attribuer une valeur dans le programme. Il est alors probable que des contraintes, prévues pour un cas général, ne soient tout simplement pas consistantes dans le cas particulier. Pour illustrer cela, reprenons la portion de code 4.13, et ajoutons un nouveau prédicat qui sera le nouveau point d'entrée du programme. Le résultat est visible dans la portion de code 4.14.

Portion de code 4.14 – Programme S après dépliage complet et nouveau point d'entrée

```
p_start(Max) :- {A1=0, B1=0, C1=0, G1=0, H1=0, D1=5, E1=4, F1=10},
        p_max(A1,B1,C1,D1,E1,F1,G1,H1,Max).

p_max(A,B,C,D,E,F,G,H,I):- {D>=F, D>=E, D=I, G=H}.
p_max(A,B,C,D,E,F,G,H,I):- {D<F, D>=E, F=I, G=H}.
p_max(A,B,C,D,E,F,G,H,I):- {D<E, E>=F, E=I, G=H}.
p_max(A,B,C,D,E,F,G,H,I):- {D<E, E<F, F=I, G=H}.
```

Le nouveau point d'entrée ($p\_start/1$) initialise les valeurs à comparer : D se voit attribuer la valeur 5, E la valeur 4 et F la valeur 10. Le programme est supposé calculer le maximum de ces trois nombres et placer la valeur résultat dans Max. Déplions la nouvelle clause et nous obtenons la portion de code 4.15.

Portion de code 4.15 – Programme S après dépliage complet et nouveau point d'entrée et nouveau dépliage

```
% Dépliage par rapport à la première clause de p_max
p_start(Max) :- {A1=0, B1=0, C1=0, G1=0, H1=0, D1=5, E1=4, F1=10,
        A1=A, B1=B, C1=C, D1=D, E1=E, F1=F, G1=G, H1=H, Max=I,
        D>=F, D>=E, D=I, G=H}.

% Dépliage par rapport à la deuxième clause de p_max
p_start(Max) :- {A1=0, B1=0, C1=0, G1=0, H1=0, D1=5, E1=4, F1=10,
        A1=A, B1=B, C1=C, D1=D, E1=E, F1=F, G1=G, H1=H, Max=I,
        D<F, D>=E, F=I, G=H}.

% Dépliage par rapport à la troisième clause de p_max
p_start(Max) :- {A1=0, B1=0, C1=0, G1=0, H1=0, D1=5, E1=4, F1=10,
        A1=A, B1=B, C1=C, D1=D, E1=E, F1=F, G1=G, H1=H, Max=I,
        D<E, E>=F, E=I, G=H}.

% Dépliage par rapport à la quatrième clause de p_max
p_start(Max) :- {A1=0, B1=0, C1=0, G1=0, H1=0, D1=5, E1=4, F1=10,
        A1=A, B1=B, C1=C, D1=D, E1=E, F1=F, G1=G, H1=H, Max=I,
        D<E, E<F, F=I, G=H}.
```

```
p_max(A,B,C,D,E,F,G,H,I):- {D>=F, D>=E, D=I, G=H}.
p_max(A,B,C,D,E,F,G,H,I):- {D<F, D>=E, F=I, G=H}.
p_max(A,B,C,D,E,F,G,H,I):- {D<E, E>=F, E=I, G=H}.
p_max(A,B,C,D,E,F,G,H,I):- {D<E, E<F, F=I, G=H}.
```

À présent, les valeurs à comparer ayant été instanciées, nous devrions nous attendre à ce qu'une seule des trois valeurs soit le maximum. Or le programme, tel qu'il est, semble séparer les possibilités en quatre cas, et dans ces quatre cas, les trois maxima possibles sont encore envisagés. En y regardant de plus près, nous pouvons remarquer que les contraintes de trois des quatre clauses sont en fait **inconsistantes**.

Définition 4.6 (Contrainte inconsistante)**.** *Soit C une contrainte bâtie sur un domaine de calcul $\mathcal{D}$. S'il existe une variable $v \in Var(C)$ telle que $dom(v, C) = \emptyset$, alors la contrainte C est dite inconsistante.*

Ainsi, dans la première clause par exemple, les contraintes $D1 = D, F1 = F, D1 = 5, F1 = 10, D \geq F$ provoquent une inconsistance puisque la conjonction logique $D1 = D \wedge F1 = F \wedge D1 = 5 \wedge F1 = 10 \wedge D \geq F$ est une contradiction dans la logique du premier ordre définie sur l'ensemble des nombres naturels. Le domaine de la variable $D1$ est en effet d'abord réduit à $\{5\}$, puis contraint de façon à ce que $D1 \geq 10$, ce qui ne laisse plus de valeurs possibles pour $D1$.

Lorsqu'une clause *cl* présente une contrainte inconsistante, nous nous permettrons l'abus de langage de dire que *cl* est inconsistante. De même, si toutes les clauses définissant un prédicat sont inconsistantes, nous dirons que ce prédicat est inconsistant.

En termes opérationnels, le passage en revue d'une clause inconsistante par le moteur d'inférences rendra toujours le sac de contraintes inconsistantes. En présence de l'hypothèse du monde clos, il est donc déclarativement équivalent d'ôter cette clause du programme. La transformation consistant en la suppression de clauses inconsistantes est définie comme suit.

Définition 4.7 (Transformation : suppression de clauses inconsistantes)**.** *La suppression de clauses inconsistantes, notée* incons *est une transformation de programme y telle que, pour tout programme P_1, $y(P_1) = P_2$ où P_2 est identique à P_1 sauf que des clauses inconsistantes peuvent y avoir été supprimées.*
Cette transformation admet un paramètre : clauses*, un ensemble d'indices correspondant aux indices des clauses à supprimer, par défaut vide. Ces clauses doivent être inconsistantes. La suppression de clauses inconsistantes admet un méta-paramètre :* suppr_preds*, par défaut à $false$. S'il vaut $true$, alors il est interdit de supprimer l'entièreté des clauses définissant un prédicat. S'il vaut $false$, alors la suppression totale d'un prédicat implique la suppression, en cascade, de toutes les clauses qui y font appel, ainsi que de toutes les clauses qui appelaient les prédicats totalement supprimés par ce nouveau processus de suppression, et ainsi de suite.*

Le méta-paramètre *suppr_preds* permet de définir la stratégie de suppression de prédicats. Soit la suppression totale d'un prédicat n'est pas permise, même si toutes les clauses qui le constituent sont inconsistantes et qu'en conséquence, la valeur de vérité de ce prédicat est toujours $false$. Soit nous permettons la suppression de ces prédicats inconsistants, puisque leur présence ou non dans la base de connaissances n'a pas d'impact sur le programme au vu de l'hypothèse du monde clos. Dans ce cas, toute clause qui fait appel à ces prédicats est elle-même inconsistante et peut être supprimée ; les pans d'inconsistance du programme peuvent alors entièrement être élagués. Cette dernière stratégie est la stratégie par défaut du méta-paramètre. Toutefois, aucune des deux valeurs de *suppr_preds* n'a d'impact sur la sémantique déclarative des programmes en présence de l'hypothèse du monde clos.

Proposition 4.8. *La suppression de clauses inconsistantes est une transformation préservatrice de sem_0, peu importe la valeur de son méta-paramètre.*

Notons que si toutes les clauses d'un programme sont inconsistantes, alors cette transformation peut supprimer l'entièreté du programme lorsque la valeur $false$ est assignée à son méta-paramètre. Un tel programme *vide* est bien déclarativement équivalent à un programme dont les chemins d'exécution mènent tous irrémédiablement à $false$.

Le dépliage est typiquement couplé à la suppression de clauses inconsistantes. Sans cela, les programmes dépliés tendent à contenir un nombre de clauses inconsistantes relativement élevé, en fonction (entre autres) du degré d'instanciation des variables. Reprenons la portion de code 4.15 et appliquons-lui une suppression de clauses inconsistantes. Nous obtenons la portion de code 4.16.

Portion de code 4.16 – Programme S après dépliage complet et nouveau point d'entrée et nouveau dépliage et suppression des clauses inconsistantes

```
p_start(Max) :- {A1=0, B1=0, C1=0, G1=0, H1=0, D1=5, E1=4, F1=10,
        A1=A, B1=B, C1=C, D1=D, E1=E, F1=F, G1=G, H1=H, Max=I,
        D<F, D>=E, F=I, G=H}.

p_start(Max) :- {A1=0, B1=0, C1=0, G1=0, H1=0, D1=5, E1=4, F1=10,
        A1=A, B1=B, C1=C, D1=D, E1=E, F1=F, G1=G, H1=H, Max=I,
        D<E, E<F, F=I, G=H}.

p_max(A,B,C,D,E,F,G,H,I):- {D>=F, D>=E, D=I, G=H}.
p_max(A,B,C,D,E,F,G,H,I):- {D<F, D>=E, F=I, G=H}.
p_max(A,B,C,D,E,F,G,H,I):- {D<E, E>=F, E=I, G=H}.
p_max(A,B,C,D,E,F,G,H,I):- {D<E, E<F, F=I, G=H}.
```

Dans ce programme subsistent deux clauses pour le prédicat $p\_start/1$: les deux cas où c'est F le maximum. Une examination plus précise nous indique que les contraintes de l'une des clauses sont sémantiquement « inclues » (impliquées) dans celle de l'autre. Il s'agit d'une conséquence de la traduction du code impératif en code CLP, qui n'a pas d'impact sur le sens du programme. En effet, une simple agrégation suivie d'un remplacement de contraintes, et un nettoyage, réduira la portion de code 4.16 à une seule clause simple et lisible.

4.4 Pliage

Le pliage est la transformation inverse du dépliage. Là où le dépliage permet d'expliciter des calculs en les concrétisant, le pliage rajoutera des étapes à l'exécution d'un prédicat. En d'autres mots, le pliage permet d'ajouter des intermédiaires, alors que le dépliage a tendance à supprimer les intermédiaires.

Ces deux transformations se répondent et une seule d'entre elle suffirait dans notre cadre de travail pour nous armer face à cette problématique duale. Nous avons néanmoins choisi d'introduire brièvement le pliage pour plusieurs raisons :

- D'abord, le pliage n'est pas nécessairement l'inverse exact du dépliage. En effet, après l'exécution d'un dépliage, les contraintes de la clause dépliée sont dénaturées puisqu'elles sont la conjonction de contraintes issues de diverses clauses. Il est difficile de définir le pliage de telle façon à ce qu'il soit capable de retrouver quelles contraintes sont issues de quelles clauses dans le but d'effectuer l'opération dans l'autre sens. À l'inverse, un pliage est en général réversible à l'aide du dépliage. Mais il existe différentes versions de pliage, dont certaines ne sont pas inversibles ([3]).
- Ensuite, le pliage est parfois utilisé comme un complément au dépliage plutôt que comme sa réciproque. Dans [11], pliage et dépliage sont utilisés de concert dans le but d'apparier deux prédicats entre eux. Dans [29], plusieurs stratégies de transformations de programmes sont exposées, et dans presque toutes ces stratégies, pliage et dépliage sont utilisés de façon complémentaire.
- Enfin, des techniques de transformation basées sur le seul pliage ont également été investiguées. Dans [36], par exemple, une règle de pliage est inventée dans le but d'éliminer les variables existentielles des contraintes afin que celles-ci ne soient exprimées qu'en fonction des variables apparaissant dans les têtes des clauses. Le pliage, à lui seul, peut donc avoir des effets intéressants sur les clauses, que le dépliage ne pourrait avoir.

Notre définition du pliage s'appuie principalement sur [35] et [3]. Ces sources proposent également des algorithmes de pliage. Nous ne faisons ici que survoler la transformation et renvoyons vers ces articles pour les questions relatives à son implémentation.

Définition 4.9 (Pliage d'une clause)**.** *Soit P un programme écrit en $CLP(X)$ et sem une sémantique sur $CLP(X)$. Soient h une clause de P de la forme $H : -\{C_h\}, B_h$. et d une clause de P de la forme $D : -\{C_d\}, B_d$. S'il existe :*

- *une contrainte C_e,*
- *un renommage de variables θ, et*
- *une conjonction d'appels R*

*tels que h soit sémantiquement équivalente par rapport à sem à $H : -\{C_e \wedge \theta(C_d)\}, \theta(B), R$, alors h est **pliée selon** d en la clause $H : -\{C_e\}, \theta(D), R$.*

Définition 4.10 (Transformation : pliage)**.** *La transformation de pliage, notée* pliage*, est une transformation de programme y telle que, pour tout programme P_1, $y(P_1) = P_2$ où P_2 est identique à P_1 sauf que des clauses peuvent y avoir été pliées.*
Cette transformation admet un paramètre : clauses*, une fonction associant à des clauses qui doivent être pliées, les indices des clauses avec lesquelles elles seront pliées (il faut que les clauses respectent les conditions de la définition 4.9), et le résultat du pliage. Elle n'admet pas de méta-paramètre.*

L'exemple suivant issu de [35] illustre le concept de pliage.

Exemple 4.11. *Soient les deux clauses suivantes :*

$$p(X,Y) : -\{X > 1, X > T\}, q([\,], T), r(Y). \tag{4.4}$$

et

$$s(U,V,A) : -\{U > 1, V > 0, U > W\}, q(A,W). \tag{4.5}$$

Alors plier 4.4 selon 4.5 peut générer la clause :

$$p(X,Y) : -\{X > 1, X >= U\}, s(U,V,[\,]), r(Y). \tag{4.6}$$

Nous disons « peut générer » parce qu'en général, plusieurs pliages différents peuvent être possibles pour une clause par rapport à une autre.

Le résultat suivant est prouvé dans [35] et [3]. Nous nous en convaincrons en observant que déplier le résultat d'un pliage revient (à un renommage de variables près) à récupérer les clauses dans l'état dans lequel elles étaient avant le pliage.

Proposition 4.12. *Le pliage est une transformation préservatrice de sem_0.*

Des versions alternatives du pliage sont discutées dans [35].

La condition nécessaire pour effectuer un pliage n'est pas toujours remplie. Une transformation souvent liée au pliage, et qui permet d'assurer son bon fonctionnement, est la définition de clauses. En définissant une nouvelle clause qui remplisse les exigences de la définition 4.9, un pliage sera possible ([11]). La définition de clauses est en quelque sorte l'inverse du nettoyage.

Définition 4.13 (Transformation : définition de clauses)**.** *La transformation de définition de clauses, notée* def*, est une transformation de programme y telle que, pour tout programme P_1, $y(P_1) = P_2$ où P_2 est identique à P_1 sauf que de nouvelles clauses, chacune définissant un nouveau prédicat qui n'apparaît pas dans P_1, peuvent y avoir été introduites.*
Cette transformation admet un paramètre : clauses*, l'ensemble des clauses qui seront ajoutées au programme (ces clauses doivent chacune définir un prédicat distinct et n'apparaissant pas dans le programme). Elle n'admet pas de méta-paramètre.*

Exemple 4.14. *Soit h la clause $p(X,Y) : -\{X > 10\}, q(X), r(Y), s(X,Y)$.*
Appelons d une nouvelle clause $def(A,B,C,D) : -\{\}, q(A), r(B), s(C,D)$.
Alors l'on peut plier h selon d, générant la clause

$$p(X,Y) : -\{X > 10\}, def(X,Y,X,Y).$$

La définition de clause ne modifie pas le sens déclaratif d'un programme, à condition que l'on y tolère l'existence d'un point d'entrée.

Proposition 4.15. *La définition de clauses est préservatrice de sem_1.*

4.5 Tranchage

Pour pallier la situation de l'ajout de variables, il nous faudrait une transformation qui puisse supprimer du programme les variables n'ayant pas d'impact sur celui-ci. Dans la même veine, les appels redondants doivent encore être gérés par notre cadre de travail : il serait pratique de pouvoir les supprimer. Nous avons pour cela choisi le **tranchage**, une transformation éprouvée dans le monde de la programmation logique. Le tranchage est particulièrement explicité dans [38], [44], [33] et [40]. En sa forme la plus générale, le tranchage consiste à extraire d'un programme donné des *tranches* de programme. Une tranche est alors une variante du programme initial où des clauses, appels, arguments ou contraintes peuvent avoir été supprimés. Nous avons déjà rencontré

deux formes de tranchage de clauses à travers le nettoyage et la suppression de clauses inconsistantes. Le remplacement de contraintes représente par ailleurs une façon de trancher des contraintes en les simplifiant.

Dans cette section, nous nous concentrerons d'abord sur le tranchage d'appels et de clauses dont la présence n'est pas nécessaire. Nous nous focaliserons ensuite sur le tranchage d'arguments et en présenterons une extension particulière : l'extraction de tranches de programmes par rapport à des variables dites *d'intérêt*. De fait, la plupart des auteurs présentent le tranchage dans un but de débogage. En extrayant une tranche de programme correspondant aux seules instructions (ou clauses, contraintes et appels dans le cas des programmes CLP) en lien avec certaines variables d'intérêt, il est possible de se focaliser sur le flux d'exécution de ces variables et donc de comprendre à quel niveau une erreur peut survenir dans leur manipulation. Nous verrons en quoi cette forme de tranchage peut également avoir sa place dans notre cadre de travail.

4.5.1 Tranchage d'appels

Nous avons vu que la duplication de code peut entraîner la répétition, dans une même clause, d'appels vers un même prédicat et avec les mêmes arguments.

Définition 4.16 (Transformation : tranchage d'appels)**.** *Le tranchage d'appels est une transformation y notée* tranch_appels *telle que, pour tout programme P_1, $y(P_1) = P_2$ où P_2 est identique à P_1 à la seule différence près que des appels de prédicats peuvent avoir été supprimés.*
Cette transformation admet un paramètre : f*, une fonction associant à zéro, un ou plusieurs identifiants de clauses, un ou plusieurs appels devant être supprimés. Elle admet également un méta-paramètre,* redondants*, par défaut à true. Si ce méta-paramètre est mis à true, alors la transformation est forcée de ne supprimer que des appels « doublons » et aucun autre appel dans le programme.*

La proposition suivante découle du fait que la sémantique déclarative des programmes n'est pas altérée lorsque des appels redondants sont supprimés.

Proposition 4.17. *Le tranchage d'appels est préservateur de sem_0.*

4.5.2 Tranchage de tautologies

Une autre duplication, plus sémantique, au niveau des appels, est le cas d'un prédicat qui, dans une de ses clauses, ne s'appelle que lui-même sans ne rien contraindre. Ce cas peut par exemple survenir lorsqu'un développeur malintentionné souhaite bouleverser la compréhension d'un code plagié en y ajoutant des clauses inutiles.

Définition 4.18 (Clause récursivement tautologique)**.** *Soit P un programme logique. Alors une clause c de P est dite récursivement tautologique si c est de la forme $p(V_1, V_2, ..., V_n) : -\{\}, p(V_1, V_2, ..., V_n)$. C'est-à-dire :*

1. *c n'a pas de contraintes ;*
2. *c fait un seul appel, qui est strictement identique à sa tête.*

Intuitivement, une telle clause est tautologique parce que l'implication sous-jacente est $p(V_1, V_2, ..., V_n) \Rightarrow p(V_1, V_2, ..., V_n)$: une tautologie (peu importe la valeur des variables, la clause sera vraie).

Toutes les clauses tautologiques ne le sont pas nécessairement récursivement ; certaines le sont, par exemple, à cause de la forme de leurs contraintes. Dans la suite, nous parlerons de *tautologie* pour dénoter toute clause de Horn tautologique.

Exemple 4.19. *Dans le code suivant écrit en $CLP(\mathbb{N})$, c'est tout le prédicat $p/1$ qui est tautologique, puisque ses deux clauses sont des tautologies. Elles sont en effet vraies peu importe la valeur de l'argument X.*

```
p(X) :- {X > 5 \/ X < 5 \/ X = 5}.
p(X) :- {X >= 0}.
```

Il est naturel de souhaiter supprimer les tautologies puisque leur présence dans la base de connaissances est superflue. L'implication qu'elles soutiennent est en effet sous-entendue. Si un prédicat est tautologique, il serait également pratique de le supprimer, et de supprimer tous les appels effectués vers ce prédicat, puisque ces appels sont équivalents à un atome qui réussit toujours (l'atome *true*, en Prolog).

Définition 4.20 (Transformation : tranchage de tautologies)**.** *Le tranchage de tautologies est une transformation y notée* tranch_taut *telle que, pour tout programme P_1, $y(P_1) = P_2$ où P_2 est identique à P_1 à la seule différence près que des clauses tautologiques peuvent avoir été supprimées. Lorsque la dernière clause définissant un prédicat p/a est supprimée de cette façon, tous les appels vers ce prédicat sont également supprimés.*
Cette transformation admet un paramètre, clauses*, un ensemble d'entiers correspondant aux indices des clauses tautologiques à supprimer. Elle n'admet pas de méta-paramètre.*

L'inoffensivité du tranchage de tautologies (ou, de façon équivalente, de l'ajout de tautologies) quant au sens purement déclaratif des programmes logiques sans contraintes est assuré par le *principe d'extension raffiné* exploré dans [1]. L'extension de ce résultat aux programmes CLP est garanti par le fait que sem_0 n'est jamais qu'une extension de la sémantique déclarative des programmes logiques sans contraintes et que la notion de tautologie y est donc identique.

Proposition 4.21. *Le tranchage de tautologies est préservateur de sem_0.*

Notons que, pour des sémantiques opérationnelles telles que celle du moteur d'inférences de Prolog, trancher une tautologie n'est pas sans conséquences sur l'exécution du programme. En effet, une tautologie telle que $p :- p$ entraîne potentiellement, selon sa place dans le programme, une boucle infinie et le sens du programme est alors $\bot$. Dans sem_0, seul l'aspect déclaratif des programmes a de l'importance. Or énoncer une tautologie ou non n'affectera pas les modèles logiques qui sous-tendent le programme CLP, et n'affectera donc pas sem_0.

Nous l'avons vu : les tautologies peuvent revêtir différentes formes. L'identification de ces formes, cruciale pour l'implémentation du tranchage de tautologies, est un travail futur.

4.5.3 Tranchage d'arguments

Dans de nombreuses situations, l'on pourrait souhaiter supprimer des arguments inutiles pour le bon fonctionnement d'un programme, ou pour un mode de fonctionnement particulier du programme. Une transformation utile dans ces circonstances est le tranchage d'arguments, défini comme suit.

Définition 4.22 (Transformation : tranchage d'arguments)**.** *Le tranchage d'arguments, noté* tranch_args *est une transformation y telle que, pour tout programme P_1, $y(P_1) = P_2$ où P_2 est identique à P_1 à la seule différence près que des arguments de prédicats peuvent avoir été supprimés. Ces arguments sont alors ôtés de toutes les têtes des clauses définissant ces prédicats, et les arguments apparaissant aux mêmes positions dans tous les appels effectués à ces prédicats sont également supprimés du programme.*

Cette transformation est à première vue trop puissante pour préserver nos sémantiques. En effet, avec cette définition, un argument pourtant essentiel au bon fonctionnement d'un programme peut lui être ôté. Nous allons, comme précédemment, cadenasser le comportement de cette transformation à l'aide de paramètres et méta-paramètres. Ceux-ci sont présentés dans le tableau 4.2.

Même avec ces méta-paramètres, le tranchage n'est pas préservateur de sem_1 car sem_1 est une sémantique purement déclarative (mis à part qu'elle ne se soucie pas des clauses non-accessibles depuis un point d'entrée). Ôter des arguments dans des prédicats accessibles depuis le point d'entrée va donc modifier le sens déclaratif du programme par rapport à sem_1.

Nous avons déjà argumenté en faveur de l'inclusion de points d'entrée dans nos programmes. À présent, il convient de se demander si les arguments de ces points d'entrée sont les seuls qui nous « intéressent ». De fait, lors de l'exécution d'un programme CLP, aucun résultat n'est renvoyé : seulement une unification des variables dans la requête posée au programme. Or les seules requêtes posées au programme seront faites à son point d'entrée puisque par définition, il s'agit de la représentation de la « première instruction » du programme. Le **comportement extérieur** du programme est donc entièrement défini par les unifications possibles sur les requêtes de type $pe(V_1, ..., V_n)$, où pe/n est le point d'entrée du programme. Nous allons noter sem_2 la sémantique sem_1 restreinte telle que le sens d'un programme est uniquement défini par son point d'entrée. Autrement dit, dans sem_2, tout modèle logique du point d'entrée est un modèle logique du programme complet, et réciproquement. Le fonctionnement interne du programme, qui dépend de ses autres clauses, n'a pas d'impact sur cette fonction sémantique. Il s'agit donc d'une sémantique moins forte (ou plus générale) que sem_1 et, *a fortiori*, que sem_0.

Le lien entre le tranchage d'arguments et sem_2 est explicité dans la proposition suivante.

Proposition 4.23. *Si ses trois méta-paramètres sont instanciés avec leurs valeurs par défaut, le tranchage d'arguments est préservateur de sem_2.*

Clairement, le premier méta-paramètre est nécessaire pour assurer ce résultat. Sans lui, le tranchage pourrait supprimer des arguments de façon à ce que deux prédicats pourtant différents portent le même nom et aient la même arité, ce qui est susceptible de modifier le comportement du programme. Nous reviendrons sur le deuxième méta-paramètre après avoir introduit les classes d'équivalence pour la relation $\sim_P$ définie dans le paragraphe 4.5.4. Quant au méta-paramètre *point_entree*, sa valeur par défaut empêche le tranchage de modifier un point d'entrée, et donc d'altérer le sens du programme par rapport à sem_2.

Tableau 4.2 – Paramètres et méta-paramètres du tranchage d'arguments

(Méta-)paramètre	Type	Explication	Valeur par défaut
preds	Paramètre	Fonction associant à zéro, un ou plusieurs prédicats du programme, un ensemble d'indices correspondant aux indices des arguments à trancher.	{}
predicats_du_meme_nom	Méta-paramètre	Si mis à *true*, interdit, dans un programme contenant un prédicat p/a_1 le tranchage de n arguments d'un prédicat p/a si $a - n = a_1$.	*true*
variables_utiles	Méta-paramètre	Si mis à *true*, interdit le tranchage de tout argument en position α tel que la classe d'équivalence $[\alpha]_{\sim_P^*}$ contient au moins une position dans une contrainte. (Voir paragraphe 4.5.4)	*true*
point_entree	Méta-paramètre	Si mis à *true*, permet le tranchage d'arguments de tout prédicat du programme. Si mis à $false$, interdit le tranchage d'arguments du point d'entrée du programme.	$false$

4.5.4 Tranchage en deux passes

Nous adaptons ici un algorithme de tranchage général (dans le sens où il peut amener à supprimer clauses, appels, arguments, et contraintes) exposé dans [39]. Cet algorithme nous permettra ensuite d'investiguer des questions liées à la place du tranchage dans notre cadre de travail. Nous ne donnons ici que l'intuition de cet algorithme dont la correction est assurée dans l'article en question.

L'algorithme fait appel à trois concepts centraux : les arcs de contraintes, les arcs de transition et les arcs locaux. Ces concepts s'appuient sur la notion de position dans le programme [5].

Définition 4.24 (Position)**.** *Soit P un programme CLP. Alors une position de ce programme est l'un des éléments suivants :*

- *un argument formel dans la tête d'une clause ;*
- *une variable dans une contrainte ;*
- *un argument dans un appel.*

Définition 4.25 (Relation de dépendance directe)**.** *Soient α et β deux positions dans un programme P. Alors la relation de dépendance directe $\sim_P$ est une relation d'équivalence définie telle que $\alpha \sim_P \beta$ si et seulement si une des conditions suivantes est vérifiée :*

- *α et β sont des positions de variables dans une même contrainte, et y sont liées par un symbole prédicatif. On parle alors d'**arc de contrainte**. Par exemple si α représente la variable X et β la variable Y, alors en présence de la contrainte $X = Y + 2$, on a $\alpha \sim_P \beta$.*
- *α est une position d'argument en tête d'une clause définissant un prédicat p/a et β est une position d'argument ayant même indice dans un appel à p/a. On parle alors d'**arc de transition**.*
- *α et β sont une même variable dans une même clause. On parle alors d'**arc local**.*

Nous revenons sur le méta-paramètre *variables_utiles* du tranchage d'arguments. Ce méta-paramètre représente la permission ($false$) ou l'interdiction ($true$) de supprimer des arguments liés par la relation de dépendance directe à une position dans une contrainte. Lorsqu'il est mis à $true$, nous ne pourrons donc supprimer que les arguments qui n'ont aucun impact sémantique sur le programme (s'ils avaient eu un impact, ils auraient été contraints ou auraient servi à contraindre une autre variable).

Ce méta-paramètre diminue donc fortement la puissance du tranchage d'arguments quand il est mis à $true$. En effet, considérons les clauses de la portion de code 4.17 d'un programme écrit en $CLP(\mathbb{N}_0)$. L'argument de $p/1$ ne sera pas tranché parce qu'il est contraint, mais cette contrainte n'a pas d'implication sémantique étant donné que nous travaillons déjà en $CLP(\mathbb{N}_0)$. L'argument de $q/1$ non plus ne pourra être tranché, parce qu'il est contraint. En l'occurrence, il doit être égal à une variable non contrainte D. Enfin, aucun des deux arguments de $r/2$ ne pourra être tranché, alors que le premier argument (X) n'a pas d'influence sur le sens du programme. Mais le fait qu'il soit contraint ($X \leq D$) le rend à nouveau intouchable. Dans $s/2$, le premier argument est également intouchable, à cause de l'arc de transition qui existe entre la troisième et la quatrième clauses.

Portion de code 4.17 – Arguments non tranchables

```
p(X) :- {X > 0}.
q(X) :- {X = D}.
r(X, Y) :- {X =< D, Y > 3}, s(D, Y).
s(X, Y) :- {Y = 5}.
```

Dans le premier cas, un tranchage de tautologies pourra résoudre le problème. Dans le second cas, un remplacement de contrainte, accompagné par un tranchage de tautologies, fera l'affaire.

Dans la troisième clause, cependant, il nous faudrait un mécanisme plus puissant pour détecter que non seulement X, mais aussi D, sont des variables parasites dans $r/2$. De même, X est une variable inutile au fonctionnement de $s/2$. C'est pour ce genre de situations que l'algorithme de [39] nous sera utile.

Pour déterminer la tranche de programme liée à une certaine variable d'intérêt située en position α, l'idée de cet algorithme va être de calculer intelligemment la classe d'équivalence $[\alpha]_{\sim_P^*}$. Intuitivement, pour construire cette classe d'équivalence, il suffit de suivre les règles exposées dans la définition 4.25. Les positions n'appartenant pas à cette classe seront tranchées hors du programme car elles n'entreront pas en compte dans le flux

5. Pour les deux définitions ci-dessous, nous faisons par pure simplicité l'hypothèse que les arguments de tous les atomes des clauses sont des variables et non des constantes ou des termes imbriqués. Cette hypothèse, toujours vérifiable en présence du remplacement d'arguments, facilitera la présentation de la suite de cette section. Elle n'a toutefois aucun impact sur les résultats.

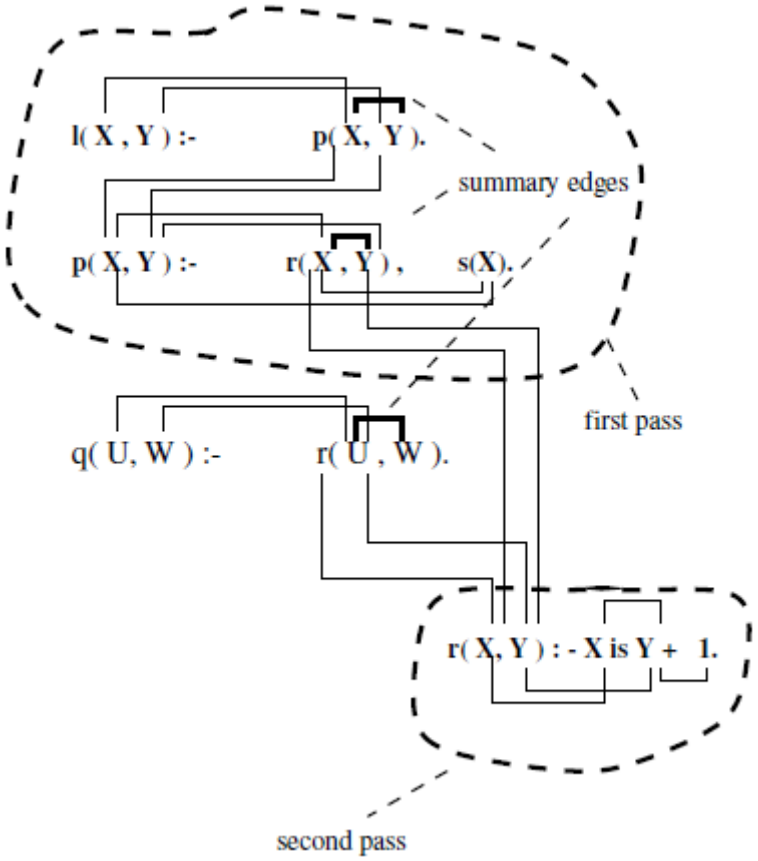

FIGURE 4.1 – Tranche de programme calculée avec et sans technique des deux passes ([39])

d'exécution de la variable de départ. Le même processus peut être appliqué à un groupe de variables plutôt qu'à une seule.

Une subtilité discutée en détail dans [39] est le problème du **contexte d'appel**. En résumé, les classes d'équivalence en tant que telles risquent d'englober de trop nombreuses positions dans une tranche à cause des arcs de transition : les arguments d'un prédicat appelé de nombreuses fois vont être reliés à chacun de ces contextes appelants, sans préoccupation pour la tranche courante. Cette problématique est illustrée dans la figure 4.1, où la tranche de la variable X de la deuxième clause est calculée. Le prédicat $q/2$, qui n'a pas véritablement de rapport avec le cycle de vie de cette variable, fera malgré tout partie de la tranche, parce que U et W y sont reliés par des arcs de transition à la clause qui définit $r/2$.

Pour résoudre le problème, les auteurs suggèrent d'utiliser une technique dite en **deux passes**. Lors de la première passe, les arcs de transition depuis des appels vers des têtes sont interdits. Lors de la seconde passe, les arcs de transition depuis des têtes vers des appels sont interdits. Ainsi, dans la figure 4.1, l'on commence par « remonter » dans le programme en cherchant les appels à $p/2$. Il y en a un : dans la première clause. L'on continue à remonter en cherchant les appels à $l/2$. Il n'y en a pas ; c'est la fin de la première passe. L'on marque bien dans les clauses ainsi parcourues dans la première passe les positions liées par des arcs de contraintes et locaux. Nous pouvons alors entamer la deuxième passe, en redescendant par les appels dont des positions ont été marquées. C'est ainsi que le prédicat $r/2$ est atteint. Comme il n'y a plus d'arc à explorer, le tranchage est terminé.

Remarquons que ce processus nous a permis d'éviter l'inclusion de $q/2$ dans la tranche. Il aurait fait partie de la tranche si l'on s'était permis d'effectuer un arc de transition depuis la tête de $r/2$ vers l'appel de $q/2$. Cette situation, en cas réel, est très fréquente, et c'est ce qui nous a motivé à implémenter le tranchage en suivant cet algorithme dans le prototype développé en parallèle du présent mémoire.

Une dernière remarque notable est qu'avec cet algorithme employé tel quel, le Y de la deuxième clause n'aurait pas été ajouté à la tranche, puisqu'il n'est nulle part contraint avec X. Cette difficulté est adressée dans [31] notamment, où l'on suggère l'utilisation d'arcs particuliers (*summary edges*) qui lient les positions corres-

pondant à des variables dont on sait qu'elles seront contraintes de concert par la suite. Dans l'exemple, X et Y sont contraintes par X *is* $Y+1$ dans $r/2$ et doivent donc être liées par un arc particulier dès la première passe.

Nous pouvons à présent définir la transformation de tranchage général.

Définition 4.26 (Transformation : tranchage général)**.** *Le tranchage général, noté* tranch_gen*, est une transformation de programmes CLP y telle que, pour tout programme P_1, $y(P_1) = P_2$ où P_2 est une copie de P_1 où des variables, clauses, appels et contraintes peuvent avoir été supprimés, de façon à ce que P_2 puisse être obtenu comme tranche de P_1 par rapport à des variables d'intérêt avec le processus de tranchage en deux passes présenté ci-dessus.*
Cette transformation admet un paramètre : vars*, les variables d'intérêt (sous la forme d'un ensemble de positions dans le programme) dont la tranche doit être extraite.*

Pour trancher un programme au mieux, il est naturel d'appeler l'algorithme de tranchage en deux passes sur les clauses définissant son point d'entrée, avec comme positions initiales celles de tous leurs arguments. Le flot de ces arguments dans le programme sera automatiquement parcouru et leur classe d'équivalence (optimisée par la technique des deux passes) calculée dans le même temps. Tous les arguments ne faisant, au final, pas partie de cette classe d'équivalence, pourront être supprimés du programme sans impact sur celui-ci. De même pour les clauses, appels et contraintes dont aucune position n'apparaît dans la classe d'équivalence. Le fait d'inclure tous les arguments du point d'entrée dans le processus initial garantit que le sens de celui-ci sera conservé.

Proposition 4.27. *$tranch\_gen(vars = Pos(Vars(pe)))$, avec $Vars(pe)$ les variables qui apparaissent dans les têtes des clauses définissant le point d'entrée du programme, et $Pos(Vars(pe))$ leurs positions dans ces têtes, est préservateur de sem_2.*

4.5.5 Connexion de tranches

Nous avons souvent considéré qu'un programme recelait un seul algorithme. Il nous a donc semblé naturel de considérer tous les arguments d'un point d'entrée comme intouchables par le tranchage. Dans les faits, dans un même programme peuvent coexister plusieurs calculs distincts. Par exemple, dans le programme de la portion de code 4.18, dont le point d'entrée est $p/3$, la somme cumulée et le produit cumulé d'une liste sont tous deux calculés en parallèle.

Portion de code 4.18 – Algorithmes de somme et de produit coexistants

```
p(Xs, Somme, Produit) :- {}, q(Xs, 0, Somme, 1, Produit).
q([], SommeAcc, Somme, ProduitAcc, Produit) :- {SommeAcc = Somme, ProduitAcc = Produit}.
q([X|Xs], SommeAcc, Somme, ProduitAcc, Produit) :-
        {SommeAcc1 = SommeAcc + X, ProduitAcc1 = ProduitAcc * X},
        q(Xs, SommeAcc1, Somme, ProduitAcc1, Produit).
```

Si nous comparons ce programme à un programme ne calculant que la somme cumulée d'une liste, avec nos transformations habituelles, nous arriverions irrémédiablement à la conclusion que les deux algorithmes capturés par ces programmes ne sont pas $\mathcal{Y}$-connectés. Or il pourrait être intéressant de pouvoir extraire l'algorithme commun des deux programmes. La procédure de tranchage présentée dans le paragraphe précédent peut nous aider en ce sens. Pour illustrer cela, reprenons la portion de code 4.18 et calculons les tranches des variables *Somme* et *Produit* de $p/3$. Les résultats respectifs sont présentés dans les portions de code 4.19 et 4.20.

Portion de code 4.19 – Tranche de la variable *Somme*

```
p(Xs, Somme) :- {}, q(Xs, 0, Somme).
q([], SommeAcc, Somme) :- {SommeAcc = Somme}.
q([X|Xs], SommeAcc, Somme) :- {SommeAcc1 = SommeAcc + X},
        q(Xs, SommeAcc1, Somme).
```

Portion de code 4.20 – Tranche de la variable *Produit*

```
p(Xs, Produit) :- {}, q(Xs, 1, Produit).
q([], ProduitAcc, Produit) :- {ProduitAcc = Produit}.
q([X|Xs], ProduitAcc, Produit) :- {ProduitAcc1 = ProduitAcc * X},
        q(Xs, ProduitAcc1, Produit).
```

Dans les tranches de programmes générées, l'on retrouve bien l'indépendance des deux algorithmes du programme de départ. Notons bien que si nous avions tranché à partir de la variable Xs, la tranche résultat aurait été le programme complet. Il n'est donc pas possible de créer un partitionnement du programme en tranches univoques.

Définition 4.28 ($\mathcal{Y}$-connexion par les tranches)**.** *Soient T_1 et T_2 deux tranches issues des programmes respectifs P_1 et P_2 (avec éventuellement $T_1 = P_1$ ou $T_2 = P_2$). Soit $\mathcal{Y}$ un ensemble de transformations. Alors on dira que P_1 et P_2 sont* ***$\mathcal{Y}$-connectés par leurs tranches*** *T_1 et T_2 si T_1 et T_2 sont $\mathcal{Y}$-connectées.*
Si T_1 et T_2 sont $\mathcal{Y}$-équivalentes, on dira que P_1 et P_2 sont ***$\mathcal{Y}$-équivalents par leurs tranches*** *T_1 et T_2.*

Une extension naturelle de sem_2, que nous noterons sem_3, est la sémantique orientée point d'entrée qui fait abstraction d'un certain nombre d'arguments du point d'entrée, et ne considère que les arguments résiduels comme ayant du sens pour le programme. sem_3 est donc paramétrée par un ensemble S, qui correspond aux indices des arguments du point d'entrée à laquelle de l'importance est accordée. Nous capturons l'incarnation de sem_3 par rapport à l'ensemble S dans la notation sem_3^S. Dans la portion de code 4.18, le tranchage effectué sur le deuxième argument de $p/3$ est ainsi préservateur de $sem_3^{\{1,2\}}$, alors que le tranchage effectué sur le troisième argument est préservateur de $sem_3^{\{1,3\}}$. Pour un programme ayant comme point d'entrée pe/n, sem_3^S est au moins aussi générale que sem_2, puisqu'elle est soit moins exigente (si $|S| < n$), soit aussi exigente (si $S = \{1, 2, \ldots, n\}$) que sem_2.

L'observation suivante s'attache à faire le lien entre $\mathcal{Y}$-équivalence par rapport à sem_3, tranchage en deux passes et $\mathcal{Y}$-équivalence par les tranches. Elle restera à l'état de conjecture, sa preuve dépassant l'objectif du présent mémoire.

Conjecture 4.29. *Soient P_1 et P_2 deux programmes $\mathcal{Y}$-équivalents par leurs tranches T_1 et T_2. Si T_1 et T_2 peuvent être obtenues par un tranchage en deux passes effectué sur des arguments du point d'entrée de P_1 et P_2 respectivement, alors il existe*

- *un ensemble $S \subset \mathbb{N}$,*
- *des copies de P_1 et P_2, notées P_1' et P_2', où les positions des arguments des points d'entrée ont éventuellement été permutées,*

tels que P_1' et P_2' sont $\mathcal{Y}$-équivalents par rapport à sem_3^S.

4.6 Gestion des boucles

Dans les programmes logiques, les boucles se matérialisent par des chaînes d'appels cycliques [6]. Nous avons vu que même notre transformation la plus puissante, le dépliage, n'est pas capable d'expliciter les boucles de façon générale puisque cela risque d'engendrer un processus infini. Or certaines boucles peuvent être explicitées de façon finie, à l'aide de ce que l'on appelle le *déroulage de boucles*. De plus, certaines boucles n'ont pas d'impact sur le programme dans lequel elles sont définies. D'autres sont écrites de façon non conventionnelle, enchevêtrées dans un flux d'exécution particulier. Nous voyons, dans cette section, certaines limites de notre cadre de travail que ces situations mettent en lumière.

4.6.1 Déroulage de boucles

Dans certains cas, les boucles peuvent être déroulées. L'exemple typique, en termes impératifs, est une boucle `for` où il est évident que l'indice est incrémenté un nombre fini de fois. Voici un exemple de programme logique où un calcul est effectué en trois tours de boucle (l'incrément est la variable I) :

```
p(X, Resultat) :- {I = 0}, q(I, X, Resultat).
q(I, X, Resultat) :- {I < 3, X1 = X + X, I1 = I + 1}, q(I1, X1, Resultat).
q(I, X, Resultat) :- {I = 3, Resultat = X}.
```

Avec des dépliages successifs sur la première clause, intercalés par la suppression des clauses dont les contraintes sont inconsistantes, et par un remplacement de contraintes dans un souci de lisibilité, nous obtenons les résultats successifs présentés dans la portion de code 4.21.

Nous le voyons : la simple utilisation du dépliage et de la suppression de clauses inconsistantes permet le déroulage de boucles, grâce à la capacité du dépliage d'exécution des programmes manipulés, et donc d'explicitation de chaque tour de boucle.

6. Le terme *boucle* est plus adapté aux langages impératifs, mais nous nous permettrons par abus de langage de l'utiliser pour dénoter les chaînes d'appels cycliques qui peuvent apparaître dans les programmes logiques.

Portion de code 4.21 – Boucle `for` avec 3 itérations traduite en $CLP(\mathbb{N})$ et dépliée

```
% Premier dépliage
p(X, Resultat) :- {I = 0, I = J, J < 3, X = Y, Y1 = Y + Y, J1 = J + 1},
        q(I, X, Resultat).
p(X, Resultat) :- {I = 0, I = J, J = 3, X = Y, Resultat = R, Y = R}.

% Suppression de la deuxième clause (inconsistante) et simplification des contraintes
% de la première
p(X, Resultat) :- {I = 0, X1 = X + X, I1 = I + 1}, q(I1, X1, Resultat).

% Deuxième dépliage
p(X, Resultat) :- {I = 0, X1 = X + X, I1 = I + 1, I1 = J1, X1 = Y1,
        Resultat = R, J1 < 3, Y2 = Y1 + Y1, J2 = J1 + 1}, q(J2, Y2, R).
p(X, Resultat) :- {I = 0, X1 = X + X, I1 = I + 1, I1 = J1, X1 = Y1,
        Resultat = R, J1 = 3, Y1 = R}.

% Suppression de la deuxième clause (inconsistante) et simplification des contraintes
% de la première
p(X, Resultat) :- {I = 0, X1 = X + X, I1 = I + 1, X2 = X1 + X1,
        I2 = I1 + 1}, q(I2, X2, Resultat).

% Troisième dépliage
p(X, Resultat) :- {I = 0, X1 = X + X, I1 = I + 1, X2 = X1 + X1,
        I2 = I1 + 1, I2 = J2, X2 = Y2, Resultat = R, J2 < 3,
        Y3 = Y2 + Y2, J3 = J2 + 1}, q(J3, Y3, R).
p(X, Resultat) :- {I = 0, X1 = X + X, I1 = I + 1, X2 = X1 + X1,
        I2 = I1 + 1, I2 = J2, X2 = Y2, Resultat = R, J2 = 3, Y2 = R}.

% Suppression de la deuxième clause (inconsistante) et simplification des contraintes
% de la première
p(X, Resultat) :- {I = 0, X1 = X + X, I1 = I + 1, X2 = X1 + X1,
        I2 = I1 + 1, X3 = X2 + X2, I3 = I2 + 1}, q(I3, X3, R).

% Quatrième dépliage
p(X, Resultat) :- {I = 0, X1 = X + X, I1 = I + 1, X2 = X1 + X1,
        I2 = I1 + 1, X3 = X2 + X2, I3 = I2 + 1, I3 = J3, X3 = Y3,
        Resultat = R, J3 < 3, Y4 = Y3 + Y3, J4 = J3 + 1}, q(J4, Y4, R).
p(X, Resultat) :- {I = 0, X1 = X + X, I1 = I + 1, X2 = X1 + X1,
        I2 = I1 + 1, X3 = X2 + X2, I3 = I2 + 1, I3 = J3, X3 = Y3,
        Resultat = R, J3 = 3, Resultat = Y3}.

% Suppression de la première clause (inconsistante) et simplification des contraintes
% de la deuxième
p(X, Resultat) :- {Resultat = 8 * X}.

% Plus d'appel à deplier : on a fini
```

Dans la portion de code 4.21, le nombre de dépliages successifs est fini. Or il serait possible d'avoir dans un programme une boucle non déroulable de façon finie, ce qui engendrera un processus de dépliage infini. Une amélioration majeure du processus de déroulage de boucles serait donc de la coupler à une détection automatique des boucles « déroulables » de façon finie afin d'éviter des dépliages successifs infinis. Cette détection n'est pas aisée en programmation logique vu les nombreuses formes que peuvent revêtir les boucles. Dans la portion de code 4.21, nous pourrions imaginer de détecter la boucle sur base de son schéma relativement classique (un incrément sur lequel un test est effectué dans les deux clauses de $q/3$). Mais qu'en serait-il si, au lieu de la comparaison $I < 3$, nous avions $I < D$, avec D contraint ailleurs dans le programme ? En d'autres mots, qu'en est-il si la valeur de l'incrément dépend d'une variable ?

Dans le cas où le domaine de calcul est fini, c'est-à-dire si nous travaillons en $CLP(X)$ avec X un ensemble fini, le dépliage pourra générer tous les cas possibles et l'élimination de clauses inconsistantes pourra se faire en

considérant les valeurs possibles dans X. Si par exemple $X = 0..100$, alors si la constante 3 est remplacée par une variable dans le programme d'exemple ci-dessus, le dépliage donnera lieu à 100 nouvelles clauses, dont une seule sera consistante une fois la variable contrainte à une valeur. Dans ce cadre, des optimisations sont possibles : par exemple, si nous savons que, à partir du point d'entrée du programme, dans tous les cas le domaine d'une variable sera contraint d'une certaine façon en arrivant dans la boucle, alors le processus de dépliage (ou plutôt la détection de clauses inconsistantes) peut être adapté en conséquence.

Des efforts en ce sens ont déjà été déployés dans la communauté scientifique pour le cas de la programmation logique classique. La technique de déduction partielle exposée dans l'article [22] construit des arbres de dérivation logique au moyen d'un dépliage dont la finitude est contrôlée. Dérouler une boucle se fait alors par un processus de spécialisation. Nous verrons, dans le chapitre suivant, en quoi consiste la spécialisation, et nous montrerons que des boucles CLP peuvent aussi être déroulées par son biais.

4.6.2 Boucles impératives et boucles logiques

Les boucles sont le cœur de bon nombre d'algorithmes. Il est donc fréquent d'en croiser dans les programmes. Mais leur forme peut différer, notamment à cause de la réécriture intentionnelle de boucles (présentée en début de chapitre). Cela peut compliquer l'approche que nous avons présentée jusqu'ici pour la comparaison de programmes contenant des boucles.

Il serait alors tentant de comparer les boucles entre elles, indépendamment des programmes où elles se trouvent. Cette approche est naturelle puisqu'une boucle représente intuitivement un petit programme à elle seule. Mais si, dans la syntaxe des langages impératifs, il est aisé de repérer les limites d'une boucle (typiquement grâce à des accolades ou à l'indentation), en programmation logique, repérer ces limites est moins évident. Cela dit, la plupart des boucles présentent des schémas récurrents. En programmation logique, une boucle sera typiquement un prédicat récursif défini par plusieurs clauses, certaines représentant les cas de base et les autres les cas récursifs.

Exemple 4.30. *Considérons le programme impératif suivant.*

```
function p(int[] xs) {

        int somme = 0;
        for(x in xs) {
                somme += x;
        }

        int n = somme;
        int y = 1;
        while(n > 1) {
                y *= n;
                n--;
        }

        // suite ...
        // ... (calculs avec y)

        return r;
}
```

Dans une traduction typique et propre — au sens du Prolog exposé dans [7] — de ce code en CLP(ℕ), le prédicat p/n s'appuie sur deux boucles. La première calcule la somme d'une liste d'entiers : elle correspond à la boucle `for` *du programme impératif. La seconde calcule la factorielle d'un entier, comme la boucle* `while`*. Après ces calculs, p/n appelle le prédicat suivant dans le programme, symbolisé par suite/2. Le prédicat p présente un argument R, qui correspond à la valeur de retour de la fonction.*

```
p(Xs, R) :- {AccSomme = 0, AccFacto = 1}, somme(Xs, AccSomme, Somme),
        facto(Somme, AccFacto, Y), suite(Y, R).

somme([], Acc, Somme) :- {Somme = Acc}.
somme([X|Xs], Acc, Somme) :- {Acc1 = X + Acc}, somme(Xs, Acc1, Somme).
```

```
facto(N, Acc, F) :- {N =< 1, F = Acc}.
facto(N, Acc, F) :- {N > 1, Acc1 = N * Acc, N1 = N - 1}, facto(N1, Acc1, F).

% suite ...
```

Une autre traduction, plus littérale, du programme impératif, serait la suivante :

```
p(Xs, R) :- {Somme = 0}, p2(Xs, Somme, R).

p2([], Somme, R) :- {Y = 1}, p3(Somme, Y, R).
p2([X|Xs], Somme, R) :- {Somme1 = Somme + X}, p2(Xs, Somme1, R)

p3(Somme, Y, R) :- {Somme = 0}, suite(Y, R).
p3(Somme, Y, R) :- {Somme = 1}, suite(Y, R).
p3(Somme, Y, R) :- {Somme >= 1, Y1 = Somme * Y, Somme1 = Somme - 1}, p3(Somme1, Y1, R).

% suite ...
```

Dans cette version, les boucles ne sont plus isolées : elles sont intégrées dans le flux du programme, comme c'est le cas dans la version impérative.

L'exemple 4.30 soulève plusieurs questions.

Dans la première version du programme en $CLP(\mathbb{N})$, les calculs de la somme d'une liste et de la factorielle d'un nombre sont des modules à part entière du programme. Elles sont appelées sur un argument et génèrent un résultat, mais leur définition est indépendante du reste du programme. Le style de ce programme est plus classique en programmation logique.

Dans la deuxième version du programme en $CLP(\mathbb{N})$, les boucles sont intégrées dans le flux du programme. Il y est impossible d'appeler le prédicat $p2$ pour calculer la somme d'une liste puisque ce prédicat fait des appels aux prédicats suivants dans le programme en plus d'effectuer ce calcul ; il passe le résultat de la somme au prédicat suivant mais ne le manifeste pas dans ses propres arguments. Bref, le calcul de la somme (et de même pour la factorielle) est effectué de manière récursive mais il est incrusté dans le programme, ce qui se ressent notamment par le fait que les prédicats $p2$ et $p3$ présentent une série d'arguments sans impact sur le calcul mais qui doivent être passés de prédicat en prédicat pour la bonne exécution du programme. Le style de ce programme logique est fortement proche du style impératif, ce qui est moins courant dans le monde de la programmation logique. Mais c'est face à ce genre de programme que nous risquons de nous retrouver, lorsque les programmes manipulés sont issus d'outils de génération de clauses de Horn contraintes tels que [43], [20] et [12].

Supposons à présent que nous cherchions à comparer les deux programmes CLP de l'exemple 4.30. Les mêmes calculs y sont effectués, dans le même ordre, et de la même manière (la somme d'une liste est calculée avec un accumulateur, de même que la factorielle). Mais avec les outils que nous avons à notre disposition, il est impossible de conclure à la connexion des deux programmes. Essayer de ramener le premier programme au deuxième par le biais de pliages est impossible, car ces pliages continueront à contenir des appels vers $somme/3$ et $facto/3$. De même pour toute tentative de dépliage, à cause de la récursivité de ces prédicats. Et réciproquement si nous tentons de rapprocher le deuxième programme du premier, à cause de la récursivité de $p2/3$ et $p3/3$. Du reste, nos autres transformations sont impuissantes dans ces circonstances.

S'il nous sera, en pratique, rare de devoir comparer un programme écrit dans un style logique classique avec un programme écrit dans un style impératif, cela reste une limitation forte de notre approche. La définition d'une transformation permettant par exemple d'extraire les calculs effectués dans une chaîne de prédicats pour les définir indépendamment, ailleurs dans le programme, serait une bonne façon d'adresser cette problématique. En réalité, le pliage réalise déjà cela en partie ; la transformation d'extraction que nous imaginons ici serait une variante du pliage. Nous laissons la définition formelle d'une telle transformation, ainsi que la preuve de sa correction par rapport aux sémantiques adéquates, en travaux futurs. Nous pouvons néanmoins remarquer qu'une telle transformation serait *a priori* préservatrice de sem_2 pour sa faculté à ne pas modifier les points d'entrée du programme, ni les calculs effectués par ceux-ci.

Exemple 4.31. *Soit le programme suivant :*

```
p(X, R) :- {N = X - 2, Acc = 0}, p2(N, Acc, R).
p2(N, Acc, R) :- {N =< 0, Somme = Acc}, suite(Somme, R).
p2(N, Acc, R) :- {N > 0, N1 = N - 1, Acc1 = Acc + N}, p2(N1, Acc1, R).
```

Une transformation d'extraction de boucles pourrait générer le prédicat somme/3 suivant :

```
somme(N, Acc, Somme) :- {N =< 0, Somme = Acc}.
somme(N, Acc, Somme) :- {N > 0, N1 = N - 1, Acc1 = Acc + N}, somme(N1, Acc1, Somme).
```

Le prédicat p2/3 serait altéré de la façon suivante :

```
p2(N, Acc, R) :- {}, somme(N, Acc, Somme), suite(Somme, R).
```

Déplier le prédicat p/2 rendrait le programme comme suit :

```
p(X, R) :- {N = X - 2, Acc = 0}, somme(N, Acc, Somme), suite(Somme, R).
```

ce qui correspond au style d'écriture classique des programmes logiques.

Notons enfin que, quand bien même les calculs seraient ainsi extraits, et donc les structures des programmes similaires, pour peu que les sous-algorithmes employés ne soient pas connectés (par exemple, le calcul de la somme d'une liste de façon récursive d'un côté et avec accumulateur de l'autre), notre approche transformationnelle ne pourra, avec les transformations dont nous disposons, conclure à une connexion entre les algorithmes principaux. Nous revenons sur cette observation lorsque nous abordons les travaux futurs dans le chapitre 6.

4.6.3 Boucles inutiles

Revenons à la problématique des boucles dites « inutiles », insérées dans les programmes, par exemple pour y feindre une structure complexe non existante. Soit c la clause `p_start(X) :- {}, suite(X).` d'un programme écrit en $CLP(\mathbb{N})$. Considérons les deux exemples suivants.

Exemple 4.32.
```
p_start(X) :- {I = 1}, p(I, X).
p(I, X) :- {I = 10}, suite(X). % Cas de base
p(I, X) :- {I < 10, I1 = I + 1}, p(I1, X). % Cas récursif
```

Exemple 4.33.
```
p_start(X) :- {I = 1}, p(I, X).
p(I, X) :- {I = X}, suite(X). % Cas de base
p(I, X) :- {I < X, I1 = I + 1}, p(I1, X). % Cas récursif
```

L'exemple 4.32 présente le cas où une boucle inutile définie sur l'itérateur I est introduite. I y est incrémenté jusqu'à atteindre la valeur constante 10 ; la suite du programme, symbolisée par *suite*/1, est appelée après cette boucle, capturée dans le prédicat $p/2$. Dans ce cas, nous disposons d'une arme capable de supprimer les manipulations inutiles : le tranchage d'arguments en deux passes. En effet, la tranche correspondant à la variable X, seule variable du point d'entrée du programme ne contient pas le premier argument de $p/2$, qui peut être supprimé du programme. Le programme a alors la forme suivante :

```
p_start(X) :- p(X).
p(X) :- {}, suite(X).
p(X) :- {}, p(X).
```

Cette portion de programme se rapproche de c. Le tranchage de la dernière clause (tautologique), suivie par un dépliage de *p_start*/1, achève d'aplanir la différence avec c. Dans l'exemple 4.33, en revanche, le nombre de tours de boucles dépend de l'incrément I qui est contraint, non avec une constante, mais avec la variable X. Pour cette raison, un tranchage ne permettra pas d'éliminer la boucle. Nous ne pouvons alors même pas effectuer de dépliage puisque la variable X peut être une des entrées du programme et il faudra donc attendre l'exécution du programme pour connaître sa valeur. Cette boucle inutile berne donc notre approche.

4.6.4 Syntaxe des boucles

Comme nous l'avons vu, les boucles n'ont, en programmation logique, pas de forme universelle. Une technique qui serait dès lors intéressante est la standardisation syntaxique des boucles, sur base de leur détection préalable dans le programme. Des travaux sur la standardisation des boucles dans les programmes Prolog étendus aux principes de CLP ont été réalisés dans [32]. L'auteur y fait un effort d'extension syntaxique de Prolog de telle façon à ce que les boucles y soient plus clairement identifiées, au moyen de nouveaux mots-clés. L'idée

est de réduire l'effort de lecture des boucles dans les programmes logiques. Si toutes les boucles d'un programme étaient standardisées, il serait plus simple de les repérer, de connaître leur champ d'application et, dans une certaine mesure, de les comparer (sur base syntaxique) entre elles. Une telle procédure pourrait également permettre de détecter et supprimer les boucles n'ayant aucun impact sur le déroulement du programme, en plus de rendre la détection de boucles déroulables plus aisée.

En l'état, la gestion des boucles est une limitation de notre cadre de travail. Il suffit d'introduire une boucle sans intérêt dans un programme pour le déconnecter d'autres programmes recelant pourtant des algorithmes identiques sur le plan sémantique. Le sujet est encore à creuser dans des travaux futurs.

4.7 Stratégie typique de transformation de programmes à comparer

Soit $\mathcal{Y}$ l'ensemble des transformations que nous avons rencontrées jusqu'ici. Nous allons, dans cette section, brièvement revenir sur l'algorithme naïf présenté dans la section 3.8 et en proposer une version heuristique. Cette version prend appui sur des observations que nous avons tirées de l'utilisation de notre prototype. Nous décrivons ci-dessous l'heuristique que nous proposons, et l'illustrons ensuite sur un exemple.

En pratique, il convient d'exploiter au mieux la puissance du dépliage, vu sa capacité d'explicitation des programmes. Mais le processus de dépliage est potentiellement gourmand en temps et en espace pour peu que le nombre de clauses et d'arguments des prédicats soit fort élevé. L'on aura tendance à effectuer un nettoyage au préalable, ainsi qu'un tranchage général, une suppression des clauses inconsistantes et un tranchage des tautologies, afin de tenter de diminuer le nombre de clauses et d'arguments [7]. Le dépliage peut alors être lancé sur le programme, avec comme but de déplier au maximum ce qui peut l'être (c'est-à-dire tout sauf les cycles). Après chaque dépliage, néanmoins, il convient d'effectuer une nouvelle suppression des clauses inconsistantes, à nouveau afin d'éviter une explosion du nombre de clauses à manipuler. Un dépliage peut amener des clauses à être inaccessibles depuis le point d'entrée du programme ; l'on tendra donc à « nettoyer » le programme après un tel dépliage répété. Enfin, une fois les programmes dépliés au maximum, les transformations du chapitre précédent pourront être appliquées de façon guidée pour faire correspondre un programme à l'autre, comme nous l'avons exercé dans la section 3.9. S'il existe des fonctions de renommage, de remplacement d'arguments, de changement d'ordre, de remplacement de contraintes et d'agrégations de clauses qui permettront de mener les deux programmes à une même forme, alors ceux-ci sont $\mathcal{Y}$-connectés. Sinon aucune conclusion n'est possible.

À ce stade, il peut rester des cycles dans les programmes. Comme nous l'avons vu dans la section précédente, nous ne disposons pas d'armes efficaces pour gérer certaines de leurs manifestations. Toutefois, nos transformations seront suffisantes dans les situations où les boucles sont structurellement proches.

Ce que nous avons décrit ci-dessus comprend une **stratégie de transformation**. Une telle stratégie consiste en un ensemble de transformations et de règles à suivre pour leur application. Le but est de transformer un programme d'une façon déterminée pour que le programme résultat présente une certaine forme. Dans notre cas, nous avons cherché à obtenir la forme dépliée « au maximum », tranchée et nettoyée des programmes parce que cette forme est intuitivement plus simple à manipuler pour la comparaison, et présente une forme d'universalité. D'autres stratégies de transformation de programmes CLP, avec d'autres buts et d'autres transformations, sont présentées dans [29] et dans [30].

Nous concluons ce chapitre avec un exemple visant à illustrer notre stratégie de transformation et à argumenter en faveur de celle-ci. Nous allons pour cela observer deux programmes $CLP(\mathbb{N})$, P et Q, respectivement capturés dans les portions de code 4.22 et 4.23.

Portion de code 4.22 – Programme P

```
p_start([A,B,C,D], [A1,B1,C1,D1]):- {},
        tri_bulles(A,B,C,D,A1,B1,C1,D1).

tri_bulles(A,B,C,D,AX3,BX3,CX2,DX) :- {},
        tri_bulles_une_etape(A,B,C,D, AX, BX, CX, DX),
        tri_bulles_une_etape(AX,BX,CX,AX2,BX2,CX2),
        tri_bulles_une_etape(AX2, BX2, AX3, BX3).
```

7. Techniquement, le nettoyage ne changera pas le nombre de clauses affectées par le dépliage, mais il nous semble être une bonne pratique de commencer par « nettoyer » les programmes manipulés.

```
tri_bulles_une_etape(A,B,C,D, MINAB, MINABC, MINABCD, MAXABCD) :- {},
        max_min(A,B,MAXAB, MINAB),
        max_min(MAXAB,C, MAXABC, MINABC),
        max_min(MAXABC, D, MAXABCD, MINABCD).

tri_bulles_une_etape(A,B,C,MINAB, MINABC, MAXABC) :- {},
        max_min(A,B,MAXAB, MINAB),
        max_min(MAXAB,C, MAXABC, MINABC).

tri_bulles_une_etape(A, B, MINAB, MAXAB):- {},
        max_min(A,B,MAXAB, MINAB).

max_min(A,B,B,A):- {A =< B}.
max_min(A,B,A,B):- {A > B}.
```

Portion de code 4.23 – Programme Q

```
p_start([A,B,C,D], [A1,B1,C1,D1]) :- {},
        tri_fusion(A,B, X1, X2),
        tri_fusion(C,D, X3, X4),
        fusion(X1, X2, X3, X4, A1, B1, C1, D1).

tri_fusion(A,B,A,B):- {A < B}.
tri_fusion(A,B,B,A):- {B =< A}.

fusion(X1, X2, X3, X4, MIN13, MIN132, MIN1324, MAX1324):- {},
        tri_fusion(X1, X3, MIN13, MAX13),
        tri_fusion(MAX13, X2, MIN132, MAX132),
        tri_fusion(MAX132, X4, MIN1324, MAX1324).

tri_fusion(X1, Y2, O1, O2).
```

Le programme P est la traduction déclarative d'un **tri de quatre entiers**, basé sur l'algorithme du *tri à bulles*. Le programme Q réalise également le tri de quatre entiers, mais cette fois sur base du *tri par fusion*. Les deux programmes réalisent le même calcul en soi, mais de façon différente : là où le premier simule un tri à bulles sur trois éléments, le second effectue toutes les étapes d'un tri par fusion [8]. Il est évident que pour trier trois entiers, des versions simplifiées de ces programmes existent. Mais ces programmes sont typiquement générés par la **spécialisation** d'algorithmes plus généraux (le tri à bulles et le tri par fusion, en l'occurrence). Nous revenons sur ce concept dans le chapitre suivant.

Essayons de comparer les deux programmes. En suivant la stratégie de transformation exposée ci-dessus, nous devons d'abord éliminer les clauses inaccessibles, inconsistantes, tautologiques ou sans intérêt pour les deux arguments du point d'entrée. Il n'y en a pas ; nous pouvons dès lors passer à l'étape du dépliage. Nous arrivons aux versions suivantes des deux programmes au terme d'un certain nombre de dépliages de $p\_start/2$ accompagnés de suppressions de clauses inconsistantes. Dans les deux cas, les programmes ont pu être déroulés jusqu'à leur point d'entrée.

Portion de code 4.24 – Programme P après dépliage répété

```
p_start([E,F,G,H],[E,F,G,H]):- { F=<G, E=<F, G=<H}.
p_start([F,E,G,H],[E,F,G,H]):- { F=<G, G=<H, F>E}.
p_start([E,G,F,H],[E,F,G,H]):- { E=<F, G=<H, G>F}.
p_start([G,E,F,H],[E,F,G,H]):- { E=<F, G=<H, G>F, G>E}.
p_start([E,F,H,G],[E,F,G,H]):- { F=<G, E=<F, H>G, F=<H}.
p_start([F,E,H,G],[E,F,G,H]):- { F=<G, H>G, F=<H, F>E}.
p_start([E,H,F,G],[E,F,G,H]):- { F=<G, E=<F, H>G, H>F, E=<H}.
p_start([H,E,F,G],[E,F,G,H]):- { F=<G, E=<F, H>G, H>F, H>E}.
p_start([F,G,E,H],[E,F,G,H]):- { F=<G, F>E, G=<H, G>E}.
```

8. Nos implémentations de ces algorithmes de tri sont inspirées de [34].

```
p_start([G,F,E,H],[E,F,G,H]):- { F>E, G=<H, G>F}.
p_start([F,H,E,G],[E,F,G,H]):- { F=<G, F>E, H>G, H>E, F=<H}.
p_start([H,F,E,G],[E,F,G,H]):- { F=<G, F>E, H>G, H>F}.
p_start([E,G,H,F],[E,F,G,H]):- { G>F, E=<G, H>F, G=<H, E=<F}.
p_start([F,G,H,E],[E,F,G,H]):- { G>E, F=<G, H>E, G=<H, F>E}.
p_start([G,E,H,F],[E,F,G,H]):- { G>F, H>F, G=<H, E=<F, G>E}.
p_start([G,F,H,E],[E,F,G,H]):- { H>E, G=<H, F>E, G>F}.
p_start([E,H,G,F],[E,F,G,H]):- { G>F, E=<G, H>G, E=<F, E=<H}.
p_start([F,H,G,E],[E,F,G,H]):- { G>E, F=<G, H>G, F>E, F=<H}.
p_start([H,E,G,F],[E,F,G,H]):- { G>F, E=<G, H>G, E=<F, H>E}.
p_start([H,F,G,E],[E,F,G,H]):- { G>E, F=<G, H>G, F>E, H>F}.
p_start([G,H,E,F],[E,F,G,H]):- { G>F, G>E, H>F, H>E, E=<F, G=<H}.
p_start([G,H,F,E],[E,F,G,H]):- { G>F, H>F, F>E, G=<H}.
p_start([H,G,E,F],[E,F,G,H]):- { G>F, G>E, E=<F, H>G}.
p_start([H,G,F,E],[E,F,G,H]):- { G>F, F>E, H>G}.
```

Portion de code 4.25 – Programme Q après dépliage répété

```
p_start([G,E,H,F],[E,F,G,H]):- { F=<H, E=<G, G<H, E<F, F<G}.
p_start([G,F,H,E],[E,F,G,H]):- { E=<H, G<H, E=<F, F<G}.
p_start([F,E,H,G],[E,F,G,H]):- { E=<F, G<H, E<G, F=<G}.
p_start([G,G,H,E],[E,G,G,H]):- { E=<H, G<H, E=<G}.
p_start([H,E,G,F],[E,F,G,H]):- { F=<G, E=<H, G=<H, E<F, F<H}.
p_start([H,F,G,E],[E,F,G,H]):- { E=<G, G=<H, E=<F, F<H}.
p_start([F,E,H,H],[E,F,H,H]):- { E=<F, E<H, F=<H}.
p_start([H,H,G,E],[E,H,G,H]):- { E=<G, G=<H}.
p_start([E,G,H,F],[E,F,G,H]):- { F=<H, G<H, E<F, F<G}.
p_start([F,G,H,E],[E,F,G,H]):- { E=<H, F<G, G<H, E=<F}.
p_start([E,F,H,G],[E,F,G,H]):- { E<F, G<H, E<G, F=<G}.
p_start([E,H,G,F],[E,F,G,H]):- { F=<G, G=<H, E<F, F<H}.
p_start([F,H,G,E],[E,F,G,H]):- { E=<G, F<H, G=<H, E=<F}.
p_start([E,F,H,H],[E,F,H,H]):- { E<F, E<H, F=<H}.
p_start([G,E,F,H],[E,F,G,H]):- { E=<G, G<H, E<F, F<G}.
p_start([G,F,E,H],[E,F,G,H]):- { E<H, G<H, E=<F, F<G}.
p_start([F,E,G,H],[E,F,G,H]):- { G<H, E=<F, E<G, F=<G}.
p_start([G,G,E,H],[E,G,G,H]):- { E<H, G<H, E=<G}.
p_start([H,E,F,G],[E,F,G,H]):- { F<G, E=<H, G=<H, E<F, F<H}.
p_start([H,F,E,G],[E,F,G,H]):- { E<G, G=<H, E=<F, F<H}.
p_start([H,H,E,G],[E,H,G,H]):- { E<G, G=<H, E=<H}.
p_start([E,G,F,H],[E,F,G,H]):- { G<H, E<F, F<G}.
p_start([F,G,E,H],[E,F,G,H]):- { E<H, F<G, G<H, E=<F}.
p_start([E,F,G,H],[E,F,G,H]):- { G<H, E<F, E<G, F=<G}.
p_start([E,H,F,G],[E,F,G,H]):- { F<G, G=<H, E<F, F<H}.
p_start([F,H,E,G],[E,F,G,H]):- { E<G, F<H, G=<H, E=<F}.
```

Les nombreuses clauses restantes sont les fruits de l'exploration par le programme de toutes les possibilités envisageables. Dans le programme Q, certaines de ces clauses sont incluses les unes dans les autres ; certaines contiennent des contraintes sans impact sur le résultat, vestiges de l'algorithme de tri dont elles sont issues. Une agrégation des clauses judicieuse (pour Q), suivie par un remplacement d'arguments et un remplacement de contraintes sur le résultat de façon à simplifier les contraintes en question (pour P et Q), puis un renommage de variables (pour P et Q), et enfin un changement d'ordre des clauses (pour P et Q) permettent de transformer les deux programmes en un même programme visible dans la portion de code 4.26.

Portion de code 4.26 – Programmes P et Q après application de la stratégie

```
p_start([A,B,C,D],[A,B,C,D]):- {A =< B, B =< C, C =< D}.
p_start([A,B,C,D],[A,C,B,D]):- {A =< C, C =< B, B =< D}.
p_start([A,B,C,D],[A,B,D,C]):- {A =< B, B =< D, D =< C}.
p_start([A,B,C,D],[A,C,D,B]):- {A =< C, C =< D, D =< B}.
p_start([A,B,C,D],[A,D,B,C]):- {A =< D, D =< B, B =< C}.
```

```
p_start([A,B,C,D],[A,D,C,B]):- {A =< D, D =< C, C =< B}.
p_start([A,B,C,D],[B,A,C,D]):- {B =< A, A =< C, C =< D}.
p_start([A,B,C,D],[B,A,D,C]):- {B =< A, A =< D, D =< C}.
p_start([A,B,C,D],[B,C,A,D]):- {B =< C, C =< A, A =< D}.
p_start([A,B,C,D],[B,C,D,A]):- {B =< C, C =< D, D =< A}.
p_start([A,B,C,D],[B,D,A,C]):- {B =< D, D =< A, A =< C}.
p_start([A,B,C,D],[B,D,C,A]):- {B =< D, D =< C, C =< A}.
p_start([A,B,C,D],[C,A,B,D]):- {C =< A, A =< B, B =< D}.
p_start([A,B,C,D],[C,A,D,B]):- {C =< A, A =< D, D =< B}.
p_start([A,B,C,D],[C,B,A,D]):- {C =< B, B =< A, A =< D}.
p_start([A,B,C,D],[C,B,D,A]):- {C =< B, B =< D, D =< A}.
p_start([A,B,C,D],[C,D,A,B]):- {C =< D, D =< A, A =< B}.
p_start([A,B,C,D],[C,D,B,A]):- {C =< D, D =< B, B =< A}.
p_start([A,B,C,D],[D,A,B,C]):- {D =< A, A =< B, B =< C}.
p_start([A,B,C,D],[D,A,C,B]):- {D =< A, A =< C, C =< B}.
p_start([A,B,C,D],[D,B,A,C]):- {D =< B, B =< A, A =< C}.
p_start([A,B,C,D],[D,B,C,A]):- {D =< B, B =< C, C =< A}.
p_start([A,B,C,D],[D,C,A,B]):- {D =< C, C =< A, A =< B}.
p_start([A,B,C,D],[D,C,B,A]):- {D =< C, C =< B, B =< A}.
```

Les vingt-quatre clauses représentent les vingt-quatre possibilités de permutations des quatre valeurs à comparer. P contenait déjà vingt-quatre clauses après les dépliages successifs, là où Q en contenait vingt-six. Remarquons que nous avons conservé des répétitions dans les noms des arguments. Ces clauses pourraient, après un remplacement d'arguments, être agrégées en une seule clause. Par souci de lisibilité, nous garderons le programme tel quel.

La simplicité de ce programme tient au fait que le dépliage, couplé à la suppression de clauses inconsistantes (et, dans ce cas, à deux agrégations de clauses pour Q), mène le programme à sa forme la plus simple. Si, en général, déplier un programme peut en décupler la taille, dans les cas où quelques branches d'exécution seulement sont possibles, le dépliage portera mal son nom.

Notre conclusion est bien que P et Q sont $\mathcal{Y}$-connectés, et même $\mathcal{Y}$-équivalents par rapport à sem_1. Intuitivement, ce résultat n'est pas étonnant, puisque les deux programmes réalisent le même calcul : le tri de quatre éléments. Néanmoins, cette $\mathcal{Y}$-équivalence ne semble pas tenir pour les mêmes algorithmes dans leur version la plus générale (c'est-à-dire pour le tri de n éléments), avec les transformations que nous avons introduites. De fait, le tri à bulles et le tri par fusion sont des algorithmes traditionnellement considérés comme différents ([34]). Cette observation nous conforte donc dans l'idée que notre définition d'équivalence algorithmique tient la route, dans le sens où elle compare des algorithmes et non uniquement des calculs.

Mais est-il dès lors judicieux de juger les versions du tri de quatre éléments connectées alors que les algorithmes de tri généraux ne le sont pas ? Ce phénomène semble être engendré par l'utilisation du dépliage, qui ramène les programmes à une forme plus simple en annihilant dans certains cas leur structure algorithmique. Dans le chapitre suivant, nous investiguons plus en avant la question de l'équivalence algorithmique, notamment en questionnant la puissance du dépliage.

Chapitre 5

Équivalence algorithmique

Nous semblons être arrivés à un ensemble de transformations suffisamment puissant pour la comparaison de programmes simples. Mais sa puissance risque d'être trop marquée dans certaines circonstances. En fin du chapitre précédent, nous avons évoqué l'idée que deux programmes pourtant différents dans leur structure et leur fonctionnement pouvaient être considérés comme connectés par notre cadre de travail, en particulier lorsque le dépliage est une des transformations acceptées par celui-ci.

Dans ce chapitre, nous prenons du recul sur notre approche. Si elle semble assez solide pour comparer des programmes en fonction d'un ensemble de transformations et d'une sémantique donnée, nous questionnons la pertinence des transformations présentées quant à la préservation du cœur algorithmique de ces programmes. Nous traitons en particulier de spécialisation de programmes, et analysons plus en avant le lien entre dépliage et complexité algorithmique. Nous ouvrons cette réflexion vers d'autres possibilités de manifestations des algorithmes au sein des programmes. Enfin, nous étudions ce qu'est exactement un « cœur algorithmique » à la lumière des définitions qui en sont données, directement ou indirectement, dans la littérature.

5.1 Spécialisation de programmes CLP

Nous avons rencontré dans la section 4.7 un exemple où des versions dites spécialisées du tri à bulles et du tri par fusion sont comparées. Dans l'exemple, la spécialisation se faisait sur le nombre d'éléments à trier. Nous n'avions pas affaire à des algorithmes de tri généraux : la liste à trier contenait non un nombre quelconque, mais quatre éléments. Dans cette section, nous allons tenter de formaliser le lien qui existe entre la version générale et les versions spécialisées d'un programme.

Définition 5.1 (Version $\mathcal{Y}$-spécialisée)**.** *Soit $\mathcal{Y}$ un ensemble de transformations. Un programme S est une version $\mathcal{Y}$-spécialisée du programme P si*

- *S est $\mathcal{Y}$-connecté à un programme Q ;*
- *Q est identique au programme P sauf que n variables $V_1, ...V_n$, éventuellement membres de clauses distinctes, se sont vu attribuer une valeur à l'aide d'une contrainte du type $V_i = x_i$ ($i \in 1..n$, $n > 0$).*

La spécialisation d'un programme logique est souvent définie comme l'union de celui-ci et d'une requête adressée à son point d'entrée ([22]). La version spécialisée est alors une version du programme destinée à répondre à une requête particulière : elle se fait par rapport à des arguments qui représentent les entrées du programme (dans le cas du tri, la liste à trier). Or nous avons ici défini la spécialisation au moyen des contraintes du paradigme CLP, non au moyen d'une requête. En réalité, ces contraintes auront le même effet d'instanciation de variables qu'une requête, sauf que toute variable de toute clause peut être contrainte dans un but de spécialisation, et non les seuls arguments du prédicat apparaissant dans la requête (ce prédicat se limitant tout logiquement au point d'entrée du programme). Nous avons donc choisi cette formalisation plus générale et à notre sens plus adaptée à la programmation logique avec contraintes.

La spécialisation a grandement été étudiée au travers de recherches sur l'évaluation partielle. Le processus est traditionnellement appelé *déduction partielle* dans le cas de la programmation logique ([22]). Il s'agit alors typiquement de s'appuyer sur les transformations de pliage et dépliage pour spécialiser un programme. Dans [13], des techniques de déduction partielle sont exposées et leurs apports en efficacité sont démontrés. Il y est prouvé que, quand la déduction partielle est maîtrisée, elle permet d'améliorer drastiquement l'efficacité des programmes spécialisés. D'autres applications de l'évaluation partielle sont discutées en détail dans [19].

Nous allons à présent illustrer le concept de spécialisation sur l'algorithme du tri par fusion. La version générale de cet algorithme (c'est-à-dire pour une liste de taille quelconque) est fournie dans la portion de code 5.1.

Portion de code 5.1 – Programme S : tri par fusion - version générale

```
p_start(Xs, Rs) :- {Xs = [], Rs = []}.
p_start(Xs, Rs) :- {Xs = [X], Rs = [X]}.
p_start(U, S) :- {}, division(U, G, D), p_start(G, SG), p_start(D, SD), fusion(SG, SD, S).

division(Xs, Gs, Ds) :- {Xs = [], Gs = [], Ds = []}.
division(Xs, Gs, Ds) :- {Xs = [X], Gs = [X], Ds = []}.
division(Xs, Gs, Ds) :- {Xs = [G,D|T], Gs = [G|GT], Ds = [D|DT]}, division(T, LT, RT).

fusion(Xs, Ys, Zs ):- {Xs = [], Ys = Zs}.
fusion(Ys, Xs, Zs):- {Ys = Zs}.
fusion(Gs, Ds, Rs) :- {Gs = [G|GT], Ds = [D|DT], Rs = [G|T], G =< D, D2s = [D|DT]},
        fusion(GT, D2s, T).
fusion(Gs, Ds, Rs) :- {Gs = [G|GT], Ds = [D|DT], Rs = [D|T], G > D},  fusion(Gs, RS, T).
```

La spécialisation de ce programme se fait en fixant une valeur à un élément de celui-ci. Par exemple, forcer l'élément en tête de liste à une certaine valeur (ici, 25) se ferait en réécrivant le prédicat $p\_start/2$ de la façon suivante :

```
p_start(Xs, Rs) :- {Xs = [], Rs = []}.
p_start(Xs, Rs) :- {Xs = [X], Rs = [X], X = 25}.
p_start(U, S) :- {U = [25|Us]}, division(U, G, D), p_start(G, SG), p_start(D, SD),
        fusion(SG, SD, S).
```

Seule des listes dont le premier élément est 25 pourront rendre ce prédicat vrai. De la même façon, forcer une longueur de liste en particulier (ici 4) se fait comme suit :

```
p_start(Xs, Rs) :- {Xs = [], Rs = [], Xs = [X, Y, Z, W]}.
p_start(Xs, Rs) :- {Xs = [X], Rs = [X], Xs = [X, Y, Z, W]}.
p_start(U, S) :- {U = [X, Y, Z, W]}, division(U, G, D), p_start(G, SG), p_start(D, SD),
        fusion(SG, SD, S).
```

Clairement, les deux premières clauses contiennent des clauses inconsistantes. La troisième tient encore compte de toutes les tailles de liste. L'outil le plus adapté dont nous disposons pour poursuivre la spécialisation en simplifiant le programme est le dépliage. Couplés à une suppression de clauses inconsistantes et à un remplacement de contraintes, des dépliages successifs génèrent le même programme que celui auquel nous étions arrivés en fin de chapitre précédent, c'est-à-dire le programme qui liste les cas possibles de liste de quatre éléments triée en fonction des relations qui tiennent entre les éléments de la liste de départ. Le programme Q considéré en fin de chapitre précédent est donc $\mathcal{Y}$-connecté à S.

Pour résumer, nous avons été capables d'obtenir une version spécialisée à partir d'un programme de tri général ; à l'aide du dépliage, la version spécialisée peut être fortement simplifiée, et même complètement déroulée, à condition que la donnée contrainte (ici, la taille de la liste) soit un paramètre d'induction. En effet, le cycle présent dans le programme initial sert à prendre en compte toutes les tailles de listes possibles. Fixer une taille de liste rend le programme déterministe et donc totalement dépliable (toujours en présence d'une suppression de clauses inconsistantes). Observons que nous aurions pu contraindre deux variables plutôt qu'une seule, forçant par exemple la liste à avoir une certaine taille *et* à avoir comme premier élément une certaine constante.

La puissance du dépliage n'est plus à démontrer. Mais la spécialisation de programmes telle qu'exposée ci-dessus soulève des questions. En effet, deux algorithmes différents [1], pour peu qu'ils adressent la même problématique (ici le tri), auront des versions spécialisées sur un même paramètre $\mathcal{Y}$-connectées entre elles. Cela peut avoir des implications sur notre cadre de travail.

1. Nous verrons plus loin que la notion d'algorithmes différents n'est pas univoque dans la communauté scientifique. Mais il nous semble raisonnable de considérer ici que le tri à bulles et le tri par fusion sont bien deux algorithmes différents.

Tableau 5.1 – Temps de dépliage du tri à bulle et du tri par fusion en fonction des tailles de listes à trier

Taille de la liste	Tri à bulles	Tri par fusion
2	0.156 s	0.008 s
3	0.662 s	0.366 s
4	4.403 s	1.775 s
5	Erreur de sortie de pile après 55.667 s	4.649 s

D'un côté, l'on pourrait définir une nouvelle connexion, nommée $\mathcal{Y}*$-connexion, plus souple que la $\mathcal{Y}$-connexion. Cette connexion se ferait par rapport à un paramètre des programmes, et serait telle que deux programmes sont connectés si un certain nombre de versions spécialisées de ceux-ci sont $\mathcal{Y}$-connectées entre elles. Ainsi, l'on pourrait conclure que le tri à bulles et le tri par fusion (généraux) sont $\mathcal{Y}*$-connectés pour peu que leurs versions spécialisées pour n tailles de listes différentes soient $\mathcal{Y}$-connectées. Cette nouvelle connexion est donc moins exigeante, dans le sens où elle considère deux programmes comme reliés sémantiquement sur base de leurs versions spécialisées. Néanmoins, une telle connexion peut être utile dans notre cas pour lever le problème des boucles. En effet, la spécialisation sur paramètre d'induction nous permet de nous débarrasser de certains cycles, comme dans l'exemple précédemment explicité. Deux programmes en réalité équivalents sémantiquement, tels que le tri à bulles et le tri par fusion, n'auraient (en l'état actuel du cadre de travail) jamais été considérés comme connectés sans prendre leur comportement spécialisé en compte (la spécialisation se faisant alors sur la taille de la liste).

De l'autre côté, comme nous l'avons déjà signalé, le cœur algorithmique des programmes est potentiellement entièrement annihilé par le dépliage. Dans cette optique, contrairement à la $\mathcal{Y}*$-connexion discutée ci-dessus, une connexion moins laxiste pourrait être utilisée pour le cas où l'on souhaite capturer, non seulement la préservation d'une sémantique, mais aussi celle d'une structure algorithmique, au travers de nos transformations [2]. Une idée intuitive en ce sens serait d'utiliser le méta-paramètre *limite* du dépliage. Pour rappel, ce méta-paramètre fixe le nombre de dépliages autorisés lors d'une comparaison de deux programmes. Le dépliage étant l'acteur principal dans le phénomène de spécialisation, mais aussi une transformation très utile, le cadenasser en jouant sur son méta-paramètre peut être une solution pour limiter sa puissance.

5.2 Dépliage et complexité

L'utilisation de l'outil Prolog développé comme support au présent mémoire met en lumière une particularité du dépliage directement liée à la complexité des algorithmes manipulés. Nous avons déjà vu que, conceptuellement, déplier un tri à bulles pour une liste de taille x reviendra, au niveau du résultat, à déplier un tri par fusion pour une liste de la même taille. Mais l'exécution réelle du dépliage, implémenté de la même façon qu'il est défini dans le chapitre précédent, prendra plus de temps dans un cas que dans l'autre. Déplier un tri à bulles sera plus gourmand en temps que déplier un tri par fusion. L'analyse des dépliages successifs nous pousse à croire que les dépliages sont plus chronophages à cause du nombre de chemins plus importants à explorer dans le cas du tri à bulles. Les clauses, avant d'être éventuellement supprimées par une suppression de clauses inconsistantes, sont alors aussi plus nombreuses dans la mémoire du programme. Le processus de dépliage est donc non seulement plus gourmand en temps, mais aussi en espace. De fait, plus un programme expose de chemins à explorer, plus le dépliage prendra de temps pour ce faire. Ce résultat n'est donc pas étonnant en lui-même. En fait, le dépliage répété d'un programme est directement lié à sa complexité en temps, puisque la complexité en temps est un indicateur du nombre d'opérations élémentaires qui sont exécutées par un algorithme en fonction de la taille de ses entrées ([34]) [3]. Or le tri à bulles « classique » que nous avons implémenté en CLP est connu pour être de complexité en temps $\mathcal{O}(n^2)$ alors que le tri par fusion a une complexité en temps de l'ordre de $\mathcal{O}(n \log n)$ ([34]).

Pour illustrer cela, nous avons comparé le dépliage du tri à bulles pour différentes tailles de liste ; de même pour le tri par fusion. Les temps d'exécution sont repris dans le tableau 5.1. Nous n'avons pas été au-delà de

2. Tout dépend de la définition de structure algorithmique. Nous y reviendrons dans la section 5.4.

3. Différentes nuances de complexité existent, dont les plus connues sont la complexité dans « le pire des cas » et la complexité moyenne ([34]). Dans la suite, nous considérons la complexité dans le pire des cas, plus adaptée à notre étude du dépliage, qui cherche à emprunter tous les chemins possibles, y compris le chemin qui correspond au « pire des cas ».

Tableau 5.2 – Temps de dépliage du calcul de nombres de Fibonacci en deux façons

N	Version naïve	Version ascendante
3	0.109 s	0.191 s
4	0.642 s	0.349 s
5	5.536 s	0.674 s
6	68.730 s	0.890 s
7	1h 9min	1.361 s

5 dans la taille de liste car à partir d'une liste de taille 6, le nombre de dépliages et de suppressions de clauses nécessaires pour le tri à bulles n'était plus supporté par notre outil. Ces temps sont évidemment plus importants que ceux d'exécution de ces algorithmes avec des valeurs fixées. Cela vient du fait qu'ils comprennent également les étapes de suppression de clauses inconsistantes (et donc de détection des inconsistances), ainsi que d'autres ralentissements, dont celui causé par le langage Prolog lui-même. De plus, les chiffres sont approximatifs et varient légèrement (pas de plus d'un demi-seconde) d'une exécution à l'autre. Mais la tendance qui se dessine reste évidente. La suppression de clauses inconsistantes est nécessaire, car elle nous assure que seuls les chemins d'exécution viables sont conservés entre deux étapes et que donc aucune explosion combinatoire n'apparaît à cause de clauses inconsistantes.

Nous pouvons voir que pour une taille de liste de 5, la complexité en espace (due à la place nécessaire pour toutes les clauses générées) ne permet même pas, avec les paramètres de Prolog par défaut, d'obtenir un résultat.

La même comparaison a été faite pour deux implémentations du calcul du n-ième nombre de Fibonacci. La première implémentation est l'implémentation récursive naïve où le calcul du n-ième nombre de Fibonacci est calculée comme la somme du $(n-1)$-ième nombre de Fibonacci additionné du $(n-2)$-ième nombre de Fibonacci, tous deux calculés récursivement de la même façon. La deuxième implémentation fait le calcul de façon ascendante, en construisant le nombre de Fibonacco par des additions successives. Les deux implémentations sont inspirées de [15]. Le tableau 5.2 présente les résultats observés. Il est connu que la première version est de complexité exponentielle alors que la deuxième n'est que de complexité linéaire ([15]). À nouveau, cette tendance se ressent dans les temps d'exécution du dépliage. Le dépliage de programmes CLP ayant un effet similaire à leur exécution, ce résultat est intuitif.

En résumé, même si les versions spécialisées, une fois dépliées, sont un seul et même programme, le processus de dépliage en lui-même est d'autant plus long que l'algorithme sous-jacent est complexe. Le dépliage, quand il est répété, reflète donc la complexité des algorithmes manipulés. Vu sa capacité d'exécuter des programmes CLP, le dépliage est une manière de reporter la complexité d'un algorithme dans une phase de « pré-exécution » du programme. La complexité en temps de l'algorithme se ressentira dans cette pré-exécution. Le programme déplié, quant à lui, aura typiquement (comme dans l'exemple du tri) une complexité d'ordre constant. Dans notre contexte de comparaison d'algorithmes, la complexité d'une version spécialisée — et donc de l'algorithme initial — peut être appréhendée par son dépliage. Si nous faisons l'hypothèse qu'aucune boucle « inutile » n'a été introduite dans les programmes initiaux, le temps nécessaire à leur dépliage peut donc être un paramètre à prendre en compte dans leur comparaison, puisqu'une différence importante à ce niveau implique une différence de complexité, et donc d'algorithme [4].

Cette propriété du dépliage a notamment été exploitée dans [37]. Dans cet article, un algorithme naïf pour la recherche de sous-chaîne de caractères est transformé, par pliages et dépliages successifs, en une variante de l'algorithme de Knuth-Morris-Pratt, célèbre pour sa complexité bien moindre ([37]). La technique utilisée s'appuie sur l'évaluation partielle du programme initial. Dans [13], d'autres cas concrets de déduction partielle sont exposés. Les gains en complexité des programmes spécialisés y sont détaillés, de même que le coût en complexité du processus de pliage et dépliage soutenant la déduction partielle. Comme nous pouvons le voir, spécialisation et complexité sont en pratique des concepts indissociables.

4. Du moins pour certaines définitions d'algorithme. Voir section 5.4.

5.3 Autres manifestations des algorithmes

Les algorithmes peuvent se manifester autrement que par leur seule complexité. Notre outil permet d'isoler, au sein des contraintes, la valeur d'une variable, en propageant les occurrences de celle-ci au sein des autres contraintes. La forme des calculs qui composent la valeur d'une variable peut révéler la structure de l'algorithme sous-jacent.

Exemple 5.2. *Prenons les deux versions du calcul d'un nombre de Fibonacci comparées dans la section précédente, spécialisées pour le calcul du 5e nombre de Fibonacci. Après dépliage, sans effectuer de remplacement de contraintes, la forme du prédicat p_start/1 est la suivante pour la version récursive naïve :*

```
p_start(A):- {B=5, A4=Y3-1, B4=Y3-2, Z3=C4+D4, Y3=B, Z3=A, Q4=O4-1, R4=O4-2, P4=S4+T4, O4=B4,
P4=D4, I4=G4-1, J4=G4-2, H4=K4+L4, G4=A4, H4=C4, Y4=W4-1, Z4=W4-2, X4=A5+B5, W4=Q4, X4=S4, C5<2,
D5=C5, C5=R4, D5=T4, W5=U5-1, X5=U5-2, V5=Y5+Z5, U5=J4, V5=L4, O5=M5-1, P5=M5-2, N5=Q5+R5, M5=I4,
N5=K4, A6<2, B6=A6, A6=Y4, B6=A5, I6<2, J6=I6, I6=Z4, J6=B5, Q6<2, R6=Q6, Q6=W5, R6=Y5, Y6<2,
Z6=Y6, Y6=X5, Z6=Z5, O7<2, P7=O7, O7=P5, P7=R5, K7=I7-1, L7=I7-2, J7=M7+N7, I7=O5, J7=Q5, E8<2,
F8=E8, E8=L7, F8=N7, W7<2, X7=W7, W7=K7, X7=M7}.
```

Et la forme suivante dans la version ascendante :

```
p_start(A):- {B=5, A4=0, B4=1, Y3=B, Z3=A, K4=G4+H4, L4=I4-1, G4=A4, H4=B4, I4=Y3, J4=Z3, U4=Q4+R4,
V4=S4-1, Q4=H4, R4=K4, S4=L4, T4=J4, E5=A5+B5, F5=C5-1, A5=R4, B5=U4, C5=V4, D5=T4, O5=K5+L5,
P5=M5-1, K5=B5, L5=E5, M5=F5, N5=D5, S5=1, R5=T5, Q5=L5, R5=O5, S5=P5, T5=N5}.
```

Notre outil génère le résultat suivant après isolement, par propagations successives, de la valeur de la variable A (correspondant au résultat du calcul) :

```
p_start(A) :- {A=5-1-1-1-1+ (5-1-1-1-2)+ (5-1-1-2)+ (5-1-2-1+ (5-1-2-2))+ (5-2-1-1+ (5-2-1-2)+
(5-2-2)) }.
```

Et le résultat suivant dans la version ascendante :

```
p_start(A) :- {A=1+ (0+1)+ (0+1+ (1+ (0+1)))}.
```

Dans les deux cas, la valeur finale est 5. Mais la forme des contraintes donne une idée de l'algorithme utilisé. Dans la version naïve, la récursivité se ressent au travers des nombreuses contraintes imbriquées. Dans l'autre version, la simplicité des calculs ascendants est évidente.

Si l'on considère que la forme des contraintes a de l'importance sur l'algorithme utilisé, alors le remplacement de contraintes devra être adapté dans notre cadre de travail. La façon précise de déceler la trace d'algorithmes au sein de contraintes « propagées » (et non remplacées) dépasse le cadre de ce mémoire et pourrait être adressée dans de futurs travaux. Il est toutefois intéressant pour nous de noter que la manifestation d'un algorithme ne se limite pas à sa complexité, mais peut aussi se retrouver dans la trace des calculs effectués pour arriver à un résultat.

Dans la même veine, une autre trace structurelle laissée par les algorithmes se situe, non plus au niveau des contraintes, mais au niveau de la structure de l'implémentation. Déplier un programme (général) au maximum ne laisse plus que ses cycles logiques en place ; la plupart des autres prédicats sont évincés par le dépliage. Cela donne lieu à un squelette structurel de programme dont il pourrait être intéressant de faire entrer la forme en compte dans la comparaison avec d'autres programmes. Néanmoins, comme nous l'avons vu dans le chapitre précédent, il convient alors de se méfier des « fausses » structures : par exemple, les boucles inutiles insérées dans un programme afin de le distinguer structurellement d'un autre.

5.4 Équivalence algorithmique dans la littérature

Nous l'avons vu : la construction d'un cadre de travail unique pour la comparaison algorithmique n'est pas évidente, parce que la souplesse à adopter dépend de l'application, mais aussi de l'idée que l'on se fait d'équivalence entre programmes. S'agit-il d'une équivalence purement axée sur la syntaxe des programmes, auquel cas seules quelques transformations de base (changement d'ordre, renommage…) sont à prendre en compte ? S'agit-il plutôt d'une équivalence sémantique dans les calculs réalisés, auquel cas des transformations plus puissantes, telles que le dépliage et le tranchage, peuvent entrer en scène ? Ou s'agit-il de comparer les structures

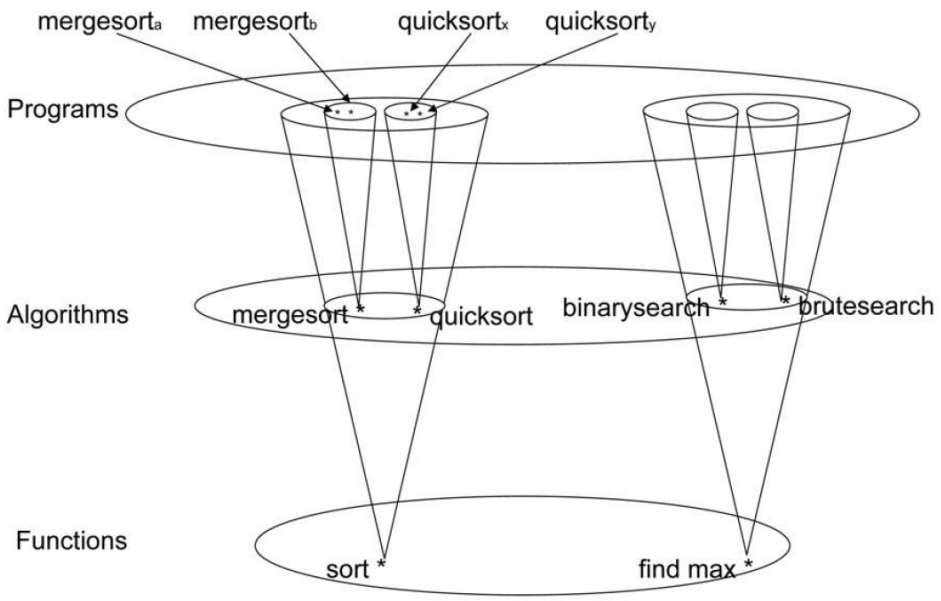

FIGURE 5.1 – Programmes, algorithmes et fonctions ([46])

algorithmiques des programmes, auquel cas certaines transformations, comme le dépliage, doivent pouvoir être limitées dans leurs effets sur ces structures ? Nous avons jusqu'ici défini quatre sémantiques, et donné des pistes d'autres facteurs qui peuvent entrer en scène dans la manifestation des algorithmes. Ces facteurs sont-ils pertinents ? Quelle sémantique est la plus adaptée pour comparer deux algorithmes ? Nous faut-il en définir une nouvelle en ce sens ?

Nous ne cherchons pas ici à donner une réponse univoque à ces questions, pour la simple raison qu'une telle réponse univoque n'existe pas dans la communauté scientifique. Des dissensions sont en effet visibles dans les textes des uns et des autres. C'est aussi pour cette raison que notre cadre de travail est resté le plus général possible. Nous tâcherons dans cette section d'investiguer les tendances qui se dessinent parmi les auteurs qui ont abordé la reconnaissance ou l'équivalence algorithmiques. Le sujet n'est pas nouveau et a été adressé à de nombreuses occasions dans les dernières décennies. Un état de l'art complet dépasserait le cadre de ce mémoire. Pour cette raison, nous ne balayons ici que certaines théories : celles qui nous semblent les plus pertinentes pour raisonner sur notre cadre de travail. Chaque théorie définit ses critères pour l'identification d'un algorithme en particulier, et donne donc un certain sens à la notion d'algorithme.

5.4.1 Classes d'équivalences et mathématiques

Une approche théorique de la notion d'algorithmes est décrite dans [46]. L'auteur se recentre sur le fait qu'un *programme* n'est jamais que l'implémentation d'un certain *algorithme* qui calcule une certaine *fonction*. Différents programmes peuvent implémenter un même algorithme, et différents algorithmes peuvent calculer une même fonction. La figure 5.1 illustre le lien entre fonctions, algorithmes et programmes.

L'on peut remarquer qu'un algorithme est en quelque sorte entièrement déterminé par les programmes qui l'implémentent. Il serait donc pratique de pouvoir calculer la classe de programmes qui implémentent un algorithme donné. Une telle classe serait obtenue à partir d'une relation d'équivalence sur les programmes, $\approx$, telle que $P_1 \approx P_2$ si et seulement si P_1 et P_2 sont deux programmes implémentant un même algorithme. C'est ce qui est recherché dans [46], et c'est également la direction que nous aimerions prendre dans notre cadre de travail.

Nous savons déjà que le problème de déterminer l'équivalence entre deux programmes quant à la fonction qu'ils calculent est indécidable. Le problème de savoir si deux programmes sont équivalents modulo une certaine description (algorithmique) risque de présenter une limitation du même acabit. Cela n'empêche pas les auteurs de [46] de construire une théorie mathématique pour la représentation d'algorithmes et des relations qui les régissent. La première étape est la représentation mathématique des fonctions que les algorithmes calculent, sous forme d'objets mathématiques récursifs organisés en arbres. Puis vient la définition d'opérateurs sur ces fonctions. Ensuite, de nombreuses relations d'équivalence sur ces arbres sont avancées. Ces relations donnent naissance à des classes d'équivalence parmi les programmes, supposées correspondre à un algorithme particulier.

Quoique la problématique initiale soit fort similaire à la nôtre, cette théorie a la particularité d'utiliser des objets mathématiques complexes pour la représentation de programmes et, à travers eux, d'algorithmes. Nous n'entrerons pas ici dans la description des relations présentées, trop nombreuses pour être reprises dans ce mémoire. Nous observerons toutefois que leur auteur ne parvient pas à définir une unique relation d'équivalence algorithmique entre les programmes.

5.4.2 Instance algorithmique

Dans [27], où le but est le remplacement automatique d'algorithmes par des appels de librairies équivalents et plus efficaces, l'équivalence algorithmique entre deux programmes est définie comme étant une relation d'équivalence entre ces deux programmes qui tient si et seulement si l'un peut être obtenu à partir de l'autre moyennant les transformations suivantes :

— renommage de variables ;
— changement d'ordre des instructions, en préservant la sémantique du programme ;
— changement d'ordre des arguments des opérateurs commutatifs.

Cette définition rejoint la notion d'$\mathcal{Y}$-connexion au niveau de la méthodologie transformationnelle qu'elle implique. Elle est raffinée par une procédure permettant de tester si un programme est une *instance algorithmique* d'un autre programme. Cette procédure va, dans l'ordre :

1. comparer les structures de contrôle des programmes ;
2. comparer les instructions simples et les variables qui y sont utilisées ;
3. comparer les conditions (des blocs `if` et `while` typiquement) et les variables qui y sont utilisées ;
4. comparer les incréments des boucles.

La dernière étape est un point important de cette approche, qui considère les incréments comme des valeurs clés dans les algorithmes. Les étapes décrites sont effectuées au moyen d'heuristiques afin d'éviter le même type d'explosion combinatoire que nous risquons en appliquant notre algorithme naïf (voir section 3.8). Cela provient ici du fait que le problème de déterminer si un programme est une instance algorithmique d'un autre programme au sens de [27] est NP-complet.

Nous pouvons voir des liens évidents entre cette théorie et la nôtre. Les différences importantes portent sur les points suivants :

— cette théorie est principalement orientée sur les programmes impératifs, même si elle est conceptuellement indépendante du langage utilisé ;
— elle vise à transformer des programmes dans le but de les optimiser, non de comparer des programmes potentiellement modifiés de façon intentionnelle ;
— le sous-ensemble de transformations qu'elle admet pour l'équivalence algorithmique est fixé, et réduit à quelques transformations seulement.

5.4.3 Valeurs fondamentales

Dans [47], des algorithmes sont comparés dans le but de déceler du plagiat. Au cœur de cette approche est la notion de **valeurs fondamentales** (*core values*). L'idée est de s'abstraire des différentes implémentations possibles des algorithmes, et de tous les artifices de brouillage syntaxique qui peuvent y être insérées intentionnellement, pour capturer les seules valeurs qui sont nécessaires dans toute instance d'un algorithme. Pour les auteurs, les graphes d'appels de fonctions, les graphes de contrôles de flux, et les instructions précises que comportent les programmes n'ont pas d'impact sur l'algorithme sous-jacent. Un algorithme est ici défini comme une série de mises-à-jour, dans un certain ordre, de cases mémoire correspondant aux valeurs fondamentales, afin de générer les valeurs de sortie de l'algorithme. Soient $\mathcal{A}$ l'algorithme d'un programme P_1, et $\mathcal{B}$ l'algorithme d'un programme P_2 suspecté d'avoir plagié l'idée de $\mathcal{A}$. Le *score de similarité algorithmique* entre P_2 et P_1 est calculé comme suit.

1. Identification des valeurs fondamentales :
   — soit par tranchage de P_1, en calculant l'intersection du tranchage avant (tranche des arguments en entrée) et du tranchage arrière (tranche des variables de sortie) ;
   — soit en annotant les variables importantes de P_1, sur base de la connaissance d'experts ;
   — soit en identifiant de façon automatique les propriétés communes entre différentes implémentations de $\mathcal{A}$ (c'est-à-dire entre P_1, Q_1, ..., Q_n, où $\forall i \leq n : Q_i$ est un programme implémentant $\mathcal{A}$).
2. Construction d'une séquence de mises à jour mémoire des valeurs fondamentales et de relations entre ces mises à jour : il s'agit de la *séquence de signature* de $\mathcal{A}$.

3. Calcul du score de similarité entre P_2 et P_1 selon la formule $score = \frac{|séquence\ de\ valeurs\ communes\ entre\ \mathcal{A}\ et\ \mathcal{B}|}{|séquence\ de\ signature\ de\ \mathcal{A}|}$. Dans le cas où l'on connaît d'autres implémentations de $\mathcal{A}$ que P_1, le score peut être affiné par un calcul de moyenne.

Le score donne une indication sur la ressemblance algorithmique de P_1 et P_2 mais la limite inférieure de score nécessaire pour conclure positivement est à adapter en fonction du contexte.

Les différences majeures entre l'approche des valeurs fondamentales et la nôtre sont les suivantes :
— La technique des valeurs fondamentales cherche à abstraire la syntaxe des implémentations, à laquelle nous apportons une importance centrale dans notre méthodologie. Les difficultés d'une comparaison syntaxique sont en effet mises en évidence par les auteurs, et une attention particulière est prêtée à la protection de leurs algorithmes contre les changements d'ordre des instructions, les fissions de variables en plusieurs sous-variables et les optimisations de programmes (entre autres manipulations syntaxiques).
— Dans [47], l'on cherche à expliciter de façon précise le lien entre données d'entrée et de sortie ainsi que les modifications que les premières subissent afin de générer les secondes. Dans notre approche axée sur les programmes logiques avec contraintes, la notion d'entrée et de sortie est plus floue que pour des langages impératifs. En effet, comme nous l'avons discuté dans le chapitre 2, un même argument, en programmation logique, peut servir dans un mode d'entrée ou de sortie. Dès lors, un même programme peut calculer la version triée d'une liste, ou au contraire générer toutes les listes de départ possibles en connaissant une liste triée donnée. Cette différence forte entre programmes logiques et programmes impératifs nous pousserait à peser plus en avant les implications de la définition d'un algorithme comme étant un ensemble de valeurs fondamentales.

5.4.4 Exécution symbolique et déviations

Une amélioration opérationnelle de la technique précédente est présentée dans [28]. Le but y est toujours de détecter des cas de plagiat. Cette fois, l'idée principale est la recherche d'un exemple qui prouve que deux programmes peuvent avoir un comportement différent. Si aucun exemple de ce type ne peut être trouvé, alors les programmes sont considérés comme équivalents, et l'algorithme sous-jacent est plagié. La recherche se fait par la construction de chemins d'exécutions symboliques à partir desquels l'on génère des **déviations** censées varier en fonction des algorithmes. Une déviation entre deux programmes survient lorsque deux entrées génèrent un seul chemin d'exécution symbolique dans un programme alors qu'elles en génèrent plusieurs dans l'autre.

Un des objectifs principaux de cette technique est la résistance à des transformations de programmes malicieuses réalisées de façon automatique, par exemple par des outils de compression. Il est évident que cette approche est similaire à la nôtre dans le sens où elle tâche de ne pas se faire berner par une série de transformations de programmes préservatrices d'une certaine sémantique (en l'occurrence, la relation entre entrées et sorties). Parce qu'elle attache plus d'importance à la sémantique qu'à la syntaxe des programmes, elle est plus adaptée à des programmes de grande taille comportant de nombreux sous-calculs. Pour des programmes plus humbles, elle conclura parfois trop vite à une équivalence alors que des différences algorithmiques majeures séparent les programmes investigués.

5.4.5 Caractéristiques et concepts algorithmiques

La reconnaissance automatique d'algorithmes permet l'identification d'un algorithme au sein d'un programme. Elle est entre autres abordée dans [41] et [26]. Dans ces articles, la reconnaissance d'un certain algorithme se fait au travers de plusieurs de ses caractéristiques, au contraire de l'approche des valeurs fondamentales, où seules les valeurs fondamentales ont de l'importance pour capturer l'essence d'un algorithme.

Dans [41], les caractéristiques définissant un algorithme peuvent être de type numérique (nombre de blocs nécessaires, nombre d'instructions, etc.), ou descriptif (boucle de type incrémentale ou décrémentale, récursivité ou absence de récursivité, nécessité de stockage de valeurs en mémoire ou non...). Un analyseur automatique, qui dispose d'implémentations différentes de plusieurs algorithmes d'une même famille (par exemple les algorithmes de tri) pourra créer un arbre de décision sur base de ces caractéristiques. Dans [26], les caractéristiques d'un algorithme sont organisées de façon hiérarchique. Cette approche plus spécifique est centrée sur l'étude de code Fortran.

Il est important de noter que dans ces approches, les programmes ne doivent pas présenter de valeurs précises pour leurs caractéristiques. Il suffit que leurs caractéristiques soient toutes « proches » des caractéristiques phares de l'algorithme à reconnaître pour que ce soit cet algorithme qui soit reconnu (pour peu que la « proxi-

mité » soit définie de façon à conclure en ce sens).

Dans [2], une technique de réduction des programmes à des équations de récurrence est développée. Cette technique a pour but de reconnaître, dans une implémentation, une instance d'un algorithme plus général en faisant correspondre deux systèmes d'équations de récurrence qui définissent complètement les algorithmes examinés. Les auteurs utilisent pour cela une notion de généralisation des algorithmes : un algorithme général présente alors des algorithmes fils moins généraux, qui calculent une instance de l'algorithme général. Les équations de récurrence présentent alors des particularités communes qu'il est possible de détecter à l'aide d'heuristiques.

Plus éloignée de la reconnaissance d'algorithmes au sens strict est la compréhension de programmes. L'idée est alors de laisser un acteur (humain ou machine) énoncer les concepts qu'il voit apparaître dans un programme dans le but de comprendre ce dernier ([2]). Un algorithme est alors « compris » par l'acteur en question. Mais le problème de « reconnaître des concepts dans un programme et de construire une compréhension du programme en reliant les concepts reconnus à des portions de programmes, à son contexte opérationnel et les uns aux autres », est indécidable en général parce que l'équivalence sémantique de programmes est indécidable ([4]). Il n'est donc pas possible de construire une compréhension de programmes universelle qui expliciterait les algorithmes cachés dans les programmes aux utilisateurs. Une méthode heuristique est présentée dans [14], sous la forme d'un programme capable d'annoter les concepts décelés dans d'autres programmes. Ce processus se fait sur base de *règles* qui définissent des *événements* à comprendre. Plutôt que de reconnaître un algorithme donné, ce système peut expliquer ce que fait une portion de code, et joue donc le rôle d'assistant pour un programmeur désireux de comprendre le comportement d'un programme. C'est alors au programmeur en question d'assimiler ce comportement à un algorithme connu pour, par exemple, détecter du plagiat masqué par un code difficile à appréhender sans ce processus de compréhension automatique de programme.

Ces théories permettent principalement de repérer des algorithmes de petite taille. Dans notre cadre de travail, reconnaître un tel algorithme pourrait entre autres servir à nous abstraire des sous-calculs effectués dans un programme de plus grande taille. Dans l'exemple 4.30, les algorithmes de somme et de factorielle pourraient être reconnus d'un côté et de l'autre, ce qui les empêcherait de berner l'étude du programme plus important dans lequel ils sont définis. Nous revenons sur ce point lorsque nous abordons les travaux futurs dans le chapitre 6.

5.4.6 Impossibilité de l'équivalence algorithmique

Alors que certains tentent de capturer la notion d'algorithme et fondent une relation d'équivalence (souvent heuristique) sur celle-ci, d'autres argumentent en faveur de l'inexistence d'une telle relation d'équivalence en toute généralité.

Dans [6], les auteurs déconstruisent en effet l'idée qu'une définition universelle d'algorithme puisse exister à cause du flou qui entourera cette notion à tout jamais. Ce flou est dû à la subjectivité de la notion d'algorithme, et à l'impossibilité de trancher dans certains débats quant à sa définition. Par exemple, en fonction de l'idée que l'on se fait d'un algorithme, le fait de trier une liste de façon croissante ou décroissante à l'aide du tri par fusion donnera lieu à un ou deux algorithmes ; or les deux positions sont défendables. Un autre élément crucial apporté par [6] est le fait que la définition recherchée d'équivalence algorithmique n'est en réalité pas celle d'une relation d'équivalence. En effet, si la réflexivité et la symétrie peuvent raisonnablement y être validées, la propriété de transitivité n'est pas présente dans cette relation. De nombreux exemples sont introduits en ce sens.

Le cheminement de [6] implique donc qu'il serait impossible de définir des classes d'équivalences de programmes qui correspondraient, non pas aux fonctions qu'ils calculent, mais à la façon dont ils les calculent. La question est connue comme étant l'un des défis de la logique mathématique du vingt-et-unième siècle ([8]). C'est cette même question qui a motivé l'auteur de la recherche [46] présentée dans le paragraphe 5.4.1. Cet auteur mentionne d'ailleurs sa conscience du caractère intrinsèquement subjectif de la définition d'algorithme qu'il y construit.

Tout au long de [6], c'est la notion la plus intuitive et naturelle d'algorithme qui est déconstruite, avec toute la subjectivité qui peut l'entourer. Nous pouvons remarquer que des questions similaires, telles que l'existence d'une relation d'équivalence entre preuves mathématiques, ou entre textes littéraires écrits en différentes langues, sont au même niveau que la question qui concerne les algorithmes, et sont tout autant sujets à débat.

Comme nous l'avons vu au long de cette section, des problèmes indécidables, et des impossibilités de définition, peuvent survenir dans la quête d'une définition d'algorithme. De fait, l'étude des algorithmes est in-

dissociable de la théorie de la calculabilité. C'est pour cette raison que les techniques exposées dans le secteur sont souvent heuristiques. C'est aussi pour cela qu'il semblerait que certains auteurs définissent la notion d'algorithme — ou d'équivalence entre algorithmes — en faisant des choix subjectifs [5].

Notre cadre de travail a été construit dans le but de pouvoir comparer des programmes mais aussi, à travers eux, des algorithmes. Or les théories abordées dans cette section nous ont prouvé qu'il est compliqué — voire impossible — de formaliser la notion d'algorithme sans se faire des ennemis. Pour cette raison, nous laisserons à chaque application le soin d'ajuster notre cadre de travail au juste niveau de souplesse nécessaire pour sa compréhension de la notion d'algorithme. Comme nous l'avons prouvé dans les sections précédentes, nos transformations peuvent être manipulées de différentes manières, en fonction de ce qui importe dans les algorithmes (notamment la complexité, l'ordre des calculs ou la structure des programmes). Nous estimons donc que notre cadre de travail pourra être étendu pour supporter les différentes manifestations des algorithmes en conditions réelles.

5. Nous ne remettons pas leur pertinence en doute pour autant.

Chapitre 6

Conclusions et travaux futurs

Nous arrivons au bout de notre cheminement. Dans ce chapitre, nous en reprenons les résultats essentiels. Nous faisons le point sur ce qu'ils nous ont permis de construire, mais aussi sur les questions qu'ils ont soulevées. Dans les dernières lignes de ce mémoire, nous exposons des possibilités d'extensions du cadre de travail ainsi construit.

6.1 Un cadre de travail nouveau pour comparer des programmes logiques avec contraintes

Les clauses de Horn sont un formalisme abstrait et puissant dans le monde de la programmation. Elles permettent un niveau d'expressivité élevé pour écrire des programmes dans un style logique et déclaratif. Prenant appui sur leur puissance, un nouveau paradigme est né il y a quelques années : celui des clauses de Horn *contraintes*. Nous avons expliqué les fondements de ce paradigme. Il présente la particularité de posséder de nombreuses déclinaisons — autant qu'il y a de *domaines de contraintes*. Nous avons présenté la programmation logique avec contraintes dans une syntaxe proche de celle du Prolog. Nous en avons démontré l'expressivité pour la modélisation d'univers — mathématiques ou non — caractérisés par une certaine famille de fonctions, de relations et de valeurs.

Nous avons ensuite envisagé la création d'un cadre de travail qui cherche à formaliser la similarité qui peut exister entre deux programmes CLP. Notre méthode s'appuie sur un ensemble de transformations de programmes, noté $\mathcal{Y}$, en fonction duquel deux programmes peuvent être *connectés*, voire *équivalents*. La première de ces notions est la plus générale : deux programmes seront connectés s'il est possible de les transformer en un même programme tiers par des pioches successives de transformations de $\mathcal{Y}$. La deuxième notion est plus précise : elle exige que la connexion se fasse par des transformations préservatrices d'une certaine sémantique pour garantir l'équivalence des programmes par rapport à cette sémantique.

Nous avons, en particulier, défini les notions de fonction sémantique, de transformation, de série de transformations et de transformation préservatrice d'une sémantique. Nous avons introduit la sémantique déclarative sem_0 que l'on attribue généralement aux programmes CLP. Nous avons ensuite montré des exemples de transformations qui préservent cette sémantique, ainsi que des exemples de transformations qui nécessitent l'introduction d'une nouvelle nuance dans cette sémantique pour garantir sa préservation. Les transformations introduites incluent des manipulations classiques de programmes logiques telles que le dépliage, le renommage et le changement d'ordre (de clauses, d'arguments). D'autres transformations sont spécifiques au formalisme CLP. L'on citera le remplacement de contraintes ou encore l'agrégation de clauses.

Un algorithme simpliste a été présenté pour expliciter la façon la plus intuitive d'utiliser notre cadre de travail. Nous avons également introduit une stratégie de transformation de programmes heuristique, basée sur des dépliages répétés, qui a pour but d'affiner les problèmes d'efficacité que l'algorithme naïf risque de présenter dans des cas réels. Cette stratégie a été implémentée pour $CLP(\mathbb{N})$ dans un outil de comparaison automatique écrit en Prolog, qui nous a convaincu de son efficacité en situation réelle. La plupart des transformations introduites dans ce mémoire ont été implémentées dans cet outil.

Nous avons vu, au travers d'exemples, qu'en l'état notre cadre de travail présente déjà une manière souple et paramétrable (tant au niveau des transformations que des sémantiques) de comparer bon nombre de programmes logiques avec contraintes. Nous avons aussi vu que l'ensemble de transformations présenté au fil de nos chapitres n'était pas suffisant pour couvrir tous les cas imaginables de connexion entre programmes CLP. Cela arrive

notamment quand les différences syntaxiques sont trop complexes. Notre cadre de travail se veut extensible ; de nouvelles transformations peuvent y être greffées en fonction des applications. De plus, notre approche est, par essence, attachée à la programmation CLP, mais il serait aisé de l'étendre à d'autres paradigmes de programmation. À notre connaissance, un tel cadre de travail n'avait pas encore été formalisé par le passé.

6.2 Des transformations puissantes et paramétrables

Des transformations connues dans le monde de la programmation logique, telles que le dépliage et le tranchage, ont été exposées et illustrées sur des exemples pratiques. L'introduction de chaque transformation a été motivée par au moins une problématique concrète. Ces problématiques ont, pour la plupart, été observées dans des conditions réelles de comparaison de programmes. Certaines sont plus fréquentes dans la programmation impérative ; mais elles se retrouvent alors, sous une forme similaire, dans les programmes déclaratifs issus d'une traduction littérale des programmes impératifs. Une telle traduction peut se faire par un processus semblable à celui décrit dans [20] pour le cas du (*byte*)code Java. Il est dès lors possible de comparer des algorithmes implémentés en Java, ou en d'autres langages impératifs, par le biais de notre outil.

Ces transformations nous ont permis d'introduire des concepts plus poussés comme le déroulage de boucles et la connexion par les tranches. Elles ne sont pas uniquement des mécanismes utiles pour la comparaison de programmes : il s'agit pour certaines de manipulations extrêmement puissantes sur d'autres plans, et dont nous avons dès lors discuté les effets en détail. L'on citera par exemple :

- le dépliage, qui exécute les programmes CLP, et permet le déroulage de boucles, la spécialisation, l'étude de complexité, et est inclus dans de nombreuses stratégies typiques de transformation de programmes ;
- le pliage, également utilisé dans de nombreuses stratégies de transformation de programmes, notamment pour en améliorer l'efficacité ;
- la suppression de clauses inconsistantes, un outil d'amélioration d'efficacité et de lisibilité des programmes logiques ;
- le tranchage général, qui peut être utilisé pour le déboguage, la compréhension de programmes, et même l'isolement des valeurs fondamentales d'un algorithme dans la technique des valeurs fondamentales.

Toutes les transformations introduites, à l'exception de la transformation identité, présentent la particularité de pouvoir être paramétrées. Ce sont leurs paramètres qui déterminent l'effet concret qu'elles infligent au programme sur lequel elles sont appliquées. Un autre niveau de paramétrage, appelé méta-paramétrage, permet de décliner certaines transformations en une série de variantes comportementales. Les paramètres et méta-paramètres que nous avons fournis avec les transformations ne se veulent pas exhaustifs. Chaque transformation peut encore être affinée en fonction des applications. Nous avons, pour notre cas, défini quelques (méta-)paramètres qui nous ont permis de raisonner au mieux sur l'impact des transformations envers les sémantiques présentées.

6.3 Une approche indépendante des sémantiques

Nous avons discuté la pertinence de certaines transformations, et en particulier du dépliage, quant à la préservation des algorithmes incrustés dans les programmes. Il nous semblait en effet intéressant de pouvoir raisonner sur la connexion d'algorithmes et non uniquement d'implémentations. Mais la notion d'algorithme n'est pas gravée dans le marbre : elle dépend des auteurs et des interprétations. Nous avons proposé des adaptations de certaines transformations (à travers leurs méta-paramètres) et du cadre de travail (à travers les sémantiques) pour permettre une comparaison plus fine des algorithmes cachés derrière les programmes, mais n'avons rien imposé à ce niveau pour rester général. En particulier, nous avons abordé la spécialisation, la complexité, et les manifestations structurelles des algorithmes.

Notre approche pourra donc être altérée par l'utilisation d'une sémantique spécifique. En termes impératifs, la sémantique d'un programme peut se limiter à la fonction qui relie ses entrées à ses sorties. Elle peut aussi intégrer des notions telles que les opérations effectuées sur la mémoire du système, ou l'évolution des valeurs d'entrée au cours du programme. Nous avons, au fil de ce mémoire, introduit quatre sémantiques différentes pour les programmes CLP. Elles sont reprises dans le tableau 6.1. Il est évident que ces sémantiques ne sont pas exhaustives. Il y en a autant qu'il y a d'interprétations possibles des programmes logiques.

Chaque nouvelle sémantique introduite dans ce mémoire est moins exigente (ou, autrement dit, plus générale) que la précédente, dans le sens où ses critères pour considérer deux programmes comme équivalents sont plus laxistes. Dépendant des applications, il est ainsi possible de comparer des programmes sur base de la seule fonction qu'ils calculent (sem_2) ou sur base de l'ensemble des prédicats qui les constituent (sem_0). Le choix

Tableau 6.1 – Sémantiques sem_0, sem_1, sem_2 et sem_3

Sémantique	Description
sem_0	Sémantique déclarative algébrique de [18].
sem_1	sem_0 avec la prise en compte d'un point d'entrée. Tout prédicat non accessible depuis ce point d'entrée n'a pas d'importance dans cette sémantique.
sem_2	sem_1 où le sens d'un programme est entièrement défini par son point d'entrée, peu importe la forme du reste du programme.
sem_3^S	sem_2 limitée aux arguments du point d'entrée dont les indices appartiennent à S.

de la sémantique dépend des motivations. En guise d'exemple, nous nous sommes permis l'introduction des sémantiques sem_1, sem_2 et sem_3 en observant que les programmes que nous manipulions disposaient, de par leur origine, d'un point d'entrée. Mais cela n'est pas garanti dans toutes les applications.

6.4 Travaux futurs

Ce mémoire n'est qu'un premier pas dans la direction d'un cadre de travail puissant et réaliste pour la comparaison de programmes logiques avec contraintes. Les premiers travaux à entamer pour avancer dans cette étude sont les suivants.

6.4.1 Gestion des boucles

Les boucles sont le point faible de notre cadre de travail actuel, notamment à cause de la difficulté de repérer les boucles déroulables, les boucles inutiles, et les boucles intégrées dans le flux des programmes logiques. Nous pensons que les premiers travaux à effectuer doivent se concentrer sur la gestion des boucles, forcément nombreuses dans la plupart des cas réels.

La technique de déduction partielle exposée dans [22] permet d'assurer un nombre de dépliages fini pour le déroulage d'une boucle. Mais elle concerne les programmes logiques classiques. De futures recherches pourraient l'étendre au cas de la programmation CLP.

6.4.2 Gestion des sous-algorithmes

Nous avons vu qu'un programme peut utiliser des algorithmes définis en son sein. Mais il serait malavisé de conclure que deux programmes ne sont pas connectés à cause d'une fonction secondaire (par exemple, le calcul de factorielle) implémentée différemment dans l'un et dans l'autre. Des techniques de compréhension de programmes, ou l'utilisation de sémantiques plus larges pour les sous-algorithmes, sont des pistes à explorer pour résister à ce cas de figure, potentiellement très fréquent dans les cas réels.

6.4.3 Nouvelles transformations

Nous n'avons pas introduit toutes les transformations de programmes CLP existantes, même si les principales transformations investiguées dans la littérature ont été présentées.

Il serait intéressant de rechercher d'autres transformations, soit pour combattre des cas particuliers de différences syntaxiques entre deux programmes, soit pour affiner encore la stratégie de la section 4.7. La transformation d'extraction de boucles évoquée de façon théorique dans le paragraphe 4.6.2 serait par exemple un bon ajout à notre cadre de travail puisqu'elle permettrait de résoudre une situation relativement fréquente dans la comparaison de programmes logiques.

La plupart des plagiats en cas réels sont brouillés à l'aide d'outils automatiques ([28]). Une bonne démarche serait d'observer les modifications faites par ces outils pour définir de nouvelles transformations permettant de les renverser.

6.4.4 Gestion des contraintes

En CLP, les contraintes peuvent revêtir de nombreuses formes. Un travail futur intéressant serait la recherche d'une forme normale pour ces contraintes, pour en faciliter la comparaison. Disposer d'une telle forme normale pour $CLP(X)$ serait optimal, mais une forme normale pour des domaines de contraintes fréquents comme $\mathbb{N}$ apporterait déjà une grande avancée à ce niveau.

Nous avons également évoqué le fait que les contraintes des clauses sont typiquement formées d'une certaine façon qui laisse transpirer l'algorithme sous-jacent d'un programme. À l'avenir, investiguer cette question permettrait d'y voir plus clair quant à l'implémentation de cette « propagation de contraintes » que nous avons imaginée dans la section 5.3.

6.4.5 Automatisation

L'algorithme naïf de la section 3.8 présente encore des zones d'ombre. Une version plus fine et clarifiée de cet algorithme, en particulier au niveau de sa condition d'arrêt, serait un pas considérable dans l'automatisation de notre approche.

6.4.6 Extensions du cadre de travail

Notre cadre de travail pourrait encore être étendu des façons suivantes :
— ouverture à d'autres paradigmes et langages de programmation ;
— définition de nouvelles sémantiques et catégorisation des sémantiques par rapport à des sensibilités typiques recherchées dans les applications ;
— définition de nouveaux méta-paramètres pour affiner les transformations ;
— calcul d'un score de similarité entre programmes pour nuancer le résultat des comparaisons ;
— détection systématique de tautologies ;
— ...

Bibliographie

[1] J. J. Alferes, F. Banti, A. Brogi et J. A. Leite : *Semantics for Dynamic Logic Programming : A Principle-Based Approach*, pages 8–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[2] C. Alias et D. Barthou : Algorithm recognition based on demand-driven dataflow analysis. *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, pages 296–305, 2003.

[3] N. Bensaou et I. Guessarian : *Transforming constraint logic programs*, pages 33–46. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[4] T. J. Biggerstaff, B. G. Mitbander et D. E. Webster : Program understanding and the concept assignment problem. *Commun. ACM*, 37(5):72–82, mai 1994.

[5] N. Bjørner, A. Gurfinkel, K. McMillan et A. Rybalchenko : *Horn Clause Solvers for Program Verification*, pages 24–51. Springer International Publishing, Cham, 2015.

[6] A. Blass, N. Dershowitz et Y. Gurevich : When are two algorithms the same ? *Bull. Symbolic Logic*, pages 15(2) :145–168, 2009.

[7] I. Bratko : *Prolog Programming for Artificial Intelligence*. Pearson Education, 2001.

[8] S. R. Buss, A. S. Kechris, A. Pillay et R. A. Shore : The prospects for mathematical logic in the twenty-first century. *CoRR*, cs.LO/0205003, 2002.

[9] C. Dandois et W. Vanhoof : Semantic code clones in logic programs. *E. Albert, editor, Proc. of the 22nd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'12), volume 7844 of LNCS*, pages 35–50, 2012.

[10] E. De Angelis, F. Fioravanti, A. Pettorossi et M. Proietti : *Verification of Imperative Programs by Constraint Logic Program Transformation*, volume 129 de *Electronic Proceedings in Theoretical Computer Science*, pages 186–210. Open Publishing Association, 2013.

[11] E. De Angelis, F. Fioravanti, A. Pettorossi et M. Proietti : *Relational Verification Through Horn Clause Transformation*, pages 147–169. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[12] A. Devos : *Analysis of x86 executables by transformation in the form of Horn clauses*. Université de Namur, mai 2017.

[13] Robert Glück, Jesper Jørgensen, Bern Martens et Morten Heine Sørensen : *Controlling conjunctive partial deduction*, pages 152–166. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[14] M. T. Harandi et J. Q. Ning : Pat : a knowledge-based program analysis tool. *In Proceedings. Conference on Software Maintenance, 1988.*, pages 312–318, Oct 1988.

[15] J.-M. Jacquet : *Techniques de programmation*. Université de Namur, 2016-2017.

[16] J. Jaffar et J.-L. Lassez : Constraint logic programming. *In Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 111–119, New York, NY, USA, 1987. ACM.

[17] J. Jaffar et M. J. Maher : Constraint logic programming : a survey. *The Journal of Logic Programming*, 19:503 – 581, 1994.

[18] J. Jaffar, M. J. Maher, K. Marriott et P. Stuckey : The semantics of constraint logic programs. *The Journal of Logic Programming*, 37(1–3):1 – 46, 1998.

[19] N. D. Jones, C. K. Gomard et P. Sestoft : *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[20] G. Laffineur : *A declarative approach to Java Virtual Machine modelisation and Bytecode execution*. Université de Namur, mai 2017.

[21] J.-L. Lassez et K. McAloon : A canonical form for generalized linear constraints. *Journal of Symbolic Computation*, 13(1):1 – 24, 1992.

[22] M. Leuschel et M. Bruynooghe : Logic program specialisation through partial deduction : Control issues. *Theory and Practice of Logic Programming*, 2(4-5):461–515, 2002.

[23] M. Leuschel et G. Vidal : *Forward Slicing by Conjunctive Partial Deduction and Argument Filtering*, pages 61–76. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[24] M. J. Maher : *Equivalences of logic programs*, pages 410–424. Springer Berlin Heidelberg, Berlin, Heidelberg, 1986.

[25] K. Marriott et P. Stuckey : *Programming with Constraints An Introduction*. The MIT Press, 1998.

[26] B. Di Martino, G. Iannello et H. P. Zima : An automated algorithmic recognition technique to support parallel software development. *In Software Engineering for Parallel and Distributed Systems, 1997. Proceedings., Second International Workshop on*, pages 120–129, May 1997.

[27] R. Metzger et Z. Wen : *Automatic Algorithm Recognition and Replacement*. The MIT Press, 2000.

[28] J. Ming, F. Zhang, D. Wu, P. Liu et S. Zhu : Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection. *IEEE Transactions on Reliability*, 65(4):1647–1664, Dec 2016.

[29] A. Pettorossi et M. Proietti : Rules and strategies for transforming functional and logic programs. *ACM Comput. Surv.*, 28(2):360–414, juin 1996.

[30] M. Proietti et A. Pettorossi : The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *The Journal of Logic Programming*, 16(1):123 – 161, 1993.

[31] T. Reps, S. Horwitz, M. Sagiv et G. Rosay : Speeding up slicing. *SIGSOFT Softw. Eng. Notes*, 19(5):11–20, décembre 1994.

[32] J. Schimpf : Logical loops. *In Proceedings of the 18th International Conference on Logic Programming*, ICLP '02, pages 224–238, London, UK, UK, 2002. Springer-Verlag.

[33] S. Schoenig et M. Ducassé : *A backward slicing algorithm for Prolog*, pages 317–331. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[34] R. Sedgewick et K. Wayne : *Algorithms*. Addison-Wesley Professional, 4th édition, 2011.

[35] V. Senni, A. Pettorossi et M. Proietti : Folding transformation rules for constraint logic programs. 2008.

[36] V. Senni, A. Pettorossi et M. Proietti : A folding rule for eliminating existential variables from constraint logic programs. *Fundam. Inf.*, 96(3):373–393, août 2009.

[37] M. H. Sørensen, R. Glück et N. D. Jones : A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[38] G. Szilágyi, T. Gyimóthy et J. Małuszyński : Static and dynamic slicing of constraint logic programs. *Automated Software Engineering*, 9(1):41–65, 2002.

[39] G. Szilágyi, T. Gyimóthy et J. Małuszyński : Slicing of constraint logic programs. *Computer and Information Science*, 1998.

[40] G. Szilágyi : Learning of constraint logic programs by combining unfolding and slicing techniques. 2009.

[41] A. Taherkhani, L. Malmi et A. Korhonen : Algorithm recognition by static analysis and its application in students' submissions assessment. *In Proceedings of the 8th International Conference on Computing Education Research*, Koli '08, pages 88–91. ACM, 2008.

[42] M. Triska : The power of prolog. `<https://www.metalevel.at/prolog/>`, 2016. [En ligne ; consulté le 30 janvier 2017].

[43] W. Vanhoof, F. Mesnard et E. Payet : Towards a framework for algorithm recognition in binary code. *In Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, PPDP '16, pages 202–213, New York, NY, USA, 2016. ACM.

[44] M. Ward et H. Zedan : Slicing as a program transformation. *ACM Trans. Program. Lang. Syst.*, 29(2), avril 2007.

[45] L. M. Wills : Flexible control for program recognition. *In [1993] Proceedings Working Conference on Reverse Engineering*, pages 134–143, May 1993.

[46] N. S. Yanofsky : Towards a definition of an algorithm. *J. Log. and Comput.*, 21(2):253–286, avril 2011.

[47] F. Zhang, Y.-C. Jhi, D. Wu, P. Liu et S. Zhu : A first step towards algorithm plagiarism detection. *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 111–121, 2012.