UPMC
PARIS UNIVERSITAS

THÈSE DE DOCTORAT DE
L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité : Informatique

(EDITE de Paris)

Présentée par

Julien Pierre MARTIN

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Un Langage de Modélisation à base de Règles pour la Programmation Par Contraintes

Présentée le 8 juillet 2010 devant le jury composé de :

M.	François Clautiaux, Maître de conférences	(Examinateur)
M.	Yves Deville, Professeur	(Rapporteur)
M.	François Fages, Directeur de recherche	(Directeur de thèse)
M.	Daniel Goossens, Maître de conférences	(Examinateur)
M.	Christian Queinnec, Professeur	(Président du jury)
M.	Michel Rueher, Professeur	(Rapporteur)

Université Pierre & Marie Curie - Paris 6
Bureau d'accueil, inscription des doctorants
Esc G, 2ème étage
15 rue de l'école de médecine
75270-PARIS CEDEX 06

Tél. Secrétariat : 01 44 27 28 10
Fax : 01 44 27 23 95
Tél. pour les étudiants de A à EL : 01 44 27 28 07
Tél. pour les étudiants de EM à MON : 01 44 27 28 05
Tél. pour les étudiants de MOO à Z : 01 44 27 28 02
E-mail : scolarite.doctorat@upmc.fr

Thèse préparée à
l'INRIA Paris-Rocquencourt
EPI CONTRAINTES
Domaine de Voluceau, Rocquencourt, BP 105
78 153 LE CHESNAY CEDEX

À Roger
et à Rachel

Résumé

La programmation par contraintes (PPC) est un style de programmation déclaratif qui connaît un grand succès pour la spécification et la résolution de problèmes combinatoires, y compris en milieu industriel. La PPC est fondée sur deux composantes : une composante contraintes et une composante recherche [Van99]. Les langages de modélisation se concentrent sur l'expression des contraintes du problème. Ils fournissent des constructions de haut niveau et une notation algébrique proche de la notation mathématique. Néanmoins, les constructions proposées et les concepts nécessaires peuvent être parfois nombreux et difficiles d'accès au non-programmeur. De plus, l'expression de la recherche et des heuristiques demeure une tâche de programmation spécifique difficile à appréhender pour le modélisateur, ce qui limite l'application de ces principes de programmation.

Cette thèse présente un langage de modélisation général à base de règles, nommé `Cream` (pour *Constraints with Rules to EAse Modelling*), qui a été conçu pour rendre accessible aux non-programmeurs la formulation de problèmes combinatoires et l'intégration de connaissances spécifiques à un domaine d'activité dans des bibliothèques de règles. En pratique, les problèmes réels se réduisent rarement à des problèmes purs, et pour un ingénieur il est souvent plus facile de comprendre ou de décrire une structure complexe en petits fragments plutôt qu'en un tout monolithique. C'est pourquoi le langage est fondé sur la définition de règles et leur organisation en bibliothèques, ce qui permet précisément d'exprimer et de composer facilement des fragments de connaissance dans un domaine métier. Dans le même souci d'accessibilité, le langage adopte des structures de données simples basées sur les enregistrements et les listes données avec des itérateurs, intègre contraintes réifiées et globales, et n'autorise pas les définitions récursives.

Nous montrons de plus que dans ce formalisme de règles, la composante recherche peut être spécifiée de façon déclarative. Plus précisément, les stratégies de branchement y sont définies par des formules logiques et des critères heuristiques complexes d'ordonnancement des sous-formules dissociés des stratégies de branchement peuvent y être définis de façon originale par filtrage sur les parties gauches des règles.

La compilation des modèles `Cream` produit des programmes de contraintes sur domaines finis avec contraintes réifiées et contraintes globales. Le schéma de compilation fondamental, dit statique, est formalisé par un système de réécriture de termes qui procède par expansion des règles du modèle. Nous décrivons une sémantique déclarative pour les modèles `Cream` vis-à-vis de laquelle nous prouvons la correction de la transformation, et nous montrons la confluence, la terminaison et la complexité possiblement exponentielle de cette transformation.

Certains problèmes sont de trop grande taille pour être complètement développés ou dépendent d'une valeur inconnue au moment de la compilation. À l'égard de tels problèmes dont les modèles ne peuvent être compilés selon le schéma statique, nous étudions un deuxième schéma par génération de code procédural qui conserve la structure de règles du modèle et produit du code légèrement moins efficace mais montré linéaire en la taille du modèle originel.

Nous évaluons ensuite l'expressivité du langage et l'efficacité des programmes générés sur des problèmes de placement non seulement académiques mais aussi industriels. Les modèles évalués reposent sur une bibliothèque de modélisation des connaissances en placement appliquée à des problèmes purs et des problèmes de colisage dans l'industrie automobile.

Mots-clefs : programmation par contraintes, langage de modélisation, heuristiques déclaratives, bin packing, colisage.

A Rule-Based Modelling Language for Constraint Programming

Abstract

Constraint programming (CP) is a declarative programming paradigm which aims at specifying and solving combinatorial problems. CP relies on two components: a constraint component and a search component [Van99]. Modelling languages focus on the constraint component. They provide high-level constructions and a mathematical-like notation. However, the constructions and concepts involved may be numerous and not easily accessible to a non-programmer. Moreover, search and heuristics remain a difficult programming task which limits the applicability of these principles on a larger scale.

This thesis presents a general rule-based modelling language, named `Cream` (for Constraints with Rules to EAse Modelling), designed to make easy to use for ingineers the formulation of combinatorial problems and the integration of domain-specific knowledge in libraries.

From a practical point of view, real problems seldom reduce to pure academic problems, and an engineer frequently finds it easier to understand or describe a complex design in small pieces than as a monolithic whole. `Cream` is based on the definition of rules and library of rules which allow to easily express and compose pieces of knowledge. In the same matter of concern, the language adopts simple data-structures based on records, lists given with iterators, integrates reified constraints and global constraints, and prohibits recursive definitions.

Moreover, we show that a rule-based formalism permits to specify in a declarative manner the search component. More precisely, branching strategies can be defined as logical formulae and complex heuristics criteria for ordering sub-formulae may be defined separately from branching strategies by pattern-matching on the rule left-hand sides.

The compilation of `Cream` models yields constraint programs over finite domains with reified and global constraints. The fundamental static compilation scheme is formalised by a term rewriting system that operates by rule expansion of the model. We describe a declarative semantics for `Cream` models with respect to which we prove the correctness of the compilation. We prove the confluence, the termination and an exponential complexity bound for this transformation.

Furthermore, when a problem is too large to be fully expanded or when it depends on a value which is unknown at compile-time, we provide a second scheme generating procedural code. This scheme keeps the rule structure of the model and produces less efficient code but is shown to be linear in the original model size.

We evaluate the expressiveness and the efficiency of generated programs not only on academic but also on real-size industrial bin packing problems. The evaluated models rely on a placement modelling library used on pure problems and real business problems coming from the automotive and logistic industry.

Keywords : constraint programming, modelling language, declarative heuristics specification, bin packing.

Remerciements

Je tiens à remercier mon directeur de thèse, François Fages, pour m'avoir offert la chance de me confronter à des questions stimulantes et enrichissantes et pour m'avoir permis d'aboutir. Je lui sais gré du privilège qu'il m'a accordé en me permettant de travailler avec lui dans un cadre idéal, au sein de l'équipe-projet Contraintes du centre de recherche de l'INRIA Paris-Rocquencourt. Merci de m'avoir fait profiter de son expérience, de sa créativité, de sa rigueur et de son optimisme. Je le remercie également de m'avoir guidé dans mes questionnements et de m'avoir aidé à produire, (ré)organiser et mettre en relief les réponses proposées.

Merci à Yves Deville et à Michel Rueher qui m'ont fait l'honneur de bien vouloir être rapporteurs et qui par leurs commentaires, remarques et critiques m'ont permis d'améliorer la qualité de mon mémoire de thèse. Je remercie tous les autres membres du jury. Merci à François Clautiaux pour sa disponibilité et l'attention qu'il a porté à ma thèse. Merci à Daniel Goossens qui a été le premier à me transmettre le goût de la recherche via les problématiques fondamentales liées à la résolution du problème *SAT* et avec qui je prends toujours beaucoup de plaisir à échanger. Merci aussi de m'avoir fait confiance à deux reprises pour prendre en charge un cours. Mes remerciements vont aussi à Christian Queinnec dont j'ai déjà pu apprécier l'écoute, le support et la bienveillance en sa qualité de directeur de mon école doctorale, l'EDITE de Paris.

Cette thèse doit beaucoup au projet européen Net-WMS qui s'est attaché à traiter des problèmes de colisage issus de l'industrie automobile. Ce projet fut une chance formidable pour concevoir et évaluer un langage de modélisation et de nouvelles techniques de programmation par contraintes. En effet, il a fournit à la fois les spécifications et des instances de problèmes réels et aussi les exigences et les pratiques d'ingénieurs experts en logistique mais non-programmeurs. En outre, j'ai apprécié travailler et me détendre avec tous les membres du projet dont je salue la compétence et la bonne humeur. En particulier, merci à Mats Carlsson du SICS (Uppsala) et à Nicolas Beldiceanu du LINA (Nantes) avec qui j'ai eu l'honneur et le plaisir de travailler plus étroitement. Mats et Nicolas forment une équipe de chercheurs hors pair et la passion qui les anime est communicative. Ils ont été pour moi des exemples de rigueur et de précision.

Pendant ces quatre dernières années (stage de master inclus), j'ai partagé mes journées avec les descendants des irréductibles Voluçois du fameux bâtiment 8. Il s'agit en premier lieu de tous les membres passés et actuels du projet Contraintes que j'ai eu l'occasion de côtoyer. Je remercie particulièrement Thierry Martinez pour ses années d'activité dans notre « groupe de travail ». Nos nombreuses discussions à propos de nos thèses respectives et bien d'autres sujets ont été pour moi extrêmement instructives, fertiles, et plaisantes ; j'espère pouvoir les poursuivre. Merci à Sylvain Soliman pour sa

disponibilité, pour tous ses conseils et ses critiques avisés, pour nos régulières parties de squash et pour avoir su infailliblement animer le quotidien. Ma gratitude s'adresse aussi à Grégory Batt pour nos discussions impromptues certains week-ends travaillés ; à Elisabetta De Maria pour son aménité ; à Pierre Deransart pour m'avoir aidé à me maintenir en forme ; à Steven Gay pour avoir relu spontanément une partie de l'ébauche de ce mémoire ; à Rémy Haemmerlé pour avoir partagé son bureau et son temps lors de mon arrivée ; à Domitille Heitzler, Dragana Jovanovska et Faten Nabli pour leur bonne humeur ; à András Kovács, qui a commencé le projet européen avec nous en tant que post-doctorant, pour son aide et ses encouragements et pour sa générosité et sa gentillesse ; à Aurélien Risk pour n'avoir pas pris la navette de retour sans moi lors des JFPC'08 et enfin à Surinderjeet Singh, Akhil Deshmukh et Curtis Fonger pour avoir été les premiers utilisateurs de `Cream` et avoir permis de rendre son implantation moins boguée.

Merci beaucoup à Nadia Mesrar pour le sérieux et l'amabilité avec lesquels elle m'a toujours soulagé des préoccupations administratives. Merci et bon courage à Roxane Bonin qui lui a succédé. De la même façon, merci à Myriam Brettes et Catherine Verhaeghe pour leur support.

J'aimerais ne pas oublier les autres occupants du bâtiment 8. Merci à Luc Maranget pour avoir répondu volontiers et gentiment à une ou deux questions bêtes de compilation, ainsi que pour nos discussions de toutes natures qui me furent si profitables et plaisantes. Merci à Jade Alglave pour sa solidarité et pour avoir partagé avec moi un certain nombre de ces moments qui tendent à être interdits. Merci à Bernard Lang, Jean-Jacques Lévy, et Pierre Weis pour la transmission d'un fragment de leur savoir et pour leur humour. Merci à Philippe Deschamp de s'être préoccupé de ma santé et d'avoir essayé de réveiller ma fibre artistique. Bien que Benoît Sagot soit devenu rare à Rocquencourt, je n'ai pas oublié son entrain et son goût pour la langue française. Merci à Éric Villemonte de la Clergerie pour son affabilité et pour avoir continué, après mon stage de master dans son équipe, à me faire partager l'avancement de ses travaux de recherche avec tant de passion et de pédagogie. Éric et Xavier Leroy ont été mes rapporteurs internes à l'INRIA, qu'ils en soient remerciés.

Merci à mes amis pour tous nos moments partagés ; qu'il se soit agi d'activités saines ou un peu moins saines, toutes ont su me ressourcer et me rappeler à l'essentiel.

Il serait trop long d'exprimer ici à quel point je suis reconnaissant et redevable envers mes parents et mes soeurs. Je les remercie alors le plus simplement ainsi que le reste de ma famille, entendue au sens propre comme au sens figuré.

Je remercie Ghada pour ce qu'elle est, pour le bonheur qu'elle me procure et pour son indéfectible soutien. Je la remercie pour ce qu'elle me permet d'être et surtout de devenir. Beaucoup de choses de la vie seraient plus ternes, moins savoureuses sans elle et de nombreuses autres que j'apprécie tant n'existeraient simplement pas.

Table des matières

Chapitre 1

Introduction

La *programmation par contraintes* (PPC) est un paradigme de programmation déclaratif qui s'attache à exprimer et à résoudre une grande variété de problèmes combinatoires. Depuis de nombreuses années, la PPC rencontre un grand succès dans la résolution de problèmes concrets de planification, d'ordonnancement, de configuration et de placement. La PPC s'articule autour d'une architecture simple à deux composantes : une composante *contraintes* et une composante *recherche* [Van99]. Un programme de contraintes décrit à la fois la forme des solutions d'un problème à l'aide de relations sur des variables et la stratégie de recherche à adopter pour explorer l'espace des combinaisons.

Les langages de programmation logique avec contraintes (PLC), comme Prolog avec contraintes, sont des représentants anciens et populaires de la PPC qui ont largement participé à son succès. Mais ces langages ne sont pas aussi déclaratifs et d'aussi haut niveau qu'un modélisateur non-programmeur le voudrait. Notamment, ils n'offrent pas de quantificateurs ou d'itérateurs génériques qu'il faut écrire avec des définitions récursives. De plus, ils introduisent des constructions de contrôle qui sortent du paradigme déclaratif. Enfin, la formulation des problèmes et l'expression des stratégies de recherche ne sont pas clairement séparées. Par ailleurs, les langages de PLC ne garantissent pas le terminaison des programmes. La non-terminaison de l'exécution n'est pas une propriété nécessaire pour la modélisation et rend d'autant plus difficile l'accès à ces langages pour les non-programmeurs.

Les *langages de modélisation* répondent aux besoins d'abstraction nécessaires pour se concentrer sur la formulation des problèmes dans un cadre purement déclaratif. OPL, Comet, Zinc et Essence forment l'état de l'art des langages de modélisation. Ils offrent tous des constructions de haut niveau et une notation algébrique proche de la notation mathématique. Parmi ces langages, OPL et Comet permettent à la fois de formuler le problème et, séparément, d'exprimer la stratégie de recherche. OPL est historiquement le premier langage de modélisation à offrir d'un côté un langage de contraintes riche et aussi des constructions de programmation pour écrire la recherche. Comet est un langage de modélisation orienté objet, qui peut être vu comme une évolution d'OPL vers l'hybridation des techniques de PPC et de recherche locale. Ces deux langages sont compilés en du code procédural conservant la structure des modèles (sans expansion des modèles) et la terminaison de la compilation est garantie. Ce code est ensuite exécuté par des machines virtuelles dédiées efficaces mais il n'y a pas de garantie de la terminaison de l'exécution, les modèles pouvant faire usage de boucles « tant que » (`while`) ou

de définitions récursives. Zinc et Essence, quant à eux, génèrent du code « plat » par expansion des modèles et n'offrent pas les moyens d'exprimer les stratégies de recherche. Ils garantissent la terminaison de la compilation et de l'exécution de ses modèles. Le point fort de Zinc est sa compatibilité avec la majorité des solveurs de CSPs. Il permet par ailleurs d'étendre le langage par des définitions utilisateur non récursives. Essence propose des structures très riches (multiensembles, fonctions, relations, partitions, etc.) qui autorisent l'utilisateur à déclarer des variables dont les valeurs sont des objets combinatoires. C'est le compilateur qui prend soin de reformuler et transformer en des CSPs les contraintes des modèles Essence impliquant de tels objets très structurés. Mais ce langage ne peut être étendu par des définitions utilisateur de nouvelles contraintes.

Formuler des problèmes combinatoires reste un art difficile. En pratique, un problème issu de l'industrie est plus riche et sujet au changement que la définition idéalisée étudiée dans le monde académique. Les contraintes rencontrées dans les problèmes industriels sont des règles de bon sens ou des règles dites métier. Ces dernières représentent les connaissances spécifiques d'une entité industrielle sur son domaine d'application, c'est-à-dire son expertise et ses exigences. Pour l'homme de métier, il est évidemment plus facile de comprendre ou de décrire une structure complexe en petits fragments plutôt qu'en un tout monolithique. De ce point de vue, il est essentiel de se donner les moyens d'exprimer de façon simple et compositionnelle les contraintes afin de faciliter l'expression et la modification des connaissances.

Il reste que dans la plupart des cas, une fois modélisés, les problèmes ne sont pas résolus par une stratégie de recherche standard. Alors, sans les moyens de définir et d'expérimenter rapidement une stratégie, les ingénieurs non-experts du domaine ne vont pas plus loin avec la PPC. OPL et Comet offrent une composante pour programmer la recherche, ce qui constitue un obstacle conséquent pour les non-informaticiens.

Une partie des travaux présentés ici a été réalisée dans le cadre du projet Européen Strep FP7 Net-WMS [1] qui traite des problèmes de colisage [FAB+07] dans l'industrie automobile.

Contributions de la thèse

Cette thèse présente un langage de modélisation *à base de règles* pour la programmation par contraintes, baptisé `Cream`, conçu pour rendre accessible aux non-programmeurs la formulation et la résolution de problèmes combinatoires.

Le langage adopte des structures de données simples basées sur les enregistrements et les listes données avec des itérateurs, interdit les définitions multiples et les définitions récursives et garantit la terminaison. La compilation des modèles `Cream` produit des programmes de contraintes sur domaines finis avec contraintes réifiées et contraintes globales comme par exemple la contrainte géométrique `geost` [BCP+07] consacrée aux problèmes de placement en grande dimension.

1. `http://net-wms.ercim.org`

Les contributions de cette thèse sont les suivantes :

1. La conception et l'implantation d'un langage de modélisation par règles, nommé `Cream`[2], permettant d'exprimer des connaissances sur des problèmes spécifiques par fragments plutôt qu'en un tout monolithique, de les intégrer dans des bibliothèques métier, et de les composer.
2. La démonstration que dans un tel langage de modélisation par règles, les heuristiques d'ordre de recherche peuvent être exprimées de manière déclarative, dissociées de l'expression de la stratégie de branchement, par *pattern-matching* sur les têtes de règles.
3. L'étude par un système formel de deux schémas de compilation : un schéma statique fondamental de compilation qui procède par expansion des règles des modèles et un schéma de compilation dynamique par génération de code procédural. La compilation et l'exécution des modèles `Cream` sont garanties de se terminer.
4. La conception d'une bibliothèque de règles dédiée à la modélisation des connaissances en problèmes de placement multi-dimensionnels, et son évaluation sur des problèmes industriels en taille réelle issus du projet européen.

Plan de la thèse

La première partie de la thèse introduit les notions de problèmes de satisfaction de contraintes, de programmation logique par contraintes, et de stratégies de recherche qui sont les fondements de la discipline qui nous intéresse.

Dans le chapitre 3, un rapide état de l'art des langages de modélisation pour la programmation par contraintes montre quelles réponses les auteurs ont apportées aux différents aspects de la problématique de modélisation. Cette revue amène à conclure que ces langages facilitent considérablement le travail des membres de la communauté mais qu'ils restent difficiles d'accès aux ingénieurs non spécialistes en programmation. Le dernier chapitre de la partie présente les principaux types d'exploration de l'espace de recherche et d'heuristiques d'ordonnancement de choix.

La deuxième partie décrit formellement le langage de modélisation `Cream`.

Le chapitre 5 commence avec la présentation de la syntaxe du langage qui introduit les choix des constructions (du langage) et donne des intuitions sur leur interprétation. La section 5.2 définit une sémantique déclarative pour les modèles clos.

Le chapitre suivant (Chap. 6) présente la compilation du langage des modèles par deux schémas qui produisent un code intermédiaire destiné à être transformé en programmes de contraintes avec contraintes réifiées et contraintes globales. Le schéma de compilation fondamental par expansion de règles (Sec. 6.1) définit inductivement un système de réécriture de termes qui développe complètement les définitions du langage et profite d'un mécanisme d'évaluation partielle. La définition est accompagnée de résultats de confluence, de terminaison, de complexité et de correction de la transformation vis-à-vis de la sémantique déclarative. Suit la présentation du schéma par génération

2. Une implantation du compilateur `Cream`, jusqu'à présent nommé `Rules2CP`, est diffusée sur le web depuis 2009 à l'URL suivante : `http://contraintes.inria.fr/rules2cp`

de code procédural (Sec. 6.2) qui conserve la structure de règles du modèle compilé et produit du code moins efficace mais montré linéaire en la taille du modèle original.

Le chapitre 7 montre comment écrire des heuristiques de recherche déclarativement dans un tel langage à base de règles.

La dernière partie de la thèse propose une évaluation de l'expressivité du langage et de l'efficacité des programmes générés.

Le chapitre 8 décrit une bibliothèque dédiée à la modélisation des connaissances en placement fondée sur des relations entre formes en dimensions supérieures. Elle fournit un ensemble de définitions de règles de placement générales associées aux problèmes noyaux et un ensemble de règles spécifiques impliquées dans les problèmes industriels réels.

Enfin, le chapitre 9 présente l'évaluation du langage à la fois sur des problèmes de placement académiques et sur des problèmes réels de colisage dans l'industrie automobile issus du projet Net-WMS.

Première partie

État de l'art

Sommaire

Chapitre 2

Programmation par contraintes (CP)

Sommaire

La programmation par contraintes (PPC), ou *constraint programming* (CP), est un paradigme de programmation déclaratif issu de l'intelligence artificielle, de la programmation logique et de l'optimisation combinatoire. La PPC a apporté des progrès considérables dans la résolution de problémes de planification [VG06], d'ordonnancement [BPN01], de configuration [Jun98] et de placement [Kor04, BCP+07, CJCM08, SO08]. L'intérêt d'un langage déclaratif repose sur le fait que pour un ingénieur non-programmeur, il est souvent plus naturel de décrire une exigence plutôt qu'une séquence d'actions à mener pour la satisfaire, en particulier quand un aspect combinatoire est impliqué. En pratique, la PPC n'est pas aussi abordable et déclarative que le voudraient ses utilisateurs, potentiellement non-programmeurs.

La partition entre le quoi et le comment est fondamentale pour les langages déclaratifs. Ce clivage est exprimé dans le slogan « Algorithme = Logique + Contrôle » de Kowalski qui proposait en 1979 de distinguer deux parties dans un algorithme : la partie logique qui spécifie les connaissances nécessaires à la résolution d'un problème (les contraintes et leur composition dans le cas de la PPC) et la partie contrôle qui décrit comment utiliser cette connaissance pour résoudre le problème (la résolution des contraintes et la recherche en PPC). De ce point de vue, un langage est d'autant plus déclaratif qu'il soulage l'utilisateur de la tâche de contrôle. En outre, une spécification a l'avantage d'être plus simple à maintenir et à communiquer qu'un algorithme. Malgré tout, il est à noter que le contrôle ne disparaît jamais complètement dans les langages déclaratifs et se montre souvent crucial en terme d'efficacité.

La PPC permet d'exprimer la classe de problèmes combinatoires *NP*-complets regroupés sous le nom de problèmes de satisfaction de contraintes (CSP). Un programme de contraintes décrit les propriétés souhaitées des solutions à un problème sous la forme

d'une conjonction de contraintes potentiellement définies inductivement (CLP). La composante recherche de la PPC est considérée dans le chapitre 4.

2.1 Problème d'optimisation combinatoire (COP)

Un *problème d'optimisation combinatoire* consiste à trouver les éléments d'une structure finie S respectant un ensemble de propriétés $\mathcal{P}$ et minimisant (ou maximisant) une fonction f, dite objectif. Les problèmes de décision associés sont souvent des problèmes *NP*-complets.

Définition 1 (Problème de décision combinatoire)**.** *Un problème de décision (i.e. dont la réponse est oui ou non) pose la question de l'existence d'un élément pour lequel la fonction objectif est supérieure (ou inférieure) à une valeur donnée. Il sont souvent* NP-complets*.*

Définition 2 (Problème d'optimisation combinatoire)**.** *Un problème d'optimisation combinatoire p est défini par le triplet $(N, \mathcal{P}, f)$ où N est un ensemble fini, $f : 2^N \mapsto \mathbb{R}$, et $\mathcal{P} \subseteq 2^N$. L'ensemble des solutions d'un problème p est $\{S \in \mathcal{P} \mid f(S) = \max_{S' \in \mathcal{P}} f(S')\}$*

Problèmes *NP*-complets

La théorie de la complexité organise les problèmes algorithmiques par niveau, classe de « difficulté ». Intuitivement, un problème dans la classe *NP* peut être résolu par l'énumération de l'ensemble des solutions possibles puis par un test de validité effectué par un algorithme polynomial.

Définition 3 (Classe P)**.** *Un problème de décision est dans la classe P des problème polynomiaux si et seulement s'il peut être résolu par un algorithme de complexité en temps polynomial en la taille de l'instance.*

Définition 4 (Classe *NP*)**.** *Un problème de décision est dans la classe NP des problèmes non déterministes polynomiaux si et seulement s'il peut être résolu par un algorithme non déterministe polynomial.*

Définition 5 (Problème *NP*-complet)**.** *Un problème NP-complet est un problème de décision dans NP qui permet de résoudre tout autre problème dans NP par un codage polynomial.*

Les problèmes NP-difficiles sont au moins aussi difficiles que les problèmes NP. La classe NP-difficile contient la classe NP et peut être utilisée pour qualifier la complexité d'un problème d'optimisation dont le problème de décision associé est NP-complet.

Problèmes de placement

Considérons le problème classique de placement et sa généralisation à un nombre de dimensions quelconque.

Définition 6 (Problème de Placement 1D)**.** *Un problème de placement unidimensionnel (Bin-Packing Problem) consiste à trouver le plus petit entier m tel que la partition $I_1 \cup \cdots \cup I_m$ de n articles de tailles $s_1, \ldots, s_n$ dans m récipients de taille c respecte la capacité des récipients, c'est-à-dire : $\forall b \in \{1, \ldots, m\}, \sum_{i \in I_b} s_i \leq c$*

Définition 7 (Problèmes de Placement kD)**.** *Le problème de décision de placement k-dimensionnel consiste à déterminer s'il existe un placement de n orthotopes (boîtes) de tailles $(s_1^1, \ldots, s_k^1), \ldots, (s_1^n, \ldots, s_k^n)$ dans b orthotopes de taille $(s_1, \ldots, s_k)$ tel qu'aucune paire d'orthotopes placés ne se chevauche dans les k dimensions à la fois.*

Le problème d'optimisation associé consiste à trouver le placement qui minimise le nombre b de contenants utilisés.

Proposition 1. *Le problème de décision de placement k-dimensionnel est NP-complet.*

2.2 Problème de satisfaction de contraintes (CSP)

De nombreux problèmes combinatoires peuvent être formulés comme des problèmes de satisfaction de contraintes. Informellement, résoudre un CSP consiste à trouver les affectations d'un ensemble de variables à domaines finis qui satisfont un ensemble de contraintes portant sur ces variables.

Définition 8 (Problème de Satisfaction de Contraintes)**.** *Un CSP $\mathcal{P}$ est défini par le triplet (X, D, C) où $X = \{X_1, \ldots, X_n\}$ est un ensemble de variables, $D = \{D_1, \ldots, D_n\}$ est un ensemble de domaines finis tel que $\forall i, X_i \in D_i$, et $C = \{C_1, \ldots, C_m\}$ un ensemble de contraintes. Chaque contrainte C_i est un couple (R_i, S_i), où R_i est une relation $R_i \subseteq D_{i1} \times \cdots \times D_{ik}$ définie sur un sous-ensemble des variables $S_i = \{X_{i1}, \ldots, X_{ik}\}$, nommé portée de C_i, qui détermine tous les n-uplets de valeurs pour $(X_{i1}, \ldots, X_{ik})$ qui sont compatibles entre eux.*

Définition 9 (Réseau de contraintes)**.** *Un réseau de contraintes est un hypergraphe $N = (V, E)$ où chaque noeud de V représente une variable $X_i \in X$, et où chaque hyperarête de E lie les variables S_j apparaissant dans la même contrainte $C_j \in C$.*

Définition 10 (Solution)**.** *Une solution à un CSP $\mathcal{P}$ est une affectation des variables $a : X \to D$ telle que toutes les contraintes soient satisfaites :*

$$\forall C_i \in C, \Big(a(X_{i1}), \ldots, a(X_{ik})\Big) \subseteq R_i$$

Définition 11 (Espace de recherche)**.** *L'espace de recherche désigne l'ensemble des combinaisons possibles de valeurs pour les variables, c'est-à-dire le produit cartésien de leur domaine $D_1 \times \cdots \times D_n$.*

Exemple 1 (Problème des n-reines)**.** *Le problème des n-reines consiste à placer n reines sur un échiquier de taille n, de telle sorte qu'aucune paire de reines ne puisse s'attaquer. Soit $X = \{Q_1, \ldots, Q_n\}$ les n variables du problème, chaque variable Q_i représente la ligne de l'échiquier de la reine placée sur la colonne i. Soit $D = \{1, \ldots, n\}$, le domaine des variables. Les contraintes imposent que chaque reine soit sur une ligne différente (C_1), et qu'aucune paire de reines ne soit sur une même diagonale (C_2 et C_3) :*

- $C_1 = \forall i < j \in \{1, \ldots, n\},\ Q_i \neq Q_j$
- $C_2 = \forall i < j \in \{1, \ldots, n\},\ Q_i \neq Q_j + j - i$
- $C_3 = \forall i < j \in \{1, \ldots, n\},\ Q_i \neq Q_j + i - j$

Résolution de contraintes

Dans un système de contraintes donné, à chaque contrainte primitive est associé un algorithme de propagation qui réduit l'ensemble des valeurs possibles des variables.

Afin de réduire l'ensemble des valeurs possibles pour les variables d'un réseau de contraintes, la PPC se fonde sur des algorithmes génériques de consistance de noeuds, d'arcs et de chemins dans les réseaux [Mac77, MF85, VHDT92]. On distingue dans l'ensembles des contraintes les contraintes binaires et les contraintes n-aires dites globales. Ces dernières exploitent l'efficacité d'algorithmes spécialisés dans le traitement de sous-problèmes récurrents plus structurés.

D'un point de vue opérationnel, étant données une contrainte et les variables y apparaissant, la propagation élimine du domaine de chacune de ces variables les valeurs incompatibles avec la contrainte, compte tenu du domaine des autres. Certaines variables pouvant apparaître dans d'autres contraintes, l'opération est répétée pour ces contraintes, et ainsi de suite jusqu'à atteindre un point fixe.

En général, la propagation n'élimine pas assez de valeurs du domaine des variables pour obtenir une solution et doit être combinée à une procédure de recherche (cf. Chap. 4) dans l'espace des combinaisons possibles. La propagation est itérée jusqu'au point fixe à chaque étape de la recherche.

2.3 Programmation logique avec contraintes (CLP)

Programmation logique

La programmation logique (PL), ou *logic programming* (LP) [CKPR72, Kow74, Col84] est un paradigme de programmation déclaratif. Un programme logique définit inductivement un ensemble de relations ou prédicats par des clauses logiques qui sont exécutées selon un principe de déduction automatique. Un programme logique, comme un programme de contraintes, est une formule écrite dans une logique donnée et son exécution correspond à une recherche de preuve. En programmation logique, incarnée par le langage Prolog [CKPR72, Col84]), le choix du domaine des variables est restreint à l'univers de Herbrand.

Définition 12 (Clause de Horn)**.** *Une* clause *de Horn est une clause qui contient au plus un littéral positif :* $C \vee \neg H_1 \vee \cdots \vee \neg H_n$*, ou de façon équivalente* $(H_1 \wedge \cdots \wedge H_n) \Rightarrow C$ *où* C*,* $H_1 \ldots H_n$ *sont des atomes.*

Définition 13 (Fait)**.** *Un fait est une clause de Horn positive, i.e. qui ne contient qu'un littéral positif et aucun atome négatif, e.g.* C*.*

Définition 14 (But)**.** *Un but est une clause de Horn négative, i.e. qui ne contient que des littéraux négatifs et pas de littéral positif, e.g.* $\neg H_1 \vee \neg H_2$*.*

Une clause Prolog concrète est de la forme C `:-` $H_1, \ldots, H_n$`.` et on appelle tête ou partie gauche de la clause l'atome C et corps ou partie droite de la clause la conjonction $H_1, \ldots, H_n$. Prolog autorise la définition de clauses récursives et la portée des variables

est fixée dans la définition même des clauses de Horn (du premier ordre). On appelle prédicat p, l'ensemble des clauses qui le définissent, *i.e.* l'ensemble des clauses dont le symbole de tête est p. D'un point de vue opérationnel, une clause exprime qu'à partir de la preuve de la conjonction $H_1, \ldots, H_n$, C peut être déduit.

Définition 15 (Programme Prolog)**.** *Un programme logique Prolog est un ensemble de clauses de Horn.*

Le mécanisme d'évaluation d'un programme Prolog, appelé SLD-*résolution*, consiste à prouver qu'un but est une conséquence du programme en construisant un arbre de recherche de preuve. La résolution commence par essayer d'unifier le but à la partie gauche de chaque clause du programme. Les clauses d'un même prédicat représentent des choix et sont exécutées de manière non déterministe. Lorsque l'unification réussit, la résolution est alors appliquée récursivement en prenant comme nouveaux sous-buts la partie droite de la clause sélectionnée, jusqu'à ce que l'ensemble des sous-buts soit vide (succès) ou qu'aucune nouvelle unification ne puisse être faite (échec).

Le contrôle n'est pas exprimé par la SLD-résolution et Prolog adopte la stratégie de recherche en profondeur et à gauche d'abord. Cette stratégie est incomplète (cf. Sec. 4.1) puisqu'une clause récursive peut engendrer un nombre infini d'étapes de résolution. En ce qui concerne le contrôle *et*, à chaque étape de résolution, les sous-buts en conjonction sont sélectionnés de gauche à droite. Pour le contrôle *ou*, les clauses sont essayées dans l'ordre d'apparition dans le programme. Lorsqu'un sous-but mène à un échec, la recherche de preuve *backtrack*, elle revient sur la dernière alternative de clauses et choisit la suivante jusqu'à épuisement. Prolog n'est pas purement déclaratif, il fournit un opérateur de contrôle, nommé *cut*. Cet opérateur extra-logique, dont nous n'illustrons pas l'usage ici, permet au programmeur d'élaguer un arbre de recherche, c'est-à-dire de ne pas parcourir une partie de l'arbre qu'il sait ne pas mener à succès.

Exemple 2. *Considérons le programme Prolog suivant, dédié à la généalogie.*

Faits

Soit Pierre le père de Paul, Paul le père de Jacques et de Marie et Marie la mère de Jeanne. Cet arbre généalogique est décrit par l'ensemble de faits suivants :

```
pere(pierre, paul). pere(paul, jacques).
pere(paul, marie). mere(marie, jeanne).
```

Clauses

Les deux clauses suivantes définissent le prédicat `parent/2` *en exprimant qu'un individu* `X` *est le parent d'un individu* `Y` *si* `X` *est le père de* `Y` *ou si* `X` *la mère de* `Y`*.*

```
parent(X, Y) :- pere(X, Y).
parent(X, Y) :- mere(X, Y).
```

Enfin, le prédicat `ancetre/2` *est défini récursivement comme suit :* `X` *est ancêtre de lui-même ou bien* `X` *est l'ancêtre de* `Y` *si* `X` *est parent d'un tiers* `Z` *qui est lui-même ancêtre de* `Y`*.*

```
ancetre(X, X).
ancetre(X, Y) :- parent(X, Z), ancetre(Z, Y).
```

But et résolution

Les preuves du but suivant déterminent tous les ancêtres `X` *de Jeanne.*

```
?- ancetre(X, jeanne).
```

Selon le programme, tous les individus sont des ancêtres de Jeanne. Ce but accepte donc plusieurs preuves, chacune donnant une valeur différente à `X` *qu'on appellera solution. Les deux premières solutions et arbres de preuve associés sont les suivants :*

1. `X = jeanne`
   Cette première solution est obtenue simplement par unification du but avec la première clause du prédicat `ancetre/2`*.*

```
------------------------
ancetre(jeanne, jeanne)
```

2. `X = paul`
   C'est le deuxième choix possible pour le prédicat `ancetre/2` *qui amène à cette deuxième solution. Le but est donc ici unifié à la partie gauche de la deuxième clause du prédicat* `ancetre/2`*.*
   La résolution par cette clause produit deux nouveaux buts intermédiaires à prouver : `parent(X, Z)` *et* `ancetre(Z, jeanne)`*.*
   Le premier sous-but `parent(X, Z)` *est prouvé par le fait* `pere(paul, marie)` *unifiant* `X` *à* `paul` *et* `Z` *à* `marie`*. Il reste donc à prouver* `ancetre(marie, jeanne)`*.*
   Le deuxième sous-but `ancetre(marie, jeanne)` *est prouvé par* `parent(marie, jeanne)` *(prouvé par le fait* `pere(paul, marie)`*) et* `ancetre(jeanne, jeanne)`

```
                         ----------------------
                         mere(marie, jeanne)
 -------------------     ----------------------    ------------------------
 pere(paul, marie)       parent(marie, jeanne)     ancetre(jeanne, jeanne)
---------------------    ----------------------------------------------------
parent(paul, marie)                    ancetre(marie, jeanne)
-----------------------------------------------------------------------------
                         ancetre(paul, jeanne)
```

Programmation logique avec contraintes

La *programmation logique par contraintes* (PLC), ou *constraint logic programming* (CLP) [JL87, JM94] est un développement de la programmation logique qui généralise le problème de la résolution de l'égalité entre termes du premier ordre (l'unification) au problème de la résolution de contraintes sur un domaine fixé, par exemple les contraintes linéaires sur le nombres rationnels. De nombreuses implantations de langages de PLC ont vues le jour, depuis Prolog III [Col87] et Prolog IV [Col96] jusqu'à Gnu-Prolog [DC01], Eclipse [AW07] ou `SICStus-Prolog` [C^{+}07]. Comme en programmation logique, l'exécution de toute autre stratégie de recherche que la recherche en profondeur et à gauche d'abord nécessite de programmer un méta-interpréteur [AW07].

La programmation logique par contraintes combine en quelque sorte le meilleur de la programmation logique et de la recherche opérationnelle : un langage déclaratif qui permet d'exploiter une algorithmique spécifique à certains domaines de calcul. À chaque contrainte primitive est associé un algorithme de propagation prédéfini qui élimine une partie des valeurs du domaine des variables et pour atteindre une solution une procédure de recherche doit être définie. La PLC reprend le schéma d'exécution de la programmation logique et offre des constructions non déterministes supplémentaires pour écrire de telles stratégies de recherche.

La PLC a connu plusieurs succès dans différents domaines d'application en commençant par la conception et l'analyse de circuits électroniques [JM94, Wal96]. La PLC a aussi été appliquée pour la résolution de problèmes de planification, d'ordonnancement et de placement.

Exemple 3. *Considérons le programme* ***SICStus-Prolog*** *du problème des n-reines (cf. Sec. 2.2).*

La bibliothèque CLP($\mathcal{FD}$) relative au système de contraintes sur domaines finis est importée.

```
:- use_module(library(clpfd)).
```

La clause principale du programme est `queens(N, L)`*. Elle génère une liste de* `N` *variables qui représentent le placement des n reines sur les lignes de l'échiquier (chaque reine est associée à une colonne). Puis elle initialise le domaine de chaque variable à l'intervalle allant de* `1` *à* `N`*, et fait appel au prédicat* `all_safe/1` *qui pose les contraintes et au prédicat* `labeling/1` *qui s'occupe de la recherche d'une solution.*

```
queens(N, L) :-
  length(L, N), domain(L, 1, N),
  all_safe(L, 1), labeling(L).
```

Le prédicat `all_safe/2` *est une définition récursive qui code une quantification universelle sur la liste des variables du problème. Chaque itération de* `all_safe/2` *consomme un élément de la liste, la tête de liste* `X`*, et appelle* `all_safe_2/4` *qui code aussi une quantification universelle mais sur la queue de la liste* `Xs`*. Chaque itération de* `all_safe_2/4` *fait appel au prédicat* `safe/3`*.*

```
all_safe([], _).
all_safe([X | Xs], I) :-
  J is I + 1,
  all_safe_2(X, Xs, I, J),
  all_safe(Xs, J).

all_safe_2(_, [], _, _).
all_safe_2(X, [Y | Ys], I, J) :-
  safe(X, Y, I, J),
  J1 is J + 1,
  all_safe_2(X, Ys, I, J1).
```

Le prédicat `safe/4` *pose effectivement les contraintes de non-capture entre un paire de reines donnée (*`Q1`*,* `Q2`*) grâce à la contrainte binaire de non-égalité* `#\=` *entre variables de domaines finis.*

```
safe(Q1, Q2, I, J) :-
  Q1 #\= Q2, Q1 #\= Q2 + J - I, Q1 #\= Q2 + I - J.
```

Le prédicat `labeling/1` *définit la recherche de solution. Cette clause récursive développe l'arbre élémentaire des affectations possibles des variables du problème (cf. Chap. 4). En effet, pour chaque variable* `H`*,* `labeling/1` *appelle* `indomain(H)` *de la bibliothèque CLP(*$\mathcal{FD}$*) qui énumère de façon non déterministe l'ensemble des valeurs du domaine de* `H`*.*

```
labeling([]).
labeling([H | T]):-
  indomain(H),
  labeling(T).
```

Du point de vue du modélisateur non-programmeur, la PLC n'est pas aussi déclarative et d'aussi haut niveau qu'il le voudrait. La PPC propose de dissocier la tâche de modélisation des problèmes, qui doit être purement déclarative, de la tâche de recherche. Or, la PLC hérite de la programmation logique l'opérateur de contrôle non déclaratif *cut* et souffre d'un manque de séparation entre les deux composantes contraintes et recherche. Les contraintes et les stratégies de recherche sont définies dans un ensemble uniforme de clauses qui peuvent faire usage d'opérateurs extra-logiques. Par ailleurs, des constructions logiques basiques comme les quantificateurs doivent être codées avec des clauses récursives, ce qui n'est pas naturel pour un non-programmeur. Nous comparons les langages de PLC avec les langages dits de modélisation dans le chapitre 3.

2.4 Programmation impérative avec contraintes (CIP)

La *programmation impérative avec contraintes* (PIC) [AS99], ou *constraint imperative programming* (CIP), est le résultat de la combinaison des techniques de la programmation par contraintes et de la programmation impérative (et orientée objet).

La PIC est une réponse aux préoccupations d'ingénierie logicielle et de diffusion de la PPC dans la communauté des programmeurs. En effet, la plupart des programmeurs sont formés et rompus à la programmation impérative qui jouit d'une littérature abondante, dont les différents langages sont largement diffusés, réputés performants et viennent avec de nombreuses bibliothèques en standard et des environnements de développement évolués. Cette approche de la PPC fait un pas vers les langages impératifs en y embarquant un système de contraintes pour convaincre l'industrie d'utiliser les techniques de PPC.

Le projet Alma [AS99] est un exemple de langage de PIC et il existe par ailleurs un certain nombres de bibliothèques de contraintes implantées dans des langages impératifs comme Ilog Solver [ILO] et Gecode [SLM] en C++ ou Choco [Lab00] en JAVA.

Les langages de PIC sont très puissants mais proposent des constructions de programmation de bas niveau et, comme les langages de PLC, ils ne distinguent pas clairement composante contraintes et composante recherche. Nous proposons de les comparer aux langages dits de modélisation dans la chapitre 3.

Conclusion

La PPC est un modèle de langage déclaratif qui fournit un cadre théorique élégant et unificateur pour exprimer et résoudre une grande variété de problèmes d'optimisation combinatoire.

La programmation impérative avec contraintes fait le choix d'embarquer un système de contraintes au sein d'un langage de programmation impérative pour encourager l'usage de la PPC dans la communauté des programmeurs. Cependant, le bon modélisateur n'est pas nécessairement programmeur, et la formulation d'un problème se fait mieux dans un cadre purement déclaratif. En ce sens, la programmation impérative avec contraintes n'est pas une incarnation idéale de la PPC.

La programmation logique par contraintes est un représentant historique et populaire de la PPC et elle est appliquée avec succès depuis longtemps dans de nombreux domaines d'activité. Elle intègre dans un langage logique les deux composantes de la PPC : un langage de contraintes riche, où chaque contrainte est associée à un propagateur efficace ; et des constructions non déterministes qui soulagent le programmeur pour le développement des stratégies de recherche. Mais du point de vue du modélisateur encore, la PLC souffre d'un manque d'abstraction et n'est pas assez déclarative. En particulier, les langages de programmation logique n'offrent pas certaines constructions essentielles à la modélisation comme par exemple les quantificateurs. Par ailleurs, la composante formulation des problèmes et la composante recherche de solutions ne sont pas clairement distingués au sein de l'ensemble des clauses. Enfin, la programmation d'une stratégie de recherche particulière peut nécessiter l'écriture d'un méta-interpréteur.

Chapitre 3

Langages de modélisation

Sommaire

Les langages de modélisation répondent à un besoin d'abstraction conceptuelle et méthodologique pour formuler des problèmes d'optimisation combinatoire dans un cadre purement déclaratif. OPL, Comet, Zinc et Essence constituent l'état de l'art. Ils fournissent tous une notation algébrique et différentes structures de haut niveau d'abstraction. OPL propose, et de manière dissociée de la formulation, les moyens de programmer des stratégies de branchement et des heuristiques d'ordonnancement (cf. Sec. et 4.3). Comet, évolution d'OPL, donne en plus les moyens à l'utilisateur de programmer ses propres parcours d'arbres à l'aide de fermetures et de continuations.

Nous décrivons dans ce chapitre les traits spécifiques aux différents langages de modélisation de l'état de l'art et donnons pour chacun d'entre eux le modèle du problème des n-reines correspondant. Les deux dernières sections montrent pourquoi nous considérons pas comme des langages de modélisation les langages de programmation logique (ou impérative) avec contraintes et les langages des systèmes de gestion des règles métier.

Un tableau de comparaison qualitative

Afin de comparer et de différencier ces langages de manière synthétique, nous proposons un tableau récapitulatif de 13 traits bipartis en caractéristiques des langages et caractéristiques de leur compilation et de l'exécution des programmes générés.

Les **caractéristiques des langages** sont partitionnées en caractéristiques relatives à la composante contraintes et à la composante recherche.

Composante contraintes
- *extensible* : le langage permet à l'utilisateur de définir de nouvelles contraintes.
- *règles* : les définitions de contraintes reposent sur le formalisme de règles.
- *modulaire* : le langage est doté d'un système de modules permettant d'organiser les définitions utilisateur.

Composante recherche
- *stratégie de branchement* : le langage permet à l'utilisateur d'écrire ses propres stratégies de branchement, de décider des contraintes (en disjonction et en conjonction) posées lors de la recherche (par exemple énumération des affectations possibles, le *labeling*, ou découpage de domaines, le *domain-splitting*, etc ; cf sections 4.2 et 4.3).
- *heuristiques d'ordonnancement* : l'utilisateur peut écrire ses propres heuristiques d'ordonnancement des choix (conjonctifs, par exemple *first-fail* ; et disjonctifs, par exemple *middle-out*).
- *parcours d'arbres* : le langage permet à l'utilisateur d'écrire ses propres parcours d'arbres de recherche (par exemple une recherche par approfondissements successifs, *IDDFS*, ou une recherche avec écarts limités, *LDS*, etc.).

Les **caractéristiques de la compilation et de l'exécution** des programmes générés sont partitionnées en caractéristiques relatives à la compilation (schéma adopté et vérification de types), à l'automatisation de certains choix de modélisation et à l'exécution des programmes générés :

Compilation
- *statique* : les modèles peuvent êtres compilés selon le schéma par expansion des expressions.
- *dynamique* : les modèles peuvent êtres compilés selon le schéma par génération de code procédural qui conserve la structure de définitions des modèles.
- *typage* : le langage est muni d'un système de types et la compilation vérifie la validité des expressions des modèles vis-à-vis du système de type afin d'éviter à l'utilisateur de mauvais usages des structures de données.
- *reformulation* : la compilation peut reformuler le modèle, faire des choix de modélisation et des optimisations pour l'utilisateur.

Exécution, résolution des modèles
- *hybridation* : les modèles peuvent faire coopérer plusieurs techniques de résolution différentes (par exemple recherche globale avec contraintes et recherche locale).
- *multi-solveurs* : un même modèle peut être compilé pour différents solveurs, plusieurs implantations de systèmes de résolution différents, appartenant à une même famille de technique ou pas.
- *termine* : la terminaison de l'exécution des programmes générés est assurée afin d'éviter à l'utilisateur d'écrire des modèles de problèmes combinatoires qui ne terminent pas à causes d'erreurs de programmation. Cette garantie permet donc à l'utilisateur de se concentrer purement sur la formulation et la résolution de ses problèmes.

3.1 OPL et COMET

OPL (Optimization Programming Language)

OPL [Van99] est un langage pour formuler et résoudre les problèmes d'optimisation combinatoire. OPL a rencontré un succès particulier dans la résolution des problèmes de planification et d'ordonnancement.

C'est le premier langage de modélisation à fournir à la fois une notation algébrique de haut niveau avec un langage de contraintes (composante de modélisation par contraintes) et les moyens programmer des stratégies de recherche (composante de programmation de la recherche) [HPP00]. Les expressions d'un modèle portent des types qui sont donnés explicitement et vérifiés à la compilation.

Les stratégies de branchement peuvent être exprimées grâce au quantificateur universel `forall` pour le séquencement et à l'opérateur spécifique de choix non déterministe `tryall`. Des heuristiques peuvent être écrites comme des ordres (lexicographiques) sur les éléments des listes qui paramètrent les quantificateurs universels `forall` et les itérateurs non déterministes `tryall`.

Le langage n'est pas extensible (pas de définitions utilisateur), les contraintes sont exprimées de façon monolithique, et le langage ne dispose pas de combinateurs génériques (*fold*) sur les listes.

Exemple 4. *Le modèle OPL du problème des n-reines [HPP00].*

Le type intervalle (`range`*) est utilisé pour construire le tableau* `queen` *de type variable (*`var`*) qui représente le placement des* `n` *reines sur les lignes l'échiquier (chaque reine est implicitement sur une colonne différente).*

```
int n = ...;
range Domain 1..n;
var Domain queen[Domain];
```

C'est exclusivement dans le bloc monolithique `solve` *que les contraintes du problème sont exprimées.*

```
solve {
  forall(ordered i,j in Domain) {
    queen[i] <> queen[j];
    queen[i] + i <> queen[j] + j;
    queen[i] - i <> queen[j] - j }
};
```

La composante recherche est quant à elle exprimée dans le bloc `search`*. Ici, la stratégie de branchement est un* labeling *(cf. Sec 4.3) avec une heuristique d'ordonnancement des variables et une heuristique d'ordonnancement de valeurs.*

```
search {
  forall(i in Domain ordered by increasing dsize(queen[i]))
    tryall(v in Domain ordered by increasing abs(n/2-v))
      queen[i] = v;
};
```

L'heuristique de sélection de variables applique le principe first-fail *(cf. Sec 4.3) sur la liste* `Domain` *qui paramètre le quantificateur universel* `forall`*. L'heuristique ordonne les éléments* `i` *de cette liste selon l'ordre croissant des tailles de domaine des variables* `queen[i]`*. Ici, les* `i` *jouent le rôle d'index pour accéder aux reines dans le tableau* `queen`*.*

L'heuristique de sélection de valeurs s'applique sur la liste `Domain` *qui paramètre l'itérateur non déterministe* `tryall`*. Elle ordonne les éléments* `v` *de cette liste par ordre croissant de distance au milieu du domaine* `n/2`*.*

Bien que cet exemple simple ne l'illustre pas en profondeur, la composante recherche d'OPL est une composante de programmation. Elle intègre notamment, outre le séquencement et la manipulation d'indices et de tableaux, des structures de contrôle comme les conditionnelles (`if...then...else...`) et les boucles « tant que » (`while`).

Composante Contraintes			Composante Recherche		
extensible	règles	modulaire	strat. branch.	heur. ordo.	parc. d'arb.
non	non	non	**oui**	**oui**	non

Table 3.1: Récapitulatif des traits d'OPL

Compilation				Exécution		
statique	dynamique	vérif. typ.	reform.	hybrid.	m-solveurs	term.
non	**oui**	**oui**	non	non	non	non

Table 3.2: Traits de la compilation et de l'exécution des modèles OPL

COMET

Comet [MH05] est un langage orienté objet riche qui peut être vu comme une évolution d'OPL vers l'hybridation des techniques de programmation par contraintes et de recherche locale, nommée *constraint-based local search* (CBLS).

Comme dans OPL, les deux composantes contraintes et recherche sont présentes et dissociées. Comet propose un langage de contraintes et des constructions pour programmer des stratégies de branchement, des heuristiques d'ordonnancement (de listes), et aussi des parcours d'arbres grâce aux fermetures et aux continuations.

Comet est un langage extensible, il autorise les définitions utilisateur (récursives) mais les contraintes doivent toujours être composées dans le bloc `solve`. En Comet, les contraintes utilisateur sont définies comme des classes qui étendent la classe `UserConstraint<CP>`.

Le langage offre des facilités pour l'ingénierie et permet par exemple différentes entrées/sorties pour se connecter à des bases de données ou lire des fichiers XML. Comet permet aussi de construire des interfaces utilisateur pour observer graphiquement l'évolution de l'exécution des modèles compilés.

Exemple 5. *Le modèle Comet du problème des n-reines est très proche du modèle OPL.*

On remarque l'usage de modules `import cotfd;` *et de l'opérateur d'écriture sur la sortie standard* `cout`*. L'objet* `m` *représente un solveur qui est ici de type* `CP` [1] *c'est-à-dire un solveur suivant les principes de la PPC. Il est à noter que COMET, comme OPL, fait usage de ses propres solveurs et le compilateur ne permet pas d'exécuter les modèles sur d'autres plateformes.*

```
import cotfd;

Solver<CP> m();
int n = 120;
range S = 1..n;
var<CP>{int} q[i in S](m,S);
Integer np(m.getNPropag());
```

Comme dans OPL, les contraintes sont composées exclusivement dans le bloc `solve`*. On note ici l'usage de la contrainte globale* `alldifferent` *et qu'il est possible de régler la force de propagation des contraintes avec le second argument de la méthode* `m.post`*.*

```
solve<m> {
   m.post(alldifferent(all(i in S) q[i] + i), onDomains);
   m.post(alldifferent(all(i in S) q[i] - i), onDomains);
   m.post(alldifferent(q), onDomains);
}
```

Enfin, la composante recherche, exprimée dans le bloc `using`*, est similaire à celle d'OPL. Les heuristiques d'ordonnancement sont indissociables de la définition des stratégies de branchement. Les ordres heuristiques ne peuvent être définis que vis-à-vis d'une liste de quantificateur, comme c'est le cas pour l'heuristique de type* first-fail `(q[i].getSize())` *vis-à-vis des éléments de la liste* `i in S` *qui paramètre le* `forall` *dans l'exemple suivant.*

```
using {
   forall(i in S) by (q[i].getSize()) {
     tryall<m>(v in S : q[i].memberOf(v))
       q[i] = v;
   }
}
cout << "Solution = " << q << endl;
cout << "#choices = " << m.getNChoice() << endl;
cout << "#fail = " << m.getNFail() << endl;
cout << "#propag = " << m.getNPropag() - np << endl;
```

1. Les solveurs peuvent aussi être de type `LS` (pour *local search*) et interagir avec les solveurs `CP`. De bonnes solutions peuvent être trouvées rapidement par recherche locale et servir de base pour une preuve d'optimalité avec un solveur de type `CP`.

Composante Contraintes			Composante Recherche		
extensible	règles	modulaire	strat. branch.	heur. ordo.	parc. d'arb.
oui	non	**oui**	**oui**	**oui**	**oui**

TABLE 3.3: Récapitulatif des traits de Comet

Compilation				Exécution		
statique	dynamique	vérif. typ.	reform.	hybrid.	m-solveurs	term.
non	**oui**	**oui**	non	**oui**	non	non

TABLE 3.4: Traits de la compilation et de l'exécution des modèles Comet

Les programmes OPL et Comet sont compilés à la volée (*Just-In-Time compilation*) en code procédural interprété par des machines virtuelles dédiées efficaces et ne souffrent donc pas d'explosion en taille du code généré. De plus, la terminaison de la compilation des modèles OPL et Comet est garantie précisément car elle procède par génération de procédures sans expansion des expressions. Par contre, il n'y a pas de garantie de terminaison de l'exécution de ces programmes qui peuvent par exemple faire usage de constructions `while` ou de définitions récursives. Enfin, ces langages prennent le parti de programmer la recherche. Ils offrent des structures de contrôle comme les conditionnelles et les boucles « tant que » et demandent à déclarer, transformer ou manipuler les structures de données de telle sorte qu'elles soient exploitables par des heuristiques de réordonnancement de listes.

3.2 ZINC

Zinc [dlBMRW06, RdlBMW07] est un langage de modélisation qui entend rendre ses modèles résolubles par le plus de solveurs possibles et donc établir un standard de fait.

Le langage donne les moyens à l'utilisateur d'étendre le langage avec ses propres définitions de fonctions et prédicats qui peuvent être surchargées. Zinc permet l'inclusion de fichiers qui a pour effet d'insérer les définitions contenues à l'endroit de l'inclusion. Tous les identifiants (variables, fonctions, prédicats) ont une portée globale, ils ne sont pas préfixés par un nom de module. Par ailleurs, la terminaison de la compilation et de l'exécution des programmes générés est garantie.

Zinc intègre des mécanismes de coercition comme la réification mais celle-ci doit être explicitée par la fonction `bool2int/1`, ce qui oblige à manipuler des constructions de bas niveau dans les définitions de contraintes. La compilation des modèles Zinc génère des programmes de contraintes « plats ». Il inclut contraintes ensemblistes et combinateurs de listes génériques. Le langage est typé explicitement et le compilateur est capable de lever des problèmes d'instanciation des modèles.

Le point fort de Zinc est son indépendance vis-à-vis des différents types et implantations de solveurs, de plateformes de résolution. Un fragment du langage, nommé MiniZinc [NSB+07], et un langage intermédiaire « plat », nommé FlatZinc, ont été développés et permettent aujourd'hui au langage d'utiliser les solveurs `GeCode` [SLM], `Eclipse`

[AW07], `ILOG solver` [ILO], `Minion` [GJM06], `Choco-Java` [Lab00] et `SICStus-Prolog` [C+07]. Les programmes générés sont donc sujets à l'explosion en taille.

Zinc ne possède pas les constructions dédiées à la recherche, et permet seulement d'annoter les modèles pour indiquer le type de méthode de résolution ou la stratégie de recherche à employer [RMdlB+08].

Exemple 6. *Le modèle Zinc du problème des n-reines [RMdlB+08].*

Comme illustré ci-dessous, Zinc offre sensiblement les mêmes types de base qu'OPL : type entier (`int`*), type intervalle (..), type tableau (*`array`*), variable (*`var`*).*

```
int: n;
type Domain = 1..n;
array[Domain] of var Domain :q;
```

Les contraintes actives du modèle sont préfixées par le mot clé `constraint`*. On remarque la définition utilisateur du prédicat* `noattack`*, utilisé dans la contrainte active.*

```
predicate noattack(Domain: i,j, var Domain: qi,qj) =
  qi != qj /\
  qi + i != qj + j /\
  qi - i != qj - j;

constraint
  forall(i,j in Domain where i<j)
    noattack(i,j,q[i],q[j]);
```

Le modèle indique le type de recherche utilisée par l'annotation `satisfy::backtrack` *(méthode arborescente par* backtrack*).*

```
solve satisfy::backtrack(q, std_label);
```

L'annotation est paramétrée par la fonction `std_label`*. À partir de la liste de variables* `Vs`*,* `std_label` *produit une liste de paires dont la première composante est la queue de* `Vs` *et la seconde l'affectation de la variable en tête. La liste de paires produite par* `std_label` *sert de base à un* labeling *sans heuristique. On peut remarquer l'usage du type générique* `$T` *pour le premier argument d'entrée et l'argument de retour de la fonction.*

```
function list of tuple(list of $T, var bool): std_label(list of $T:Vs) =
  if Vs = [] then []
  else [ (tail(Vs), head(Vs) == d) | d in domain(head(Vs))]
  endif;
```

Composante Contraintes			Composante Recherche		
extensible	règles	modulaire	strat. branch.	heur. ordo.	parc. d'arb.
oui	non	**oui**	non	non	non

TABLE 3.5: Récapitulatif des traits de Zinc

Compilation				Exécution		
statique	dynamique	vérif. typ.	reform.	hybrid.	m-solveurs	term.
oui	non	**oui**	non	**oui**	**oui**	**oui**

TABLE 3.6: Traits de la compilation et de l'exécution des modèles Zinc

3.3 ESSENCE

Essence [FHJ+08] est un langage de modélisation qui se concentre sur les problématiques d'abstraction et de reformulation des modèles. Il offre une grande variété de structures de haut niveau ((multi)ensembles, fonctions, relations, partitions, etc.) et un système de type fin qui guide la compilation vers les CSPs. Le type d'une variable de décision peut être n'importe quelle (imbrication de) structure(s). Par conséquent, le langage permet de déclarer des variables dont les valeurs sont directement des objets combinatoires et de laisser les choix de modélisation et d'optimisation vers les CSPs au compilateur.

La compilation des modèles Essence procède par expansion des expressions et génère des programmes « plats » destinés aux solveurs `Minion` [GJM06] ou `Eclipse` [AW07]. La taille des CSPs générés peut donc exploser malgré les différentes optimisations appliquées à la compilation. De plus, il manque des combinateurs génériques sur les structures inductives, les listes par exemple, et rien n'est prévu pour contrôler la recherche qui sera effectuée avec la stratégie par défaut du solveur utilisé. Par ailleurs, puisque que le langage n'offre ni définitions utilisateurs, ni constructions de type boucle `while`, la compilation et l'exécution sont garanties de se terminer.

La reformulation consiste à transformer un modèle posant des contraintes arbitraires sur des structures complexes en un CSP valide pour des solveurs de contraintes. La compilation par expansion des expressions du modèle qui aplatit des contraintes avec quantificateurs sur des objets très structurés a l'inconvénient de générer du code de grande taille. Mais ce code contient un certain nombre de redondances et de sous-expressions communes. La reformulation optimise une transformation naïve en une transformation de moins grande taille avec le moins de redondances possibles et peut insérer des contraintes impliquées pour améliorer l'efficacité de la propagation. Pour limiter la taille des programmes générés, il s'agit de faire produire au compilateur un code qui contienne un maximum de sous-expressions communes puis de faire un usage intensif des techniques d'élimination des sous-expression communes (CSE) ainsi que d'optimisations de code en général (optimisation des itérateurs, analyse des invariants, etc.).

Exemple 7. *Le modèle Essence du problème des n-reines.*

Essence distingue les identifiants qui doivent être instanciés à la compilation des autres avec le mot clé `given`*. La déclaration et l'initialisation des structures de données se fait après le mot clé* `letting`*.*

Dans ce modèle, la taille de l'instance `n` *doit être connue statiquement et* `Index` *représente le domaine des variables du problème.*

```
given n : int(*1..*)
letting Index be domain int(*1..n*)
```

Une reine est placée sur chaque ligne de l'échiquier et la variable combinatoire est ici la fonction `arrangement` *qui représente directement le placement des reines sur les colonnes de l'échiquier. La fonction* `arrangement` *est une bijection, ce qui assure que chaque colonne contienne exactement une reine.*

Les contraintes du problème qui concernent les diagonales sont exprimées dans le bloc monolithique `such that` *associé au but* `find`*.*

```
find arrangement : function Index -> (*bijective*) Index
such that
  forall q1, q2 : Index .
    q1 neq q2 implies |arrangement(q1) - arrangement(q2)| neq |q1 - q2|
```

Composante Contraintes			Composante Recherche		
extensible	règles	modulaire	strat. branch.	heur. ordo.	parc. d'arb.
non	non	non	non	non	non

TABLE 3.7: Récapitulatif des traits d'Essence

Compilation				Exécution		
statique	dynamique	vérif. typ.	reform.	hybrid.	m-solveurs	term.
oui	non	**oui**	**oui**	**oui**	**oui**	**oui**

TABLE 3.8: Traits de la compilation et de l'exécution des modèles Essence

3.4 s-COMMA

s-COMMA [Sot09] est un langage de modélisation orienté objet (simplification de Java) destiné aux programmeurs qui ne sont pas spécialistes de la PPC. Dans cet esprit, le langage offre une représentation graphique des modèles à la UML pour exprimer les problèmes. s-COMMA a pour ambition de participer à la définition d'un standard de langage de modélisation.

Les structures de données du langages sont les tableaux, les énumérations, les ensembles, les booléens, et les variables à domaine fini ou variables à domaine continu. Ces dernières ne pouvant pas (encore) être des variables de décision. Le langage propose un quantificateur universel et des structures de contrôle conditionnelles. s-COMMA n'offre donc pas de combinateur générique ni de quantificateur existentiel.

s-COMMA ne permet pas à l'utilisateur de définir des stratégies de branchement ni des parcours d'arbre, fixés respectivement au *labeling* et à *DFS*. Le langage propose un jeu d'options prédéfinies pour paramétrer la recherche et qui correspond à un ensemble d'heuristiques d'ordonnancement classiques.

À l'instar de Zinc et d'Essence, la compilation des modèles s-COMMA procède par expansion des expressions. Le langage est compatible avec plusieurs implantations de

solveurs (Gecode, Eclipse, GNU-Prolog et RealPlayer). Il est possible d'ajouter au langage de nouvelles contraintes globales ou fonctions d'une implantation de solveur particulière grâce à des transformations de modèles qui associent des expressions s-COMMA à des expressions du solveur cible chosi.

Exemple 8. *Le modèle s-COMMA du problème des n-reines.*

La classe `Queens` *regroupe les attributs (le tableau de reines* `q[n]`*) et les contraintes (la contrainte de non-prise* `noAttack`*) qui définissent le problème.*

L'ordre heuristique pour le labeling *est donné par les options prédéfinies* `min-dom-size` *(plus petite cardinalité de domaine) pour la sélection des variables et* `med-val` *(valeur médiane) pour la sélection des valeurs.*

```
int n := 10;

main class Queens [min-dom-size, med-val] {
  int q[n] in [1, n];

  constraint noAttack {
    forall(i in 1..n) {
      forall(j in i+1..n) {
        q[i] <> q[j];
        q[i] + i <> q[j] + j;
        q[i] - i <> q[j] - j; }}
  }
}
```

Composante Contraintes			Composante Recherche		
extensible	règles	modulaire	strat. branch.	heur. ordo.	parc. d'arb.
oui	non	**oui**	non	non	non

TABLE 3.9: Récapitulatif des traits de s-COMMA

Compilation				Exécution		
statique	dynamique	vérif. typ.	reform.	hybrid.	m-solveurs	term.
oui	non	**oui**	non	non	**oui**	**oui**

TABLE 3.10: Traits de la compilation et de l'exécution des modèles s-COMMA

3.5 Langages CLP et CIP

Les langages de programmation logique avec contraintes (PLC ou *CLP*) basés sur Prolog (comme `SICStus-Prolog`, GNU-Prolog ou Eclipse) ainsi que les langages de programmation impérative avec contraintes (PIC ou *CIP*), les bibliothèques de contraintes définies dans des langages impératifs (comme Choco, Gecode ou ILOG Solver) sont très

puissants et permettent notamment à l'utilisateur de définir de nouvelles contraintes de façon modulaire, et de programmer des stratégies de branchement, des heuristiques d'ordonnancement et des parcours de recherche.

Mais concrètement, les langages de PLC et de PIC offrent des constructions de bas niveau, la notation qu'ils proposent n'est pas purement déclarative, ils ne dissocient pas assez clairement la modélisation des traits extra-logiques et incluent rarement des constructions logiques essentielles telles que les quantificateur universels et existentiels. Enfin, la terminaison des programmes n'est pas garantie.

Les objectifs qui motivent la conception de langages de modélisation s'opposent à ces langages de programmation qui laissent implicite la modélisation induite par des constructions de programmation. Ils demandent un effort d'interprétation du code et de solides connaissances en programmation à des utilisateurs qui peuvent être ingénieurs experts d'un domaine industriel et pas nécessairement programmeurs.

Composante Contraintes			Composante Recherche		
extensible	règles	modulaire	strat. branch.	heur. ordo.	parc. d'arb.
oui	non	**oui**	**oui**	**oui**	**oui**

TABLE 3.11: Récapitulatif des traits des langages de PLC et PIC

Compilation				Exécution		
statique	dynamique	vérif. typ.	reform.	hybrid.	m-solveurs	term.
non	**oui**	**oui**	non	**oui**	non	non

TABLE 3.12: Traits de la compilation et de l'exécution des programmes PLC et PIC

3.6 Langages BRMS

Le formalisme de règles métier, ou *business rules*, a été introduit comme un formalisme de représentation des connaissances, en particulier des connaissances spécifiques à un domaine d'application, par des règles de la forme `si` *A* `alors` *B*.

En pratique, ces règles sont exploitées par des systèmes de gestion de règles métier (SGRM), ou *business rules management systems* (*BRMS*), comme JBoss Drools [JBo] ou IBM ILOG JRules [IBM]. Dans un SGRM, les règles sont considérées comme des *règles de production* et exécutées par un moteur d'inférence en chaînage avant, un algorithme de type Rete [For82], et de la forme :

`si` <*Motif*> <*Condition*> `alors` <*Action*>

Une règle est déclenchée lorsqu'un terme (dans la mémoire centrale) représentant un fait est filtré par la partie <*Motif*> et qu'un sous-ensemble des arguments du terme respecte la partie <*Condition*>. L'effet de la règle est défini par la partie droite <*Action*> qui pourra modifier l'état de la mémoire centrale. Le moteur d'inférence s'exécute jusqu'à saturation, c'est-à-dire jusqu'à ce que plus une règle ne soit déclenchable. Les parties gauches et droites de telles règles sont exprimées dans un langage de programmation impératif tel que Java, comme c'est le cas de certains langages de PIC.

Un inconvénient rédhibitoire des SGRM, du point de vue de la modélisation, est que l'écriture des règles dépend fortement de l'interprétation procédurale des moteurs d'inférence à la Rete qui les exécutent. Parce que cette sémantique opérationnelle et la sémantique déclarative des règles métier diffèrent notablement, et pour les mêmes raisons que les langages de PLC et de PIC, les SGRM sont inadaptés à la modélisation de problèmes d'optimisation combinatoire. Néanmoins, les SGRM peuvent trouver une application en PPC comme interface entre un système dynamique et un solveur de PPC comme proposé dans [vdKFLS10].

Conclusion

Un langage de modélisation se concentre sur la tâche de formulation des problèmes combinatoires. Il est important à cet égard d'offrir des constructions de modélisation d'un niveau d'abstraction suffisant, des mécanismes de coercition implicites, notamment pour la réification, et les moyens d'étendre le langage grâce à des définitions utilisateur dans un cadre purement déclaratif.

La plupart des langages de modélisation de l'état de l'art demandent à exprimer les problèmes de façon monolithique, ce qui limite la réutilisabilité des modèles et la possibilité pour l'utilisateur de définir des bibliothèques de contraintes.

À l'exception d'OPL et de Comet, qui offrent une composante pour programmer la recherche, les stratégies de recherche ne peuvent être écrites dans ces langages. Par contre, OPL et Comet ne garantissent pas la terminaison de l'exécution des modèles. La stratégie de recherche adoptée lors de l'exécution des modèles des autres langages est celle qu'offre par défaut le solveur utilisé. Cela pose un réel problème d'efficacité car, hormis les problèmes académiques classiques, il est souvent indispensable de contrôler la recherche pour résoudre effectivement un problème combinatoire, *a fortiori* un problème industriel qui impliques de nombreuses contraintes spécifiques (des règles métiers).

Chaque langage fait le choix d'une compilation par expansion des expressions ou d'une compilation par génération de code procédural qui conserve la structure des modèles, mais aucun n'a jamais présenté formellement le processus de compilation.

Afin de comparer synthétiquement les différents langages de modélisation de l'état de l'art entre eux et avec `Cream`, la table 3.13 récapitule les traits de tous les langages plus ceux de `Cream`.

Langage	Composante Contraintes			Composante Recherche		
	ext.	règles	mod.	branch.	heur.	parc.
Opl	non	non	non	**oui**	**oui**	non
Comet	**oui**	non	**oui**	**oui**	**oui**	**oui**
Zinc	**oui**	non	**oui**	non	non	non
Essence	non	non	non	non	non	non
s-Comma	**oui**	non	**oui**	non	non	non
`Cream`	**oui**	**oui**	**oui**	**oui**	**oui**	non

TABLE 3.13: Récapitulatif des traits de tous les langages de l'état de l'art plus `Cream`

Langage	Compilation				Exécution		
	stat.	dyn.	typ.	reform.	hybrid.	m-solveurs	term.
Opl	non	**oui**	**oui**	non	non	non	non
Comet	non	**oui**	**oui**	non	**oui**	non	non
Zinc	**oui**	non	**oui**	non	**oui**	**oui**	**oui**
Essence	**oui**	non	**oui**	**oui**	non	**oui**	**oui**
s-Comma	**oui**	non	**oui**	non	non	**oui**	**oui**
`Cream`	**oui**	**oui**	**oui**	non	non	**oui**	**oui**

TABLE 3.14: Récapitulatif des traits de la compilation et de l'exécution de tous les langages de l'état de l'art plus `Cream`

Nous verrons dans cette thèse que dans un langage de modélisation à base de règles comme `Cream`, il est possible de spécifier et de composer des connaissances par fragments plutôt qu'en un tout monolithique. Il sera aussi montré que `Cream` offre une composante recherche purement déclarative. `Cream` donne le choix entre les deux grands schémas de compilation qui seront présentés formellement. La compilation et l'exécution des modèles `Cream` sont garanties de se terminer. Enfin, le formalisme de règles et la modularité de `Cream` ont permis de construire une bibliothèque pour la modélisation des connaissances en placement qui sera présentée et utilisée pour résoudre des problèmes combinatoires issus de la logistique dans l'industrie automobile. Par ailleurs, un système de type pour `Cream` gérant la réification implicite a été développé par Thierry Martinez et présenté dans un article à paraître ([MMF10]).

Chapitre 4

Procédures de recherche

Sommaire

La propagation n'aboutit en général qu'à une valuation partielle que l'on souhaite compléter en une solution. Il est donc nécessaire, et crucial en terme d'efficacité, de définir une *procédure de recherche*, guidée par des heuristiques, qui explore l'espace des combinaisons de valeurs encore possibles. Nous nous intéressons essentiellement aux méthodes d'exploration arborescentes en profondeur d'abord.

Une procédure de recherche décide d'une *stratégie de branchement*, qui induit un arbre de recherche, et d'un *parcours d'arbre*. Les branches de l'arbre et l'ensemble des contraintes sur chaque branche peuvent être réordonnées selon des *heuristiques d'ordonnancement* de choix. Les heuristiques de recherche sont des principes qui exploitent des connaissances spécifiques à un problème ou l'information localement accumulée pendant le parcours d'un arbre de recherche. Elles ont pour finalité de minimiser le nombre de noeuds explorés avant de trouver une solution ou de prouver qu'il n'en existe pas.

4.1 Parcours d'arbres de recherche

L'exploration d'un espace de recherche donné décrit implicitement un arbre de recherche. En chaque noeud d'un tel arbre se présente un ensemble d'alternatives de décisions parmi lesquelles il faut faire un *choix* afin d'arriver au but.

Dans le cadre de la PPC, le but est une réduction minimale des domaines des variables (cf. stratégies de branchement, section 4.2) qui permette de décider de la satisfaction de toutes les contraintes d'un problème donné. À chaque arête de l'arbre de recherche est associée une pose de contrainte, une affectation de variable par exemple. Il est à noter que la propagation détecte en chaque noeud les cas d'insatisfaisabilité, élaguant ainsi les branches de l'arbre de recherche avant l'énumération exhaustive.

Un parcours d'arbre de recherche commence par la racine et deux décisions doivent être prises : l'axe et le sens du parcours, c'est-à-dire parcourir verticalement ou hori-

zontalement et de gauche à droite ou de droite à gauche, voire avec permutation des noeuds. Une procédure de recherche qui garantit de trouver la solution (si une solution existe) est nommée complète, incomplète sinon.

Par la suite, la stratégie de branchement est fixée et nous appelons d la profondeur de l'arbre parcouru et b le facteur de branchement, le nombre maximum d'enfants d'un noeud.

Recherche en largeur d'abord

La recherche en largeur d'abord, *breadth-first search* (BFS), explore un arbre horizontalement, niveau par niveau. Tous les noeuds de l'arbre de profondeur k sont parcourus avant tous les noeuds de profondeur $k+1$. La complexité en temps de BFS est en $\mathbf{O}(b^d)$.

L'avantage de cette exploration est qu'elle est complète, même en cas de branche infinie. L'inconvénient est que la complexité en espace de BFS est exponentielle en la profondeur, en $\mathbf{O}(b^d)$.

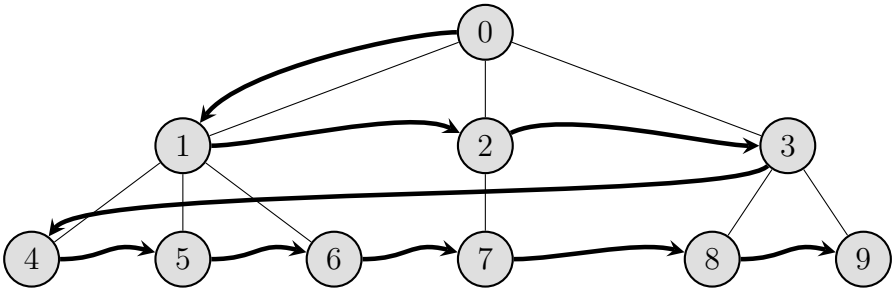


FIGURE 4.1: BFS *leftmost* (à gauche d'abord).

Recherche en profondeur d'abord

La recherche en profondeur d'abord [Tar72], *depth-first search* (DFS), explore un arbre verticalement, branche par branche. À chaque noeud de l'arbre, le premier fils du noeud est parcouru récursivement, puis le second, et ainsi de suite. À chaque feuille de l'arbre, DFS rebrousse chemin, *backtrack*, sur le dernier noeud qui possède des fils encore inexplorés et choisit le prochain enfant. La complexité en temps de DFS est en $\mathbf{O}(b^d)$.

Ce type de parcours a l'avantage d'être de complexité en espace linéaire, en $\mathbf{O}(d)$. Mais la recherche DFS n'est pas complète lorsque l'arbre considéré possède des branches infinies.

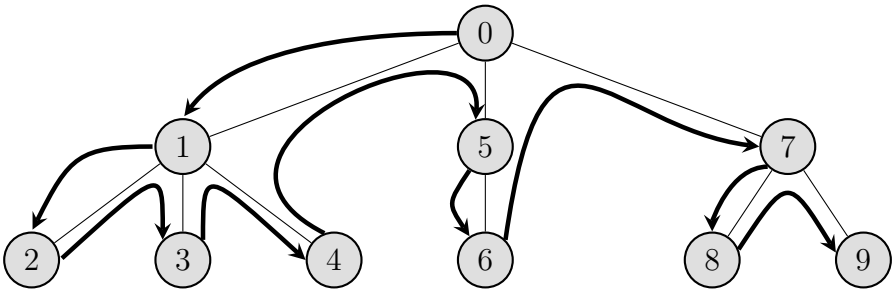


FIGURE 4.2: DFS à gauche d'abord.

Dans le contexte de la PPC, des contraintes réduisant le domaine des variables sont accumulées sur les branches. Il arrive alors que la propagation infère des domaines vides pour certaine variables avant d'arriver à une feuille, un tel échec provoque aussi un *backtrack*.

Recherche par approfondissements successifs

La recherche par approfondissements successifs [Kor85], *iterative deepening depth-first search* (IDDFS), est une recherche en profondeur itérative limitée à la profondeur 1, puis à la profondeur 2, et ainsi de suite jusqu'à l'obtention d'une solution (ou l'échec de la recherche). La complexité en temps de IDDFS est en $\mathbf{O}(b^{d+1})$.

La limitation de chaque itération à une profondeur finie rend la recherche en profondeur complète. La complexité en espace de IDDFS est linéaire, en $\mathbf{O}(d)$.

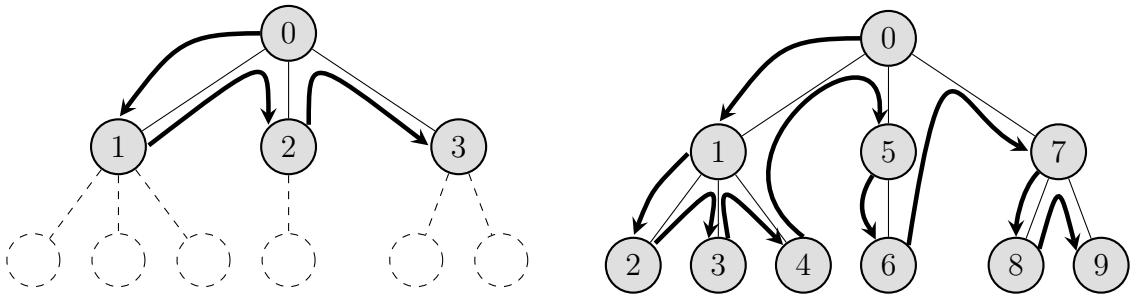
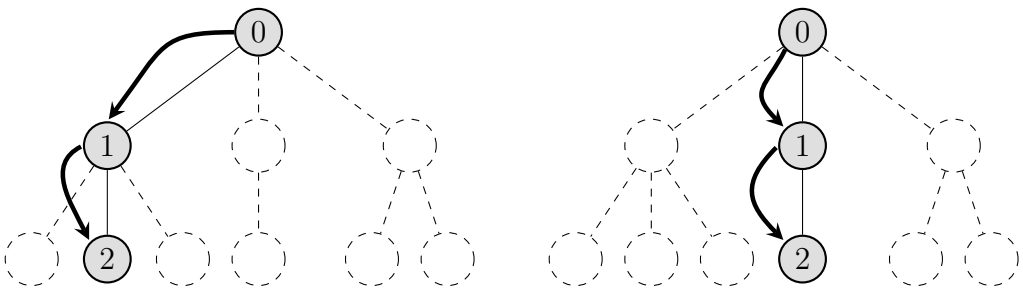


FIGURE 4.3: IDDFS à gauche d'abord, de profondeur 1 et profondeur 2.

Recherche en profondeur avec écarts limités

La recherche en profondeur avec écarts limités [HG95, Kor96], *limited discrepancy search* (LDS), est une recherche en profondeur d'abord qui explore l'arbre avec un nombre croissant d'écarts, de décisions contradictoires avec une heuristique donnée.

Cette méthode échantillonne l'espace de recherche et repose sur une heuristique donnée pour explorer la fraction de l'espace qui est probable de contenir des solutions. L'intuition est qu'une bonne heuristique amène souvent à une solution, mais pas toujours.



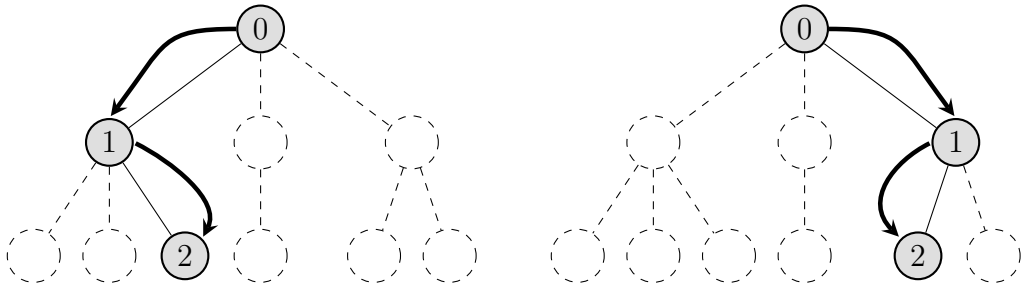


FIGURE 4.4: Itérations de LDS avec l'heuristique qui ordonne les noeuds de gauche à droite, avec 1 écart (première ligne) et 2 écarts (deuxième ligne).

Recherche du meilleur d'abord

La recherche par meilleur d'abord [DP85, VKK91, ZK93], *best-first search* (BFS), est une recherche générique qui explore les noeuds dans un ordre non décroissant de coût. À chaque noeud de l'arbre, c'est le fils de plus petit coût vis-à-vis d'une fonction d'évaluation f qui est exploré d'abord, puis le deuxième meilleur, et ainsi de suite.

En particulier, si la fonction de coût est $f(n) = \text{profondeur}(n)$, alors c'est la recherche en largeur d'abord qui est appliquée.

De façon générale, la fonction d'évaluation $f(n)$ exploite les informations de description d'un noeud n, l'information collectée jusqu'en n et les connaissances spécifiques au problème considéré.

Recherche de l'optimum par séparation et évaluation

La recherche de solution optimale par séparation et évaluation progressive [LD60], *branch and bound* (B&B), est une approche arborescente pour la recherche de solution optimale, vis-à-vis d'une fonction objectif, de problèmes d'optimisation combinatoire.

En théorie, il est toujours possible de trouver la solution optimale à un problème (s'il existe une solution) par une énumération explicite et exhaustive des combinaisons de valeurs possibles pour les variables. En pratique, on cherche à l'éviter car il existe potentiellement un nombre exponentiel de solutions en la taille du problème. On fait alors usage de méthodes qui garantissent l'optimalité sans explorer explicitement tout l'espace de recherche.

B&B à base de contraintes [Van89] est une telle méthode adaptée à la PPC qui consiste en une (séquence de) recherche avec une borne inférieure (supérieure) de la fonction objectif progressivement (dé)croissante.

Supposons, sans perte de généralité, que l'on souhaite trouver la solution à un problème de contraintes P qui minimise une fonction $f(X)$. Opérationnellement, une première recherche sur le problème $P_1 = P$ est lancée et aboutit à une solution dont le coût c_1 est évaluer par la fonction objectif. Ensuite, la contrainte $f(X) < c_1$ est ajoutée au problème, donnant le problème $P_2 = P_1 \wedge f(X) < c_1$, pour exclure toute solution de coût supérieur à c_1. Selon le schéma adopté, soit la recherche B&B *backtrack* et continue sur P_2, soit la recherche est réexécutée complètement (*restart*) pour P_2. La recherche globale termine lorsque le problème P_i est insatisfaisable, ce qui prouve que la solution optimale est de coût c_{i-1}.

4.2 Stratégies de branchement

Une stratégie de branchement détermine les (alternatives de) contraintes posées par la procédure de recherche. Dans le cadre d'une recherche en profondeur d'abord, par « *backtrack* », chaque noeud p à un niveau j de l'arbre de recherche correspond à un ensemble de contraintes dites de branchement $\{c_1, \ldots, c_j\}$ où chaque c_i, $1 \leq i \leq j$, est la contrainte posée au niveau i de l'arbre. Un noeud p est développé en créant k nouvelles branches $p \cup \{c_{j+1}^1\}$, ..., $p \cup \{c_{j+1}^k\}$ pour de nouvelles contraintes c_{j+1}^i, $1 \leq i \leq k$.

Trois stratégies de branchement sont fréquemment utilisées :

- *Énumération* des affectations de variables (*labeling* n-aire, cf. Fig. 4.5) qui développe un arbre dans lequel à chaque niveau est décidé quelle variable affecter et en chaque branchement quelle valeur choisir. Les contraintes de branchement en chaque noeud p sont donc de la forme $\{X_1 = v_1, \ldots, X_j = v_j\}$, où $v_i \in D_i$, et le noeud p est développé en créant n nouvelles branches $p \cup \{X_{j+1} = v_{j+1}^1\}$, $\ldots$, $p \cup \{X_{j+1} = v_{j+1}^n\}$ pour la prochaine variable X_{j+1} sélectionnée.

- *Choix binaires* (*labeling* binaire) qui développe un arbre dans lequel en chaque noeud p deux nouvelles branches sont créées : $p \cup \{X_{j+1} = v\}$ et $p \cup \{X_{j+1} \neq v\}$, où $v \in D_{j+1}$.

- *Découpage de domaine* (*domain splitting*) pose des contraintes d'inégalité sur les variables. Par exemple une telle stratégie génère au noeud p, pour une variable $X_i \in \{0, \ldots, 9\}$, les alternatives $p \cup \{X_i < 3\}$, $p \cup \{3 \leq X_i, X_i < 6\}$, et $p \cup \{6 \leq X_i\}$.

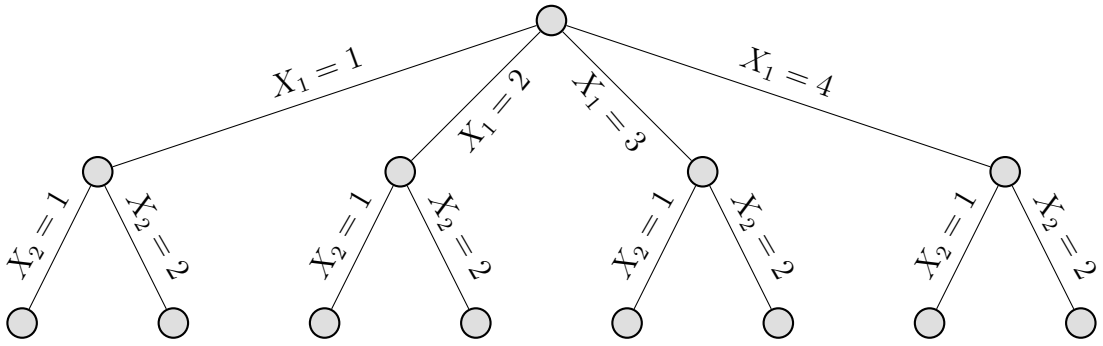


FIGURE 4.5: Un arbre de recherche généré par une stratégie de branchement par énumération des affectations, ou *labeling*, sur 2 variables $X_1 \in \{1, 2, 3, 4\}$ et $X_2 \in \{1, 2\}$.

Les stratégies de branchement ne se limitent pas à la pose de contraintes sur les domaines comme c'est le cas des trois stratégies présentées ci-dessus. Il est par exemple plus opportun et plus efficace dans les problèmes d'ordonnancement disjonctifs de brancher sur des contraintes de précédence entre tâches. Comme l'exemple de la section 7.1 l'illustre, les nouvelles branches créées en chaque noeud p d'un arbre de recherche induit par de telles stratégies sont alors de la forme $p \cup \{X_1 + d_1 \leq X_2\}, p \cup \{X_2 + d_2 \geq X_2\}$ où X_i et d_i représentent respectivement la date début et la durée de la tâche t_i.

4.3 Heuristiques d'ordonnancement de choix

Une heuristique d'ordonnancement réordonne les (alternatives de) contraintes déterminées par une stratégie de branchement afin de réduire la taille de l'arbre de recherche effectivement développé. Explorer un sous-arbre sans solutions peut être extrêmement coûteux et les décisions prises aux premiers niveaux de l'arbre de recherche sont particulièrement importantes.

Il existe deux grands types d'heuristiques d'ordonnancement :

- les heuristiques *conjonctives* qui décident de l'ordre des contraintes posées le long de chaque branche, des contraintes en conjonction (l'ordre des variables à affecter dans la figure 4.6) ;
- les heuristiques *disjonctives* qui décident de l'ordre des branches à essayer, des contraintes en disjonction (l'ordre des affectations pour chaque variable dans la figure 4.6 ou l'ordre des intervalles pour le découpage de domaine, cf. Chap. 9).

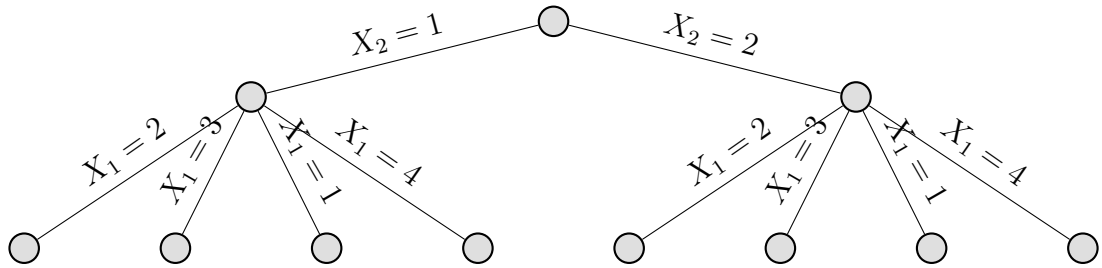


FIGURE 4.6: Un arbre de recherche généré par un *labeling* sur 2 variables $X_1 \in \{1, 2, 3, 4\}$ et $X_2 \in \{1, 2\}$ avec l'heuristique (conjonctive) de sélection de variables *min domain* et l'heuristique (disjonctive) de sélection de valeurs *middle out*.

Les heuristiques conjonctives comme les heuristiques disjonctives permettent de réduire la taille de l'arbre exploré lorsqu'il existe une solution, mais seules les heuristiques conjonctives sont utiles lorsqu'il n'en n'existe pas.

Aux deux types de choix (conjonctif ou disjonctif) correspondent deux principes heuristiques.

Le principe général régissant l'ordonnancement de contraintes en conjonction, est celui de l'échec d'abord ou *first-fail*. Pour éviter d'explorer les sous-arbres de recherche privés de solution, il faut s'apercevoir au plus vite de l'échec et donc commencer par traiter les contraintes « difficiles », les moins faciles à satisfaire. En pratique, on choisit souvent d'abord la variable de plus petit domaine (*min domain*), présente dans le plus de contraintes (*most-constrained*, *max degree*), ou encore la variable de plus petit domaine et présente dans le plus de contraintes (*min domain-over-degree*).

Au contraire, parmi les contraintes en disjonction, on choisit d'abord la plus « facile », la plus probable de mener à une solution, c'est le principe *succeed first*. Par exemple, on sélectionne la valeur qui a le plus grand support. On peut aussi énumérer le domaine dans le sens naturel (*up*), de la borne supérieure vers la borne inférieure (*down*), du milieu vers les bornes (*middle out*), ou encore des bornes vers le milieu.

Par ailleurs, les heuristiques peuvent être de deux natures :

- heuristiques *statiques* : l'ordre est fixé une fois pour toutes avant de commencer la recherche ;
- heuristiques *dynamiques* : l'ordre peut varier pendant la recherche, il est recalculer en chaque noeud de l'arbre notamment en fonction des effets de la propagation.

Il arrive qu'un critère d'ordre heuristique, comme le plus petit domaine par exemple, ne distingue pas deux variables. Si assez d'information est disponible, il est utile de les départager avec un second critère et l'on forme alors un *ordre lexicographique*.

Si des heuristiques (lexicographiques) sont définissable pour un problème, les procédures de recherche particulièrement adaptées en programmation par contraintes sont les procédures *depth-first best-first search* (DFBFS) et *limited discrepancy search* (LDS).

Le chapitre 7, et particulièrement son introduction, présente comment des heuristiques classiques en programmation par contraintes sont exprimées en `Cream`.

Conclusion

En PPC, la composante recherche s'attache à l'exploration de l'espace de recherche d'un problème combinatoire. En particulier, les heuristiques guident les choix effectués pendant la recherche pour réduire la taille de l'arbre de recherche exploré. C'est une préoccupation indispensable pour résoudre effectivement les problèmes combinatoires et de nombreuses stratégies efficaces ont vu le jour ces dernières décades.

Cependant, dans les langages de PPC, les arbres de recherche sont le résultat de l'exécution des différentes procédures de recherche, comme le *labeling* par exemple, paramétrées par des heuristiques qui ne sont pas exprimées déclarativement.

Nous verrons dans la partie suivante que dans un langage de modélisation à base de règles, il est possible de spécifier des arbres de recherche comme des formules logiques du modèle et de les réordonner de façon déclarative selon des heuristiques définies sur la structure de la formule, par *pattern-matching* sur les têtes de règles.

Deuxième partie

Langage de modélisation à base de règles

Sommaire

Chapitre 5

Syntaxe et sémantique

Sommaire

5.1 Syntaxe de `Cream`

Un modèle `Cream` consiste essentiellement en :
- un ensemble de *déclarations* dans lesquelles on distingue les définitions de constructeurs et de fonctions d'une part, des *règles* proprement dites pour la définition des prédicats d'autre part ;
- un ensemble d'*heuristiques* de réordonnancement des formules ;
- et un ensemble de formules *buts*, ou *requêtes*, pris en conjonction.

Nous décrivons ici la syntaxe de `Cream` de façon *bottom-up*, c'est-à-dire en décrivant d'abord les expressions de base du langage (données par le non-terminal *expr*), puis les heuristiques (*heuristic*) et enfin les déclarations (*declaration*) et les modèles (*model*).

Les structures de données élémentaires de `Cream` sont constituées de :

- *constantes entières*, munies des opérateurs arithmétiques et de comparaison usuels ;
- *variables à domaine fini*, avec indexicaux et la contrainte d'égalité en plus des opérateurs partagés avec les constantes entières ;
- *listes*, construites par énumération, intervalles entre deux entiers ou concaténation et parcourues avec les quantificateurs et les combinateurs ;
- *chaînes de caractères*, qui partagent les opérateurs sur les listes ;
- *enregistrements*, dont les champs étiquetés sont utilisés pour la projection.

La figure 5.1 présente la syntaxe des modèles `Cream` qui comprend les expressions de base, les expressions d'heuristiques, et les déclarations parmi lesquelles se trouvent les déclarations de règles.

Les entiers sont représentés par le non-terminal *integer* qui parcourt le domaine fini $\mathcal{D} \subseteq \mathbb{Z}$, où $\mathcal{D} = [\![\texttt{min\_integer}, \texttt{max\_integer}]\!]$. Une variable *var* est un mot commençant par une majuscule ou le tiret bas. Une chaîne de caractère *string* est un mot entre guillemets.

name ::= *ident* | *name* `:` *ident*
call ::= *name* | *name*`(`*expr*`,...,`*expr*`)`
expr ::= *integer* | *var* | *string* | `error`
| `min_integer` | `max_integer`
| *expr* ***op*** *expr* où ***op*** $\in$ {`+`, `-`, `*`, `/`}
| ***op***`(`*expr*`,` *expr*`)` où ***op*** $\in$ {`min`, `max`, `exp`, `log`}
| *expr* ***rel*** *expr* où ***rel*** $\in$ {`=`, `#`, `=<`, `<`, `>`, `>=`}
| *expr* ***logop*** *expr* où ***logop*** $\in$ {`and`, `or`, `implies`, `equiv`}
| `not` *expr*
| `forall(`*var* `in` *expr*`,` *expr*`)`
| `exists(`*var* `in` *expr*`,` *expr*`)`
| `domain(`*expr*`,` *expr*`,` *expr*`)`
| `domain_min(`*expr*`)` | `domain_max(`*expr*`)` | `domain_size(`*expr*`)`
| `variables(`*expr*`)` | `variable_name(`*expr*`)` | `variable_def(`*expr*`)`
| `[`*expr*`,...,`*expr*`]` | `[`*expr* `..` *expr*`]` | *expr* `++` *expr*
| `length(`*expr*`)` | `nth(`*expr*`,` *expr*`)` | `pos(`*expr*`,` *expr*`)`
| `{`*ident*`:` *expr*`,...,` *ident*`:` *expr*`}` | *expr*`:`*ident*
| `foldl(`*var* `from` *expr*`,` *var* `in` *expr*`,` *expr*`)`
| `foldr(`*var* `from` *expr*`,` *var* `in` *expr*`,` *expr*`)`
| `map(`*var* `in` *expr*`,` $\underline{expr}$`)`
| `let(`*var* `:=` *expr*`,` $\overline{expr}$`)`
| `minimize(`*expr*`,`*expr*`)` | `maximize(`*expr*`,`*expr*`)`
| `search(`*heuristic*`,`*expr*`)` | `constraint(`*expr*`)`
| `dynamic(`*expr*`)` | `static(`*expr*`)`
| *call*

head ::= *ident* | *ident*`(`*var*`,...,`*var*`)`
heuristic ::= `conjunctive(`***op***`(`*expr*`)` `for` *head*
| `disjunctive(`***op***`(`*expr*`)` `for` *head* où ***op*** $\in$ {`least`, `greatest`}
| `nil` | *heuristic* `and` *heuristic*
| *call*

declaration ::= `import(`*name*`).`
| *head* `=` *expr*`.`
| *head* `-->` *expr*`.`
| `domain` *ident* `= {` *ident*`,...,`*ident* `}.`
| `heuristic` *head* `=` *heuristic*
| `?` *expr*`.`

model ::= *declaration* ... *declaration*

FIGURE 5.1: Syntaxe des modèles `Cream`.

Un identifiant *ident* est un mot qui commence par une minuscule. *name* est un identifiant, potentiellement préfixé par d'autres identifiants, destiné à représenter des noms de modules. Une tête *head*, ou partie gauche de déclaration, est formée d'un *ident* et de variables distinctes en arguments lorsque son arité est non nulle.

Opérateurs logiques

Les *formules* sont considérées comme des expressions entières dans $\{0, 1\}$, elles sont dites *réifiées*. Cette coercition usuelle entre booléens et entiers apporte beaucoup d'expressivité [Van89]. La réification permet d'augmenter le niveau de discours et de parler de la valeur de vérité des contraintes.

Réifier une formule ou une contrainte c, c'est poser une équivalence entre une variable B à valeur dans $\{0, 1\}$ et la contrainte c, soit $B \Leftrightarrow c$. La sémantique est la suivante : si C est satisfaite, alors $B = 1$ est déduit et si C est insatisfaite, alors $B = 0$ est déduit ; inversement, si $B = 1$ alors C est posée et si $B = 0$ alors $\neg C$ est posée.

Variables à domaine fini

Le prédicat `domain/3` pose le domaine de la liste des variables libres contenues dans l'expression en premier argument. Les fonctions `domain_min/1`, `domain_max/1`, `domain_size/1` donnent respectivement la borne inférieure, la borne supérieure et la cardinalité du domaine.

La fonction `variables/1` renvoie la liste des variables libres contenues dans une expression. La fonction `variable_name/1` renvoie le nom d'une variable libre, déterminé par l'indexation (cf. section 5.2) et sa place dans le terme qui la contient. La fonction `variable_def/1)` renvoie la tête de déclaration instanciée qui a produit une variable libre donnée.

Listes et enregistrements

La construction de liste se fait par énumération, *e.g.* `[1, 3, 4, 5, 6, 8]` ; par intervalle, *e.g.* `[1, 3 ..6, 8]` ; ou par concaténation, *e.g.* `[1]++[3 ..6]++[8]`.

Le non-terminal *string* représente des mots quelconques entre guillemets, *e.g.* `"annotation"`. Comme les listes, les chaînes peuvent être concaténées, *e.g.* `"annotation" ++"_" ++"1"` donne `"annotation_1"`.

Un enregistrement est un ensemble de couples *champ* : *valeur*. L'évaluation de l'expression `{column :1, row :_} :column` donne la valeur du champ `column`, soit `1`. Par ailleurs, tout enregistrement porte par défaut le champ `uid` de valeur un entier naturel unique.

Combinateurs et liaison let

Le combinateur `fold(`A `from` i`,` X `in` l`,` e`)` combine, selon l'expression e, l'accumulateur courant A, initialement i, et l'élément courant X de la liste l ; et récursivement le nouvel accumulateur avec l'élément suivant de la liste, jusqu'à ce qu'elle soit épuisée. Un `foldl`, *fold left*, ou réduction par la gauche, commence avec l'élément gauche de la liste, ou tête de la liste. Un `foldr`, *fold right*, commence avec l'élément droit, c'est-à-dire le dernier élément de la liste.

Exemple 9. *Considérons la fonction* `reverse(`l`)` *qui inverse la liste* l*. Elle se définit comme* `foldl(`A `from [], `X `in `l`, [`X`] ++ `A`)`*, où* A *et* X *sont des variables fraîches. Les itérations successives de* `reverse([1, 2, 3])` *donnent :*

1. `[1]++[]`
2. `[2]++([1]++[])`
3. `[3]++([2]++([1]++[]))`

soit la liste `[3, 2, 1]`*.*

Ces combinateurs remplacent la récursivité dans le langage car il semble plus simple d'apprendre ce schéma de récursion contrôlé, partagé avec la programmation fonctionnelle, plutôt que la récursion explicite. De plus, une fois le schéma appris, il est univoque et facile à lire et donc à partager, ce qui n'est pas forcément le cas de code faisant usage de récursion explicite.

Certaines des constructions qui précèdent sont des sucres syntaxiques :
- `exists(`X `in `l`, `e`)` $\equiv$ `foldl(`A `from 0, `X `in `l`, `A `or `e`)`
- `forall(`X `in `l`, `e`)` $\equiv$ `foldl(`A `from 1, `X `in `l`, `A `and `e`)`
- `map(`X `in `l`, `e`)` $\equiv$ `foldl(`A `from [], `X `in `l`, `A `++ [`e`])`
- `reverse(l)` $\equiv$ `foldl(`A `from [], `X `in `l`, [`X`] ++ `A`)`
- `foldr(`A `from `i`, `X `in `l`, `e`)` $\equiv$ `foldl(`A `from `i`, `X `in reverse(`l`), `e`)`
- la construction `let` est étendue récursivement aux liaisons multiples. Pour tout n, un `let` de $n+1$ liaisons `let(`X_0`:=`e_0`,...,`X_n`:=`e_n`,`e`)` est défini en termes du `let` simple et du `let` de n liaisons : `let(`X_0`:=`e_0`,let(`X_1`:=`e_1`,...,`X_n`:=`e_n`,`e`))`.

Arbres de recherche et heuristiques

Le prédicat `search/2` permet de spécifier une stratégie de branchement en interprétant les disjonctions et les quantifications existentielles d'une formule logique comme des points de choix plutôt que comme des contraintes réifiées. Le premier argument est la place de l'heuristique de recherche qui s'applique sur l'arbre du modèle décrit par la formule en deuxième argument. À l'extérieur de ce prédicat, ou sous le prédicat `constraint/1`, la formule en deuxième argument est une contrainte réifiée, donc à valeur dans $\{0, 1\}$.

Une heuristique de type `conjunctive` et une heuristique `disjunctive` induisent respectivement un réordonnancement des termes des conjonctions et des disjonctions de l'arbre de modèle associé. L'arbre de modèle est la représentation arborescente d'une expression qui inclut notamment les appels de déclarations.

Une heuristique ***op***`(`e`) for `p`(`$\boldsymbol{X}$`)` définit un critère d'ordre ***op***`(`e`)` qui s'applique sur les termes unifiables avec la tête de règle $p(\boldsymbol{X})$. À chaque terme ainsi filtré est associée une évaluation du critère e fonction de l'instanciation, et l'ensemble des termes est trié selon un ordre croissant ou décroissant donné par ***op***. Une conjonction d'heuristiques de même type agrège les critères en un vecteur et définit un ordre lexicographique. Les termes d'une conjonction ou d'une disjonction sont réordonnés selon le premier critère du vecteur, puis le second en cas d'égalité, et ainsi de suite.

Exemple 10. *Considérons un modèle* `Cream` *du problème des n-reines (cf. Sec. 2.2).*

Le but du modèle `Cream` *correspondant pose les contraintes et demande la recherche de solution :*

```
? let(N = 4,
      B = board(N),
  queens_constraints(B, N) and
  queens_labeling(variables(B), N)).
```

La stratégie de branchement est définie par la formule d'affectations de variables (cf. Sec. 4.3) exprimée dans la déclaration `queens_labeling/2` *:*

```
queens_labeling(Vars, N) -->
  search(h(N), forall(Var in Vars, queens_labeling_var(Var, N))).

queens_labeling_var(Var) -->
  exists(Val in [1 .. N], queens_labeling_val(Var, Val)).

queens_labeling_val(Var, Val) -->
  Var = Val.
```

Sur cette formule du modèle, l'heuristique disjonctive middle out *(cf. Sec. 4.3) est appliquée. L'heuristique* `h(N)` *filtre les règles de la forme* `queens_labeling_val(Var, Val)` *dans la sous formule disjonctive et la réordonne en plaçant avant les autres les règles qui minimisent* `abs(N/2 -Val)` *:*

```
heuristic h(N) =
  disjunctive(least(abs(N/2 - Val)) for queens_labeling_val(Var, Val)).
```

Le code intermédiaire produit par $\longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow$ *est le code non-déterministe suivant qui aura pour effet d'essayer, pour chaque variable, les valeurs du milieu vers les bornes du domaine :*

```
search(Q_1_1 = 2 or Q_1_1 = 3 or Q_1_1 = 1 or Q_1_1 = 4 and
       Q_2_1 = 2 or Q_2_1 = 3 or Q_2_1 = 1 or Q_2_1 = 4 and
       Q_3_1 = 2 or Q_3_1 = 3 or Q_3_1 = 1 or Q_3_1 = 4 and
       Q_4_1 = 2 or Q_4_1 = 3 or Q_4_1 = 1 or Q_4_1 = 4)
```

Mode statique et mode dynamique

Par défaut, ou dans la portée du prédicat `static/1`, une expression est développée statiquement par la compilation (*cf.* section 6.1). Dans la portée du prédicat `dynamic/1`, elle est évaluée dynamiquement (*cf.* section 6.2).

Modèles et déclarations

Un modèle `Cream` consiste en une suite de déclarations. Une déclaration est soit un import de module, une déclaration d'objet (constructeur et fonctions), une déclaration de règle (prédicats), une déclaration de domaine énuméré, une spécification d'heuristique ou simplement un but (requête).

Les *déclarations récursives* et les *déclarations multiples* d'un même symbole de tête sont *interdites*.

Définition 16 (Variables libres)**.** *On définit l'ensemble* $\mathrm{fv}(e)$ *des variables libres d'une expression* e *par récurrence :*

$$\begin{aligned}
\mathrm{fv}(X) &= \{X\} \\
\mathrm{fv}(e \texttt{ op } e') &= \mathrm{fv}(e) \cup \mathrm{fv}(e') \\
\mathrm{fv}(e \texttt{ rel } e') &= \mathrm{fv}(e) \cup \mathrm{fv}(e') \\
\mathrm{fv}(e \texttt{ logop } e') &= \mathrm{fv}(e) \cup \mathrm{fv}(e') \\
\mathrm{fv}(\texttt{let(}X \texttt{ := } e'\texttt{, } e\texttt{)}) &= \mathrm{fv}(e) \setminus \{X\} \\
\mathrm{fv}(\texttt{foldl(}A \texttt{ from } i\texttt{, } X \texttt{ in } l\texttt{, } e\texttt{)}) &= \mathrm{fv}(e) \setminus \{A, X\} \\
\mathrm{fv}(f\texttt{(}X_1\texttt{,...,}X_n\texttt{) = } e\texttt{.}) &= \mathrm{fv}(e) \setminus \{X_1, \ldots, X_n\} \\
\mathrm{fv}(p\texttt{(}X_1\texttt{,...,}X_n\texttt{) --> } e\texttt{.}) &= \mathrm{fv}(e) \setminus \{X_1, \ldots, X_n\}
\end{aligned}$$

Les *variables libres* sont interdites dans les déclarations de règles. Dans les déclarations d'objets et de buts, elles *dénotent des variables à domaine fini*, les inconnues d'un problème, et *sont indexées sur la tête de déclaration*. Intuitivement, on entend par indexation d'une variable V sur une tête de déclaration $f\texttt{(}X_1\texttt{,...,}X_n\texttt{)}$ le nommage de V en fonction de l'appel $f\texttt{(}a_1\texttt{,...,}a_n\texttt{)}$, cf. Sec. 5.2.

Exemple 11. *Par exemple, la déclaration suivante introduit une variable à domaine fini dans le champ* `row` *pour chaque valeur distincte de* `I` *:* `queen(I) = {column :I, row :_}`*. La projection d'enregistrement* `queen(3):row` *permet d'obtenir cette variable qui sera nommée* $\texttt{Queen}_{(3,1)}$ *car la variable* `_` *est introduite par l'appel* `queen(3)` *et c'est la première à apparaître dans l'ordre syntaxique du corps de la déclaration.*

Afin d'éviter la collision de noms entre têtes de déclarations, le langage inclut un système de module simple qui préfixe les identifiants avec un nom de module, de façon similaire à [HF06].

5.2 Sémantique déclarative des modèles `Cream`

Nous décrivons ici la sémantique des succès de `Cream` qui caractérise les solutions d'un modèle $\mathcal{M}$. Une solution est une affectation de toutes les variables libres de $\mathcal{M}$ qui satisfait toutes les contraintes de $\mathcal{M}$.

Les variables libres vivent dans les déclarations d'objets et de buts. À chaque nouvelle instance d'un objet, un ensemble distinct de variables libres nommées selon une indexation sur la tête de déclaration instanciée.

Soit $O(\mathcal{M})$ l'ensemble des déclarations d'objets de $\mathcal{M}$, $R(\mathcal{M})$ l'ensemble des déclarations de règles, et $G(\mathcal{M})$ l'ensemble des buts de $\mathcal{M}$. Les buts sont considérés en conjonction : le but associé à $\mathcal{M}$ est $g(\mathcal{M}) = \bigwedge_{g \in G(\mathcal{M})} g$.

Exemple 12. *Soit le modèle* `Cream` *du problème des n-reines (cf. Sec. 2.2) suivant, avec à gauche les déclarations d'objets et le but et à droite les déclarations de règles :*

```
q(I) = {row : _, column : I}.

board(N) = map(I in [1 .. N], q(I)).

? let(N := 4,
      B := board(N),
  queens_constraints(B, N).
```

```
safe(L) -->
  forall(Q in L, forall(R in L,
    let(I := Q:column, J := R:column,
    I < J implies
    Q:row # R:row and
    Q:row # J - I + R:row and
    Q:row # I - J + R:row))).

queens_constraints(B, N) -->
  domain(B, 1, N) and safe(B).
```

Soit `Q_1_1`, `Q_2_1`, `Q_3_1`,`Q_4_1` *les variables issues respectivement des appels* `q(1)`, `q(2)`, `q(3)`, `q(4)` *et représentant les lignes des différentes reines sur l'échiquier.*

L'ensemble des solutions au problème des 4*-reines, c'est-à-dire des affectations de variables qui satisfont toutes les contraintes, est le suivant :* $\{($`Q_1_1` $= 2,$ `Q_2_1` $= 4,$ `Q_3_1` $= 1,$ `Q_4_1` $= 3),$ $($`Q_1_1` $= 3,$ `Q_2_1` $= 1,$ `Q_3_1` $= 4,$ `Q_4_1` $= 2)\}$

Indexation

L'indexation d'une variable libre V sur la tête de la déclaration $f(X_1, \ldots, X_n)$ qui la contient permet de produire un nom unique par θ en fonction de l'appel $f(a_1, \ldots, a_n)$ des termes `Cream` $\mathcal{T}_{\texttt{Cream}}$, de sorte que la variable indexée soit la même pour deux appels identiques. Si plusieurs variables vivent dans le corps d'une déclaration, elles sont différenciées par leur ordre d'apparition syntaxique.

$$\begin{array}{rrcl} \theta : & \mathcal{T}_{\texttt{Cream}} & \rightarrow & (\mathcal{V} \rightarrow \mathcal{V}) \\ & f(\boldsymbol{X}) & \mapsto & (Y \mapsto Z) \end{array}$$

où Z est fraîche vis-à-vis de $\boldsymbol{X} \cup \{Y\}$,
$\mathcal{T}_{\texttt{Cream}}$ représente les termes `Cream` et $\mathcal{V}$ les variables.

Afin d'assurer que l'indexation est réalisable, les arguments d'un appel à une déclaration d'objet sont restreints à être de la forme suivante :

$$\begin{array}{rcl} \textit{indexable} & ::= & \textit{integer} \\ & | & \texttt{[}\textit{indexable}\texttt{,...,}\textit{indexable}\texttt{]} \\ & | & \texttt{\{}\textit{ident}\texttt{:}\ \textit{expr}\texttt{,...,}\textit{ident}\texttt{:}\ \textit{expr}\texttt{\}}_{\textit{uid}} \end{array} \qquad \begin{array}{rcl} \textit{index} & ::= & \texttt{constant(}\textit{integer}\texttt{)} \\ & | & \texttt{[}\textit{index}\texttt{,...,}\textit{index}\texttt{]} \\ & | & \texttt{uid(}\textit{uid}\texttt{)} \end{array}$$

Sont indexables les entiers, les enregistrements et les listes d'indexables.

Chaque valeur indexable v définit un index $id(v)$.

$$id : \quad indexable \rightarrow index$$
$$\begin{cases} i \in integer & \mapsto \texttt{constant(}i\texttt{)} \\ \texttt{[}i_1\texttt{,...,}i_n\texttt{]} & \mapsto \texttt{[}id(i_1)\texttt{,...,}id(i_n)\texttt{]} \\ \texttt{\{}ident\texttt{:}\ expr\texttt{,...,}ident\texttt{:}\ expr\texttt{\}}_{uid} & \mapsto \texttt{uid(}uid\texttt{)} \end{cases}$$

Affectation

Soit fv(e) l'ensemble de variables libres dans une expression e.

Une affectation pour $\mathcal{M}$ est un couple $\nu = (\nu^G, \nu^O)$, où :

– ν^G est la famille d'affectations $\mathrm{fv}(g(\mathcal{M})) \rightarrow \mathcal{D}$
– ν^O est une famille d'affectations qui associe à tous les objets $o = f\texttt{(}X_1\texttt{,...,}X_n\texttt{)} \texttt{ = } e\texttt{.} \in O(\mathcal{M})$ et n-uplets $(i_1, \ldots, i_n) \in index^n$, une affectation $\nu^O_{f(i_1,\ldots,i_n)} : \mathrm{fv}(o) \rightarrow \mathcal{D}$

Tout affectation $\nu : var \rightarrow \mathcal{D}$ est étendue homomorphiquement sur la structure des termes `Cream` à la fonction $\tilde{\nu} : expr \rightarrow expr$.

Sémantique des succès

Soit $\rightarrow$ la définition de la sémantique des succès de `Cream`.

Nous décrivons ici comment $\rightarrow$ réduit les expressions du langage noyau de `Cream` selon une affectation de toutes les variables d'un modèle $\mathcal{M}$ par ν^G et ν^O.

La réduction d'un but est une valeur booléenne donnée par la réduction de son corps modulo l'affectation des variables libres. La réduction des appels à déclarations de règles et d'objets est le résultat de la réduction de leur corps, modulo la substitution des paramètres formels et l'affectation des variables libres.

Les affectations des variables libres dans les déclarations d'objets et dans les buts sont opérées par $\tilde{\nu}^O$, paramétrée par l'indexation de la tête instanciée, et par $\tilde{\nu}^G$, respectivement.

$$e \rightarrow \tilde{\nu}^G(e) \quad \text{si } \texttt{?}\ e\texttt{.} \in G(\mathcal{M})$$

$$f\texttt{(}e_1\texttt{,...,}e_n\texttt{)} \rightarrow \tilde{\nu}^O_{f(id(e_1),\ldots,id(e_n))}(e)[X_1 := e_1, \ldots, X_n := e_n] \quad \text{si } f\texttt{(}X_1\texttt{,...,}X_n\texttt{)} \texttt{ = } e\texttt{.} \in O(\mathcal{M}) \text{ et } (e_1, \ldots, e_n) \in indexable^n$$

$$p\texttt{(}e_1\texttt{,...,}e_n\texttt{)} \rightarrow e[X_1 := e_1, \ldots, X_n := e_n] \quad \text{si } p\texttt{(}X_1\texttt{,...,}X_n\texttt{)} \texttt{ --> } e\texttt{.} \in R(\mathcal{M}) \text{ et } V(e) \subseteq \{X_1, \ldots, X_n\}$$

La réduction d'une expression arithmétique est le résultat de son évaluation. La réduction d'une comparaison ou d'une formule logique est la réification du résultat de son évaluation ; le vrai ($\top$) est interprété par 1 et le faux ($\bot$) par 0.

Soit δ l'opérateur de réification : $\delta(\top) = 1$ et $\delta(\bot) = 0$.

$$
\begin{aligned}
n \in \mathbf{N} \ \texttt{op}\ n' \in \mathbf{N} &\to n \ op \ n' \\
n \in \mathbf{N} \ \texttt{rel}\ n' \in \mathbf{N} &\to \delta(n \ rel \ n') \\
n \in \{0,1\} \ \texttt{logop}\ n' \in \{0,1\} &\to \delta(n = 1 \ logop \ n' = 1) \\
\texttt{not}\ n \in \{0,1\} &\to \delta(n = 0) \\
\texttt{domain([}n_1\texttt{,...,}n_k\texttt{],}\ l \in \mathbf{N},\ u \in \mathbf{N}\texttt{)} &\to \forall_{i \in \{1,\ldots,k\}},\ \delta(\delta(n_i \geq l) = 1 \wedge \delta(n_i \leq u) = 1)
\end{aligned}
$$

$$
\begin{aligned}
\texttt{[}n \in \mathbf{N}\ \texttt{..}\ n' \in \mathbf{N}\texttt{]} &\to \begin{cases} \texttt{[}n,\ n+1\texttt{,...,}n'\texttt{]} & \text{si } n \leq n' \\ \texttt{[]} & \text{sinon} \end{cases} \\
\texttt{[}e_1\texttt{,...,}e_n\texttt{]}\ \texttt{++}\ \texttt{[}e'_1\texttt{,...,}e'_n\texttt{]} &\to \texttt{[}e_1\texttt{,...,}e_n\texttt{,}e'_1\texttt{,...,}e'_n\texttt{]} \\
\texttt{length([}e_1,\ \ldots,\ e_n\texttt{])} &\to n \\
\texttt{nth(}i \in \{1,\ldots,n\}\texttt{, [}e_1,\ \ldots,\ e_n\texttt{])} &\to e_i \\
\texttt{\{}f_1\texttt{:}\ e_1\texttt{,...,}f_n\texttt{:}\ e_n\texttt{\}:}f_i &\to e_i \\
\texttt{let(}X\ \texttt{:=}\ v,\ e\texttt{)} &\to e[X := v] \\
\texttt{foldl(}A\ \texttt{from}\ i\texttt{,}X\ \texttt{in [}e_1\texttt{,...,}e_n\texttt{],}e\texttt{)} &\to \ i \rhd_e e_1 \rhd_e \cdots \rhd_e e_n \\
& \quad \text{où } u \rhd_e v = e[A := u, X := v]
\end{aligned}
$$

Une expression g participe à la caractérisation de l'ensemble des solutions d'un modèle $\mathcal{M}$. Quand g est une formule, l'interpréter comme un arbre de recherche n'altère pas l'ensemble des solutions de $\mathcal{M}$. Quand g est une expression quelconque, la développer complètement statiquement ou la compiler en du code procédural dynamique n'altère pas l'ensemble des solutions de $\mathcal{M}$.

$$
\left.
\begin{aligned}
&\texttt{minimize(}g\texttt{,}\ c\texttt{)} \\
&\texttt{maximize(}g\texttt{,}\ c\texttt{)} \\
&\texttt{search(}h\texttt{,}\ g\texttt{)} \\
&\texttt{constraint(}g\texttt{)} \\
&\texttt{static(}g\texttt{)} \\
&\texttt{dynamic(}g\texttt{)}
\end{aligned}
\right\} \to g
$$

Une solution d'un modèle $\mathcal{M}$ est une affectation $\nu = (\nu^G, \nu^O)$ pour laquelle le but de $\mathcal{M}$ est réduit vers 1 par la réduction $\to$ associée à la sémantique succès.

Définition 17. *Soit $\mathcal{M}$ un modèle* `Cream`*. La sémantique succès $\mathcal{S}_s(m)$ de $\mathcal{M}$ est l'ensemble des solutions de $\mathcal{M}$:*

$$\mathcal{S}_s(\mathcal{M}) = \{(\nu^G, \nu^O) \mid \nu^G(q(\mathcal{M})) \xrightarrow{*} 1\}$$

Exemple 13. *Reprenons le modèle* `Cream` *du problème des n-reines de l'exemple 12 et vérifions que la réduction du modèle par* $\rightarrow$*, selon une affectation solution au problème, donne bien* 1*.*

Soit ν_{queens_4} *une affectation qui correspond à une solution au problème des* 4*-reines. Les seules variables libres du modèle sont introduites par la déclaration* `q(I)`*. Considérons alors uniquement les affectations de la famille* $\nu^O_{\text{queens}_4}$ *qui concernent* `q(I)` *:*

$$\begin{array}{lcll} \tilde{\nu}^O_{\texttt{q(constant(1))}} : & \texttt{Q\_1\_1} & \mapsto & 2 \\ \tilde{\nu}^O_{\texttt{q(constant(2))}} : & \texttt{Q\_2\_1} & \mapsto & 4 \\ \tilde{\nu}^O_{\texttt{q(constant(3))}} : & \texttt{Q\_3\_1} & \mapsto & 1 \\ \tilde{\nu}^O_{\texttt{q(constant(4))}} : & \texttt{Q\_4\_1} & \mapsto & 3 \end{array}$$

La réduction par $\rightarrow$ *pour cette affectation donne :*

`board(4)`	$\xrightarrow{*}$	`[{row:2, column:1}, {row:4, column:2},` `{row:1, column:3}, {row:3, column:4}]`
`domain(board(4), 1, 4)`	$\xrightarrow{*}$	1
`safe(board(4))`	$\xrightarrow{*}$	1
`queens_constraints(board(4), 4)`	$\xrightarrow{*}$	$\delta(1 = 1 \wedge 1 = 1)$

Et ainsi `let(N :=4, B =board(N), queens_constraints(B, N))` $\xrightarrow{*} 1$*.*

L'affectation $\nu_{\text{queens}_4} = (\nu^G_{\text{queens}_4}, \nu^O_{\text{queens}_4})$ *est bien une solution au modèle des* 4*-reines.*

Chapitre 6

Compilation

Sommaire

Un modèle `Cream` se compile en un programme d'optimisation ou de satisfaction de contraintes sur domaines finis, avec contraintes réifiées et contraintes globales.

Le schéma de compilation fondamental, dit statique, est formalisé par un système de réécriture de termes qui procède par expansion des règles des modèles. Ce schéma produit des CSPs, du code efficace « plat », mais il arrive que malgré l'évaluation partielle, une instance de problème soit de trop grande taille pour être complètement développée. De plus, il impose l'instanciation des structures sur lesquelles les itérateurs déroulent les expressions du modèle. Pour prendre en charge ces problèmes de grande taille ou dynamiques, un schéma de compilation par génération de code procédural est également proposé. Ce second schéma produit des programmes CLP incluant des définitions (récursives) du langage cible. La confluence, la correction vis-à-vis de la sémantique, et des bornes de complexité sur la taille des programmes générés sont prouvées pour ces transformations.

La compilation d'un modèle consiste en deux phases : une phase d'expansion du but qui, des termes `Cream` $\mathcal{T}_{\texttt{Cream}}$, produit un terme du langage intermédiaire $\mathcal{T}_{\texttt{Cream}_{\texttt{Core}}}$; et une phase de génération de code qui, des termes $\mathcal{T}_{\texttt{Cream}_{\texttt{Core}}}$ produit soit un but principal `SICStus-Prolog` [C+07] dans les termes $\mathcal{T}_{\texttt{SICStus-Prolog}}$, soit un but `Choco-Java` dans les termes $\mathcal{T}_{\texttt{Choco-Java}}$. Potentiellement, le terme du langage intermédiaire peut contenir des appels et être alors accompagné d'un ensemble de déclarations qui sont transformées par exemple en définitions de clauses `SICStus-Prolog`.

$$\mathcal{T}_{\texttt{Cream}} \quad \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow \quad \mathcal{T}_{\texttt{Cream}_{\texttt{Core}}} \quad \longrightarrow\!\!\langle \text{stc ; cg} \rangle\!\!\longrightarrow \quad \mathcal{T}_{\texttt{SICStus-Prolog}} + \mathcal{T}_{\texttt{Java-Choco}}$$

Plus précisément, deux transformations peuvent intervenir dans la compilation :
- la transformation par expansion de règles $\longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow$, qui développe complètement les déclarations, produit un programme de contraintes « plat » qui se comprend comme un *problème de satisfaction de contraintes* classique (section 6.1).
- la seconde transformation par génération de code procédural $\longrightarrow\!\langle \text{dyn} \rangle\!\longrightarrow$, qui se retient de développer les déclarations mais produit plutôt des appels et des traductions de celles-ci, c'est-à-dire un *programme logique avec contraintes* (section 6.2).

Nous ne décrivons pas la génération de code $\longrightarrow\!\langle \text{stc ; cg} \rangle\!\longrightarrow$ qui consiste à traduire le code intermédiaire de syntaxe fonctionnelle en les idiomes `SICStus-Prolog` ou `Java-Choco` et qui n'introduit pas de difficulté particulière.

Avant d'appliquer les transformations, les sucres syntaxiques sont traduits en code intermédiaire. Une erreur de compilation (`error`) est produite pour toute expression qui ne peut être réécrite. Les contraintes globales prédéfinies sont traduites par des règles de réécriture spécifiques et ne sont pas décrites ici.

6.1 Compilation vers CSP par expansion des règles

Le schéma de compilation associé au *mode statique* [FM09], dit par expansion de règles, est défini par deux transformations qui produisent du code intermédiaire : $\longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow$ développe ou réduit complètement un but en code déterministe qui pose les contraintes et passe le relais pour les sous-expressions en *mode recherche statique* à $\longrightarrow\!\langle \substack{\text{stc} \\ \text{srch}} \rangle\!\longrightarrow$ qui développe et réordonne les formules concernées en code non déterministe. Les règles de compilation du réordonnancement des formules ne sont pas présentées pour ce schéma, pour une présentation voir la section 6.2.

Dans ce schéma de compilation, les expressions closes sont simplifiées implicitement par un mécanisme d'*évaluation partielle*, de façon similaire à Zinc [dlBMRW06] et à Essence [FHJ+08]. Mécanisme qui permet de limiter la taille du code généré et un surcoût potentiel à la compilation dû aux structures de données manipulées. Le code intermédiaire suit la syntaxe des modèles `Cream`, et se trouve enrichie notamment d'un opérateur de réification.

6.1.1 Transformation en code déterministe

Commençons la revue des règles de réécriture de la compilation par les expressions de base du langage pour finir avec les déclarations.

Opérateurs arithmétiques

L'évaluation partielle s'applique notamment sur la transformation des expressions arithmétiques. La réification plonge les booléens produits par les comparaisons dans les entiers.

ARITHMÉTIQUE

$$\frac{e_1 \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow e'_1 \qquad e_2 \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow e'_2}{e_1 \ \texttt{op} \ e_2 \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow e'_1 \ \texttt{op} \ e'_2}$$

COMPARAISONS

$$\frac{e_1 \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow e'_1 \qquad e_2 \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow e'_2}{e_1 \ \texttt{rel} \ e_2 \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow \texttt{reify}(e'_1 \ \texttt{rel} \ e'_2)}$$

Opérateurs logiques

L'évaluation partielle et la réification interviennent aussi dans la transformation des expressions logiques. L'élimination des négations dans les formules en les descendant vers les contraintes fait partie de la transformation mais n'est pas présentée.

OPÉRATEURS BINAIRES

$$\frac{e_1 \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow e_1' \qquad e_2 \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow e_2'}{e_1 \ \texttt{logop} \ e_2 \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow \texttt{reify(}e_1' \texttt{ = 1 logop } e_2' \texttt{ = 1)}}$$

NÉGATION

$$\frac{e \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow e'}{\texttt{not } e \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow \texttt{reify(}e' \texttt{ = 0)}}$$

Variables à domaine fini

Les variables à domaine fini sont des structures de données qui portent une place pour une valeur entière et une place pour un nom formé de la tête de déclaration instanciée qui l'a produite et la position de la variable dans le corps de déclaration.

La fonction prédéfinie `variables(`e`)` donne l'ensemble des variables libres de e après réduction, donc des variables à domaine fini.

ENSEMBLE DES VARIABLES

$$\frac{e \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow e'}{\texttt{variables(}e\texttt{)} \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow \texttt{[}V_1\texttt{,...,}V_n\texttt{]}} \ \text{fv}(e') \ = \ \{V_1, \ldots, V_n\}$$

`variable_name(`e`)` dénote un entier représentant le nom de la variable V en laquelle se réduit e. La fonction $\eta : \mathcal{V} \to \mathbf{N}$ associe à chaque variable libre portant son nom un entier unique. `variable_def(`e`)` dénote la tête de déclaration instanciée qui a produit la variable V en laquelle se réduit e. La fonction $\delta : \mathcal{V} \to \mathcal{T}_{\texttt{Cream}}$ extrait du nom d'une variable libre la tête de déclaration qui l'a produite.

NOM D'UNE VARIABLE

$$\frac{e \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow V}{\texttt{variable\_name(}e\texttt{)} \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow \eta(V)}$$

SOURCE D'UNE VARIABLE

$$\frac{e \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow V}{\texttt{variable\_def(}e\texttt{)} \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow \delta(V)}$$

Le prédicat `domain/3` pose le domaine de toutes les variables contenues dans le terme résultat de la réduction d'une expression e quelconque.

POSE DE DOMAINE

$$\frac{\begin{array}{c}\texttt{variables}(e) \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow [V_1,\ \ldots,\ V_n] \\ e_1 \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow l \qquad e_2 \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow u\end{array}}{\texttt{domain}(e,\ e_1,\ e_2) \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow \texttt{domain}([V_1,\ \ldots,\ V_n],\ l,\ u)} \begin{cases} l, u \in \mathbf{N} \\ l \leq u \end{cases}$$

BORNES ET CARDINALITÉ

$$\frac{e \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow V}{p(e) \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow p(V)} \quad \begin{cases} p \in \{\texttt{domain\_min}, \\ \qquad \texttt{domain\_max}, \\ \qquad \texttt{domain\_size}\} \end{cases}$$

Listes

Les listes peuvent être construites par énumération, par intervalle et par concaténation. Pour une construction par intervalle, les bornes doivent être instanciées statiquement (au moment de la compilation), et l'intervalle est alors réduit en la liste d'entiers appartenant à l'intervalle au sens large.

ENUMÉRATION

$$\frac{e_1 \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow e'_1 \qquad \ldots \qquad e_n \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow e'_n}{[e_1, \ldots, e_n] \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow [e'_1, \ldots, e'_n]}$$

INTERVALLE

$$\frac{e_1 \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow l \qquad \ldots \qquad e_n \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow u}{[e_1\ ..\ e_2] \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow [l,\ l+1, \ldots,\ u]} \begin{cases} l, u \in \mathbf{N} \\ l \leq u \end{cases}$$

CONCATÉNATION

$$\frac{l_1 \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow [d_1, \ldots, d_n] \qquad l_2 \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow [e_1, \ldots, e_m]}{l_1\ \texttt{++}\ l_2 \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow [d_1, \ldots, d_n,\ e_1, \ldots, e_n]}$$

Le prédicat prédéfini $\texttt{pos}(e,\ l)$, dual de $\texttt{nth}(e,\ l)$, renvoit l'indice de la première occurence de e' dans $[e_1, \ldots, e_n]$.

LONGUEUR

$$\frac{l \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow [e_1, \ldots, e_n]}{\texttt{length}(l) \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow n}$$

ÉLÉMENT À L'INDEX

$$\frac{e \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow i \qquad l \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow [e_1, \ldots, e_n]}{\texttt{nth}(e,\ l) \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow e_i}\ i \in \mathbf{N}$$

INDEX D'UN ÉLÉMENT

$$\frac{e \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow e' \qquad l \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow [e_1, \ldots, e_n]}{\texttt{pos}(e,\ l) \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow k}\ k = \min_{i \in \{1,\ldots,n\}} \{i \mid e_i = e'\}$$

Enregistrements

A l'instar des opérations sur les listes, une projection nécessite que l'enregistrement concerné soit instancié statiquement.

CONSTRUCTION

$$\frac{e_1 \longrightarrow\langle \text{stc} \rangle\longrightarrow e'_1 \qquad \ldots \qquad e_n \longrightarrow\langle \text{stc} \rangle\longrightarrow e'_n}{\texttt{\{}f_1\texttt{:}\ e_1\texttt{,}\ldots\texttt{,}f_n\texttt{:}\ e_n\ \texttt{\}} \longrightarrow\langle \text{stc} \rangle\longrightarrow \texttt{\{}f_1\texttt{:}\ e'_1\texttt{,}\ldots\texttt{,}f_n\texttt{:}\ e'_n\ \texttt{\}}}$$

PROJECTION

$$\frac{e_i \longrightarrow\langle \text{stc} \rangle\longrightarrow e'_i}{\texttt{\{}f_1\texttt{:}\ e_1\texttt{,}\ldots\texttt{,}f_n\texttt{:}\ e_n\ \texttt{\}}\texttt{:}f_i \longrightarrow\langle \text{stc} \rangle\longrightarrow e'_i}\ f_i \in \{f_1, \ldots, f_n\}$$

Combinateurs et liaison

Les combinateurs, tous réécrits en `foldl`, réduisent complètement les listes qu'ils prennent en argument selon l'expression de leur dernier argument e, à la condition que les listes soient instanciées statiquement.

RÉDUCTION PAR LA GAUCHE

$$\frac{i \longrightarrow\langle \text{stc} \rangle\longrightarrow i_0 \qquad l \longrightarrow\langle \text{stc} \rangle\longrightarrow [e_1, \ldots, e_n]}{i_0 \rhd_e e_1 \longrightarrow\langle \text{stc} \rangle\longrightarrow i_1 \qquad i_1 \rhd_e e_2 \longrightarrow\langle \text{stc} \rangle\longrightarrow i_2 \qquad \ldots \qquad i_{n-1} \rhd_e e_n \longrightarrow\langle \text{stc} \rangle\longrightarrow i_n}$$
$$\texttt{foldl(}A\ \texttt{from}\ i\texttt{,}\ X\ \texttt{in}\ l\texttt{,}\ e\texttt{)} \longrightarrow\langle \text{stc} \rangle\longrightarrow i_n$$

où $u \rhd_e v = e[A := u, X := v]$

Les substitutions induites par la liaison de variables en général, et les constructions `let` en particulier, sont effectué modulo alpha-conversion.

LIAISON

$$\frac{v \longrightarrow\langle \text{stc} \rangle\longrightarrow v'}{\texttt{let(}X\ \texttt{:=}\ v\texttt{,}\ e\texttt{)} \longrightarrow\langle \text{stc} \rangle\longrightarrow e[X := v']}$$

Exemple 14. *À l'issu du renommage,* `let(X:=1, exists(X in [5,3,X], X+X=2))` *donne l'expression* `let(X:=1, foldl(A from 0, Y in [5,3,X], A or Y+Y=2))`*, où* `Y` *et* `A` *sont des variables fraîches, est évaluée via réification à* `1`*, c'est-à-dire à vrai.*

Mode statique et mode dynamique

Il est possible d'évaluer une expression dynamiquement plutôt que statiquement avec le prédicat `dynamic/1` qui marque l'application du schéma de compilation dit par génération de code procédural (section 6.2) aux expressions dans sa portée.

$$\frac{e \longrightarrow\langle \text{dyn} \rangle\longrightarrow e'}{\texttt{dynamic(}e\texttt{)} \longrightarrow\langle \text{stc} \rangle\longrightarrow e'} \qquad \frac{e \longrightarrow\langle \text{stc} \rangle\longrightarrow e'}{\texttt{static(}e\texttt{)} \longrightarrow\langle \text{stc} \rangle\longrightarrow e'}$$

Mode contrainte et mode recherche

Par défaut, une formule `Cream` f définit une contrainte réifiée. Dans un prédicat `search(`f`)`, f définit une stratégie de branchement.

$$\text{RECHERCHE}\quad \frac{\texttt{nil} \vdash f \multimap\!\langle {}^{\text{stc}}_{\text{srch}}\rangle\!\longrightarrow f'}{\texttt{search(}f\texttt{)} \multimap\!\langle \text{stc}\rangle\!\longrightarrow f'} \qquad \text{CONTRAINTE}\quad \frac{f \multimap\!\langle \text{stc}\rangle\!\longrightarrow f'}{\texttt{constraint(}f\texttt{)} \multimap\!\langle \text{stc}\rangle\!\longrightarrow f'}$$

Un prédicat `minimize(`f`, `c`)` minimise la valeur de la variable à domaine fini V dénoté par c selon la méthode de recherche arborescente par séparation et évaluation. f est une formule implicitement interprétée comme une stratégie de branchement qui contraint V à une affectation. Opérationnellement, l'arbre induit par f est itérativement évalué en resserrant progressivement la borne supérieure de c jusqu'à la preuve d'optimalité.

$$\text{OPTIMISATION}\quad \frac{\texttt{search(}e\texttt{)} \multimap\!\langle {}^{\text{stc}}_{\text{srch}}\rangle\!\longrightarrow e' \qquad c \multimap\!\langle \text{stc}\rangle\!\longrightarrow \text{V}}{p\texttt{(}e\texttt{, }c\texttt{)} \multimap\!\langle \text{stc}\rangle\!\longrightarrow p\texttt{(}e'\texttt{, }V\texttt{)}} \begin{cases} p \in \{\texttt{minimize}, \\ \qquad \texttt{maximize}\} \end{cases}$$

Déclarations et appels

On considère un modèle `Cream` $\mathcal{M}$. Soit $D(\mathcal{M}) = R(\mathcal{M}) \cup O(\mathcal{M})$, $E(\mathcal{M})$ l'ensemble des déclarations de domaines énuméré.

Une variable libre `Cream` dénote une variable à domaine fini ; c'est une structure de donnée qui porte une place pour sa valeur et un nom. Soit $\mathcal{V}$ l'ensemble des symboles de variable des termes `Cream`. La fonction θ (cf. Sec. 5.2) génère une fonction de renommage qui donne un nom unique à chacune des variables libres pour chaque déclaration d'objet $f\texttt{(}X_1\texttt{,}\ldots\texttt{,}X_n\texttt{)} = e$ selon l'appel $f(\boldsymbol{x})$.

La compilation commence par l'expansion de la conjonction des buts de $\mathcal{M}$.

$$\text{BUT}\quad g_1 \texttt{ and } \ldots \texttt{ and } g_n \multimap\!\langle \text{stc}\rangle\!\longrightarrow g'\theta(goal) \begin{cases} \texttt{? } g_1\texttt{.} \in G(\mathcal{M}) \\ \ldots \\ \texttt{? } g_n\texttt{.} \in G(\mathcal{M}) \end{cases}$$

Le but peut contenir des appels à des déclarations de règles ou d'objets. Les déclarations de règles ne doivent pas contenir de variables libres.

RÈGLE
$$\frac{a_1 \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow a'_1 \quad \dots \quad a_n \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow a'_n \qquad e[X_1 := a'_1, \dots, X_n := a'_n] \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow e'}{p\texttt{(}a_1\texttt{,}\dots\texttt{,}a_n\texttt{)} \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow e'} \begin{cases} r = p\texttt{(}X_1\texttt{,}\dots\texttt{,}X_n\texttt{)}\ \texttt{-->}\ e\texttt{.} \in R(\mathcal{M}) \\ \text{fv}(r) = \emptyset \end{cases}$$

Les déclarations d'objets renomment les variables libres en fonction de l'instanciation de la tête $f\texttt{(}a_1\texttt{,}\dots\texttt{,}a_n\texttt{)}$ et de leur ordre d'apparition dans e par θ, après substitution des paramètres formels.

DÉCLARATION
$$\frac{a_1 \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow a'_1 \quad \dots \quad a_n \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow a'_n \qquad e[X_1 := a'_1, \dots, X_n := a'_n] \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow e'}{f\texttt{(}a_1\texttt{,}\dots\texttt{,}a_n\texttt{)} \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow e'\theta(f(a'_1, \dots, a'_n))}\ f\texttt{(}X_1\texttt{,}\dots\texttt{,}X_n\texttt{)}\ \texttt{=}\ e\texttt{.} \in O(\mathcal{M})$$

Exemple 15. *La déclaration* `box(L)={size:L, origin:[_,_,_]}` *introduit trois variables à domaine fini dans le champs* `origin`*, différentes pour chaque valeur distincte de* `L`*. La projection d'enregistrement* `box([2,3,2]):origin` *donne la liste des trois variables suivante :* $\texttt{[Box\_1}_{\texttt{(box([2,3,2]),1)}}\texttt{, Box\_2}_{\texttt{(box([2,3,2]),2)}}\texttt{, Box\_3}_{\texttt{(box([2,3,2]),3)}}\texttt{]}$.

Dans une déclaration de domaine énuméré, chaque nom e_i est associé à l'entier naturel i, unique dans la déclaration.

DOMAINE ÉNUMÉRÉ
$$\frac{}{d \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow \texttt{[}1, 2, \dots, n\texttt{]}} \begin{cases} \texttt{domain}\ d\ \texttt{=}\ \{e_1, \dots, e_n\} \in E(\mathcal{M}) \\ \text{fv}(\{e_1, \dots, e_n\}) = \emptyset \end{cases}$$

VALEUR
$$\frac{e \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow e'}{e \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow k} \begin{cases} \texttt{domain}\ d\ \texttt{=}\ \{e_1, \dots, e_n\} \in E(\mathcal{M}) \\ e_k = e', k \in \{1, \dots, n\} \\ \text{fv}(\{e_1, \dots, e_n\}) = \emptyset \end{cases}$$

Exemple 16. *Étant données les déclarations* `domain rvb ={rouge, vert, bleu}` *et* `jaune ={comp :[rouge, vert]}`*, on a* `nth(1, jaune:comp)` $\longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow$ `1`.

Les symboles déclarés dans les déclarations de domaine énuméré vivent dans le même monde que les symboles de déclarations de règles et de déclarations. Ainsi, sachant que les déclarations multiples sont interdites, si la déclaration précédente est présente dans un modèle, aucune règle ou déclaration ne peut être de la forme f `=` e*,* $f \in \{rouge, vert, bleu\}$.

6.1.2 Transformation en code non déterministe

Par la transformation $\longrightarrow\!\langle {}^{\text{stc}}_{\text{srch}} \rangle\!\longrightarrow$, l'opérateur de conjonction `and` devient un opérateur de séquence ; et l'opérateur de disjonction `or` devient un opérateur de choix non

déterministe. Et $\multimap\!\langle {}^{\text{stc}}_{\text{srch}}\rangle\!\longrightarrow$ repose sur $\multimap\!\langle \text{stc}\rangle\!\longrightarrow$ pour toutes les autres expressions.

La formule f est développée et ladite altération des opérateurs est appliquée.

$$\texttt{nil} \vdash f \multimap\!\langle {}^{\text{stc}}_{\text{srch}}\rangle\!\longrightarrow f'$$

6.1.3 Confluence, terminaison, complexité et correction

Confluence de la compilation statique

L'interdiction de déclarations multiples et la restriction aux têtes linéaires, c'est-à-dire qui ne contiennent en arguments que des variables distinctes, permettent de montrer la confluence de la compilation [FM09]. Autrement dit, quelque soit l'ordre d'application des règles de réécriture, quelque soit la stratégie de réécriture, elles généreront le même programme de contraintes pour un même modèle d'entrée.

Proposition 2. *Pour tout modèle* `Cream`*, le système de réécriture de termes associé à la compilation* $\longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow$ *est confluent.*

Preuve 1. *Montrons que le système de réécriture de termes* $\longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow$ *est orthogonal, i.e. linéaire à gauche et non recouvrant, ce qui implique la confluence du système [Ros73].*

D'abord, les têtes de règles de réécriture de $\longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow$ *associées aux déclarations sont formées à partir d'un symbole et de variables distinctes comme arguments, d'où la linéarité à gauche de ces règles. Par ailleurs, les déclarations multiples sont interdites, et le renommage des variable par indexation sur la tête de déclaration est déterministe, d'où le non-recouvrement de ces règles. Le système de réécriture des déclarations est donc orthogonal.*

Ensuite, par définition, les autres règles $\longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow$ *associées aux prédicats prédéfinis et aux primitives ne se recouvrent pas. Elles réécrivent toutes un symbole différent et sont linéaires à gauche. Le système de réécriture des prédicats prédéfinis est orthogonal.*

Donc le système de réécriture de termes $\longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow$ *est orthogonal.* □

Précisons que cette preuve ne fait pas l'hypothèse de la terminaison [1]. La propriété de confluence des règles de compilation $\longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow$ serait donc valide si les déclarations récursives étaient autorisées. Contrairement à Zinc, `Cream` exclut cette possibilité pour assurer la terminaison de la compilation statique, une propriété de sécurité confortable du point de vue d'un non-programmeur.

Terminaison de la compilation statique

Supposons, sans perte de généralité, que les modèles `Cream` ne contiennent qu'un seul but défini par une règle nommée *solve*.

Définition 18. *Étant donné un modèle* $\mathcal{M}$*, soit* $\rho(s)$ *le* rang de déclaration *d'un symbole* s *défini inductivement par :*

- $\rho(s) = 0$ *si* s *n'est pas le symbole de tête d'une déclaration ou d'une règle de* $\mathcal{M}$*,*
- $\rho(s) = n + 1$ *si* s *est le symbole de tête d'une déclaration ou d'une règle de* $\mathcal{M}$*, et* n *est le plus grand rang de déclaration des symboles de la partie droite de la déclaration.*

Le rang de déclaration de $\mathcal{M}$ *est le plus grand rang de déclaration des symboles de* $\mathcal{M}$*.*

1. Quand l'hypothèse de terminaison est faite, la condition de non-recouvrement, ou plus généralement la confluence des paires critiques [Ter03, HF07], suffit à prouver la confluence sans la condition de linéarité.

Proposition 3. *Pour tout modèle* `Cream`*, le système de réécriture de termes* $\longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow$ *est noethérien.*

Preuve 2. *Montrons que* $\longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow$ *termine en montrant que la réécriture des termes prédéfinis et la réécriture des déclarations de règles et d'objets terminent.*

Par définition, les règles de $\longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow$ *qui concernent les prédicats et fonctions prédéfinis et les primitives n'accroissent pas les rangs de déclaration et terminent.*

Considérons les règles de réécriture de $\longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow$ *associées aux règles et aux déclarations. Un système de réécriture* $\rightarrow$ *termine s'il existe un ordre de simplification* $\succ_{RPO}$ *tel que pour toute règle* $r \rightarrow l$, $r \succ_{RPO} l$ *[Der79]. Montrons donc que pour toute réduction* $t \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow t'$ *on a* $t \succ_{RPO} t'$*. Soit* $\succ$ *l'ordre sur les symboles de déclaration :* $s \succ t \iff \rho(s) > \rho(t)$*. Soit* $\succ_{RPO}$ *l'ordre de simplification sur les termes induit par* $\succ$*. Soit la réduction* $t \longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow t'$ *et la déclaration* `f(X1,...,Xn) = u.` $\in O(\mathcal{M})$*, tels que* $t = f(a_1, \ldots, a_n)$ *et* $t' = u[a_1, \ldots, a_n]$*. Soit* g *le symbole d'un sous-terme* s *de* u *et* s_i *les sous-termes de* s*. Par définition,* $f \succ g$ *et* $\succ_{RPO}$ *est monotone vis-à-vis du contexte, et stable vis-à-vis de la substitution [Hof92], donc* $\forall i \in \{1, \ldots, n\} : t \succ_{RPO} s_i$*. Alors, pour tout symbole de sous-terme* g *de* u*,* $f \succ g \implies f(X_1, \ldots, X_n) \succ_{RPO} u$*, donc* $t \succ_{RPO} t'$*. Donc le système de réécriture* $\longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow$ *termine.* □

Complexité de la compilation statique

Une borne de complexité sur la taille des programmes générés peut être obtenue :

Définition 19. *Étant donné un modèle* `Cream` $\mathcal{M}$*, le* rang de combinateur $\alpha(s)$ *d'un symbole* s *est défini inductivement par :*

- $\alpha(s) = 0$ *si* s *n'est pas le symbole de tête d'une déclaration ou d'une règle dans* $\mathcal{M}$*,*
- $\alpha(s) = max\{n + \alpha(s') \mid s(x_1, \ldots, x_n)$ `->` $r \in D(m)$*,* r *contient une imbrication de* n *combinateurs sur une expression contenant le symbole* $s'\}$*.*

Le rang de combinateur de $\mathcal{M}$ *est le plus grand rang de combinateur des symboles de* $\mathcal{M}$*.*

Proposition 4. *Pour tout modèle* `Cream` $\mathcal{M}$*, la taille du programme généré par* $\longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow$ *est en* $\mathbf{O}(l^a \cdot b^r)$*, où* l *est la plus grande longueur de liste dans* $\mathcal{M}$ *(ou au moins 1),* a *est le rang de combinateur de* $\mathcal{M}$*,* b *est la plus grande taille de partie droite des règles et déclarations de* $\mathcal{M}$*, et* r *est le rang de déclaration de* $\mathcal{M}$*.*

Preuve 3. *La preuve se fait par induction sur* a*.*

Dans le cas de base, $a = 0$*, il n'existe pas de combinateur dans* $\mathcal{M}$*, et la taille du programme généré est borné linéairement par* r *duplications de corps ou partie droite de règle, i.e. est en* $\mathbf{O}(b^r)$*.*

Dans le cas inductif, $a > 0$*, considérons d'abord la taille du programme généré sans réécrire des occurences superficielles (ou ultrapériphériques) des combinateurs. Par induction, la taille est en* $\mathbf{O}(l^{a-1} \cdot b^r)$*. Maintenant, ce programme généré peut être dupliqué au plus* l *fois par les combinateurs superficiels, d'où la taille en* $\mathbf{O}(l^a \cdot b^r)$ *selon cette stratégie. Par la propriété de confluence 2, le programme généré est indépendant de la stratégie de réécriture.*

La taille du programme généré par $\longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow$ *est donc en* $\mathbf{O}(l^a \cdot b^r)$ *quelque soit la stratégie.* □

Exemple 17. *Soient le modèle* `Cream` *du problème des n-reines (à gauche) et le code généré (à droite) par la compilation* $\longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow$ *:*

```
q(I) = {row = _, column = I}.
board(N) = map(I in [1 .. N], q(I)).

safe(L) -->
  all_different(L) and
  forall(Q in L,
    forall(R in L,
      let(I:=Q:column, J:=R:column,
        I < J implies
        Q:row # J - I + R:row and
        Q:row # I - J + R:row))).

queens_constraints(B, N) -->
  domain(B, 1, N) and safe(B).

heuristic h(N) =
  disjunctive(least(abs(N/2 - Val))
  for queens_labeling_val(Var, Val)).

? let(N := 4, B := board(N),
    queens_constraints(B, N) and
    queens_labeling(B, N)).
```

```
domain([Q_1_1,Q_2_1,Q_3_1,Q_4_1], 1, 4) and

all_different([Q_1_1,Q_2_1,Q_3_1,Q_4_1]) and
Q_1_1 # 1+Q_2_1 and Q_1_1 # -1+Q_2_1 and
Q_1_1 # 2+Q_3_1 and Q_1_1 # -2+Q_3_1 and
Q_1_1 # 3+Q_4_1 and Q_1_1 # -3+Q_4_1 and
Q_2_1 # 1+Q_3_1 and Q_2_1 # -1+Q_3_1 and
Q_2_1 # 2+Q_4_1 and Q_2_1 # -2+Q_4_1 and
Q_3_1 # 1+Q_4_1 and Q_3_1 # -1+Q_4_1 and

search(Q_1_1 = 2 or Q_1_1 = 3 or
       Q_1_1 = 1 or Q_1_1 = 4 and
       Q_2_1 = 2 or Q_2_1 = 3 or
       Q_2_1 = 1 or Q_2_1 = 4 and
       Q_3_1 = 2 or Q_3_1 = 3 or
       Q_3_1 = 1 or Q_3_1 = 4 and
       Q_4_1 = 2 or Q_4_1 = 3 or
       Q_4_1 = 1 or Q_4_1 = 4)
```

La compilation par $\longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow$ *de l'instance des* 4*-reines produit théoriquement un code quadratique en la taille de l'instance par la double imbrication de quantificateurs universels. Cependant, la taille du code effectivement produit est inférieure au produit cartésien selon la liste des* 4 *reines avec elle-même grâce à l'évaluation partielle effectuée sur l'implication, la partie gauche de l'implication étant complètement instanciée à la compilation.*

Correction des transformations vis-à-vis de la sémantique déclarative des modèles `Cream`

La proposition suivante montre que la compilation par $\longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow$ préserve la sémantique déclarative des modèles `Cream`.

Proposition 5. *Étant donné un modèle* `Cream` $\mathcal{M}$*, soit* $\mathcal{M}'$ *tel que* $\mathcal{M} \longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow \mathcal{M}'$ *le résultat de la compilation de* $\mathcal{M}$*, alors* $\mathcal{S}_{\text{s}}(\mathcal{M}) = \mathcal{S}_{\text{s}}(\mathcal{M}')$

Preuve 4. *Pour toute affectation* (ν^G, ν^O) *pour* $\mathcal{M}$*, nous vérifions inductivement sur les dérivations de* $\longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow$ *et* $\longrightarrow\!\!\langle \begin{smallmatrix}\text{stc}\\ \text{srch}\end{smallmatrix} \rangle\!\!\longrightarrow$ *que* $\nu^G(\mathcal{M}) \stackrel{*}{\rightarrow} \nu^G(\mathcal{M}')$*.*

La plupart des dérivations de $\longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow$ *sont indépendantes de toute affectation et vérifient donc la propriété par définition et correspondent à une réduction de la sémantique. En ce qui concerne les dérivations de* $\longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow$ *liées aux appels de définitions, elles sont restreintes aux appels dont les arguments sont indexables et utilisent* ν^O *pour l'indexation. De la même façon, les buts, qui sont des déclarations d'arité nulle, utilisent* ν^G *pour l'indexation des variables libres.*

La transformation qui formalise le mode recherche $\longrightarrow\!\!\left\langle \begin{smallmatrix}\text{stc}\\ \text{srch}\end{smallmatrix}\right\rangle\!\!\longrightarrow$ *se fonde sur* $\longrightarrow\!\!\langle \text{stc} \rangle\!\!\longrightarrow$ *et n'altère pas l'ensemble des solutions par ailleurs.*

6.2 Compilation vers CLP par génération de code procédural

Le schéma de compilation associé au *mode dynamique*, dit par génération de code procédural [MMF09], se distingue du schéma statique par la définition de deux transformations qui produisent du code intermédiaire structuré : $\multimap\langle\text{dyn}\rangle\longrightarrow$ produit du code déterministe qui pose les appels (aux déclarations) de contraintes et les appels aux parties recherche. Ces sous-expressions en *mode recherche dynamique* sont transformées par $\multimap\langle{}^{\text{dyn}}_{\text{srch}}\rangle\longrightarrow$ en code non déterministe qui s'occupe de la génération du code de réordonnancement et du parcours de l'arbre de recherche.

Ce schéma dynamique est nécessaire lorsque la forme du modèle dépend d'indexicaux, c'est-à-dire d'expressions qui contiennent des variables à domaine fini. Par exemple, supposons qu'un itérateur dépende d'une liste construite par intervalle dont une des bornes est un indexical. Alors, le résultat de l'application de l'itérateur est inconnu au moment de la compilation et le schéma par expansion ne peut s'appliquer.

Le code intermédiaire suit la syntaxe des modèles `Cream`, mais sans l'interdiction de récursion qui est levée, et avec des directives sur les stratégies de branchement (reformulés par $\multimap\langle{}^{\text{dyn}}_{\text{srch}}\rangle\longrightarrow$). Un tel code intermédiaire n'introduit pas de difficulté de traduction dans le langage cible à la génération de code.

6.2.1 Transformation en code déterministe

$\boldsymbol{V} \vdash \cdot \multimap\langle\text{dyn}\rangle\longrightarrow \cdot$ transforme inductivement sur la structure des expressions `Cream`. $\boldsymbol{V}$ représente l'ensemble des variables de l'environnement de la sous-expression considérée ; $\boldsymbol{V}$ est utilisé pour passer le contexte aux définitions auxiliaires introduites par la transformation.

Chaque déclaration d'objet $p\texttt{(}\boldsymbol{X}\texttt{) = }e \in O(\mathcal{M})$ et chaque déclaration de règle $p\texttt{(}\boldsymbol{X}\texttt{) --> }e \in R(\mathcal{M})$ est transformée en le code intermédiaire $p_d\texttt{(}\boldsymbol{X}\texttt{) = }e'$ où $\text{fv}(e) \vdash e \multimap\langle\text{dyn}\rangle\longrightarrow e'$. Les appels reposent alors sur la définition : $\boldsymbol{V} \vdash p(\boldsymbol{X}) \multimap\langle\text{dyn}\rangle\longrightarrow p_d\texttt{(}\boldsymbol{X}\texttt{)}$.

Dans le langage source, la récursion est interdite et les combinateurs sur listes finies sont préférés. Cependant, ces itérateurs sont traduits dans les langages cibles par des définitions récursives travaillant sur les transformations des listes. Pour chaque `foldl`, une définition récursive itérant sur une liste est générée.

$$\frac{\boldsymbol{V} \vdash l \multimap\langle\text{dyn}\rangle\longrightarrow l' \qquad \boldsymbol{V} \vdash i \multimap\langle\text{dyn}\rangle\longrightarrow i'}{\boldsymbol{V} \vdash \texttt{foldl(}A\ \texttt{from}\ i\texttt{,}\ X\ \texttt{in}\ l\texttt{,}\ e\texttt{)} \multimap\langle\text{dyn}\rangle\longrightarrow q\texttt{(}l'\texttt{,}\ i'\texttt{,}\ \boldsymbol{V}\texttt{)}}$$

où q est un nouveau symbole de prédicat défini comme suit, et toutes les variables sont fraîches vis-à-vis de $\boldsymbol{V}$:

$$q\texttt{([],}\ I\texttt{,}\ \boldsymbol{V}\texttt{) = }I\texttt{.}$$
$$q\texttt{([}H\ \texttt{|}\ T\texttt{],}\ I\texttt{,}\ \boldsymbol{V}\texttt{) = }q\texttt{(}T\texttt{,}\ e'\texttt{,}\ \boldsymbol{V}\texttt{).} \qquad \boldsymbol{V} \vdash e[A := I, X := H] \multimap\langle\text{dyn}\rangle\longrightarrow e'$$

Les autres cas de $\longrightarrow\!\langle \text{dyn} \rangle\!\longrightarrow$ sont définis homomorphiquement aux sous-expressions, en considérant les questions de portée et de captures de noms : par exemple,

$$\frac{\boldsymbol{V} \vdash d \longrightarrow\!\langle \text{dyn} \rangle\!\longrightarrow d' \qquad \boldsymbol{V} \cdot X \vdash e[V := X] \longrightarrow\!\langle \text{dyn} \rangle\!\longrightarrow e'}{\boldsymbol{V} \vdash \texttt{let(}V \texttt{ := } d \texttt{ in } e\texttt{)} \longrightarrow\!\langle \text{dyn} \rangle\!\longrightarrow \texttt{let(}X \texttt{ := } d' \texttt{ in } e'\texttt{)}}$$

où X est une variable fraîche.

Les directives de recherche reposent sur la transformation de recherche (définie dans la section 6.2.2).

$$\frac{\begin{array}{c}\boldsymbol{V} \vdash h \longrightarrow\!\langle \text{dyn} \rangle\!\longrightarrow \texttt{ conjunctive(}o_\wedge^1\texttt{) ... and conjunctive(}o_\wedge^n\texttt{) and} \\ \texttt{disjunctive(}o_\vee^1\texttt{) ... and disjunctive(}o_\vee^m\texttt{)} \\ ([o_\wedge^1, \ldots, o_\wedge^n], [o_\vee^1, \ldots, o_\vee^m]); \boldsymbol{V} \vdash e \longrightarrow\!\langle {}^{\text{dyn}}_{\text{srch}} \rangle\!\longrightarrow e'\end{array}}{\boldsymbol{V} \vdash \texttt{search(}h\texttt{, } e\texttt{)} \longrightarrow\!\langle \text{dyn} \rangle\!\longrightarrow e'}$$

6.2.2 Transformation en code non déterministe

Dans le contexte dynamique, ni la stratégie de branchement, ni les critères des heuristiques ne peuvent être supposés connus précisément au moment de la compilation. La transformation $\longrightarrow\!\langle {}^{\text{dyn}}_{\text{srch}} \rangle\!\longrightarrow$ génère du code qui réordonne au moment de l'exécution plutôt que de réordonner statiquement au moment de la compilation et les opérateurs de disjonction `or` deviennent des points de choix.

La compilation des heuristiques de recherche repose sur la notion de *couche* O : pour $O \in \{\wedge, \vee\}$, on appelle *couche* O d'un arbre $\wedge/\vee$ un sous-graphe maximal de l'arbre qui ne contient que noeuds O. Le symbole $\wedge$ correspond à la conjonction `and` et le symbole $\vee$ à la disjonction `or`. La définition des couches O est généralisée aux expressions de la syntaxe `Cream` en leur rendant transparents les `let`, les appels de déclarations, la partie droite de `implies` et les arbres décrits en intention par `foldl`. Les noeuds enfants d'une couche sont les noeuds hors de la couche et enfants d'un noeud dans la couche. La *couche* O *racine* est la couche O qui contient le noeud racine s'il n'est pas le dual de O, ou la couche vide sinon. Par convention, le noeud racine est le seul enfant de la couche vide.

Le réordonnancement défini par les heuristiques est appliqué entre tous les noeuds enfants de chaque couche O : les critères définis pour un opérateur O conjonctif ou disjonctif associent un vecteur de coûts à chaque enfant, et les enfants sont réordonnés selon leurs coûts, lexicographiquement.

La compilation par $\longrightarrow\!\langle {}^{\text{dyn}}_{\text{srch}} \rangle\!\longrightarrow$ produit, pour un couple donné de critères $(\boldsymbol{o}_\wedge, \boldsymbol{o}_\vee)$, du code qui réordonne la racine d'une couche O de l'arbre et récursivement pour les enfants. Deux vecteurs de coût $\boldsymbol{c}_\wedge$ and $\boldsymbol{c}_\vee$, de même dimension que $\boldsymbol{o}_\wedge$ et $\boldsymbol{o}_\vee$ respectivement, sont entretenus.

$$(\boldsymbol{o}_\wedge, \boldsymbol{o}_\vee); \boldsymbol{V} \vdash \cdot \longrightarrow\!\langle {}^{\text{dyn}}_{\text{srch}} \rangle\!\longrightarrow \cdot \quad \equiv \quad (\boldsymbol{c}_\wedge^\infty, \boldsymbol{c}_\vee^\infty); \boldsymbol{V} \vdash \cdot \longrightarrow\!\langle {}^{\text{dyn}}_{\text{srch}(\wedge)} \rangle\!\longrightarrow \cdot$$

La transformation est initialisée arbitrairement avec des vecteurs de coûts maxima $\boldsymbol{c}_\wedge^\infty$ et $\boldsymbol{c}_\vee^\infty$, dont chaque composante est égale à `max_integer`, puisqu'aucun critère ne s'applique en dehors de la formule induisant l'arbre de recherche. La couche racine, potentiellement vide, peut toujours être considérée comme une couche $\wedge$.

$\cdot \multimap\!\langle {}^{\text{dyn}}_{\text{srch}(O)} \rangle\!\longrightarrow \cdot$ repose sur la transformation auxiliaire $(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash \cdot \multimap\!\langle {}^{\text{dyn}}_{\text{list}(O)} \rangle\!\longrightarrow \cdot$ qui produit du code qui génère une liste d'associations : pour chaque noeud fils d'une couche O, le vecteur de coûts du noeud est associé à l'appel de définition pour explorer le fils récursivement.

$$\frac{(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash e \multimap\!\langle {}^{\text{dyn}}_{\text{list}(O)} \rangle\!\longrightarrow e'}{(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash e \multimap\!\langle {}^{\text{dyn}}_{\text{srch}(O)} \rangle\!\longrightarrow \texttt{iter\_predicates}_O\texttt{(}e'\texttt{)}}$$

où $\texttt{iter\_predicates}_O\texttt{(}l\texttt{)}$ est une fonction interne qui sélectionne itérativement l'élément de meilleur coût dans la liste l, exécute la définition associée, et considère ensuite les autres éléments récursivement, selon O, soit en conjonction soit en disjonction.

Déclarations et appels

Pour chaque déclaration $p\texttt{(}\boldsymbol{X}\texttt{)}\ \texttt{-->}\ e\texttt{.} \in R(\mathcal{M})$, la compilation produit deux définitions en code intermédiaire, une pour chaque type de couche, *i.e.* conjonctive et disjonctive.

DÉCLARATION POUR COUCHE $\wedge$

$$\frac{(u(\boldsymbol{C}_\wedge, \boldsymbol{o}_\wedge, p\texttt{(}\boldsymbol{X}\texttt{)}), \boldsymbol{C}_\vee); \mathrm{fv}(e) \vdash e \multimap\!\langle {}^{\text{dyn}}_{\text{srch}(\wedge)} \rangle\!\longrightarrow e'}{(\boldsymbol{o}_\wedge, \boldsymbol{o}_\vee); \boldsymbol{V} \vdash p\texttt{(}\boldsymbol{X}\texttt{)}\ \texttt{-->}\ e \multimap\!\langle {}^{\text{dyn}}_{\text{srch}(\wedge)} \rangle\!\longrightarrow p_\wedge(\boldsymbol{C}_\wedge, \boldsymbol{C}_\vee, \boldsymbol{X})\ \texttt{=}\ e'}\ p\texttt{(}\boldsymbol{X}\texttt{)}\ \texttt{-->}\ e\texttt{.} \in R(\mathcal{M})$$

DÉCLARATION POUR COUCHE $\vee$

$$\frac{(\boldsymbol{C}_\wedge, u(\boldsymbol{C}_\vee, \boldsymbol{o}_\vee, p\texttt{(}\boldsymbol{X}\texttt{)})); \mathrm{fv}(e) \vdash e \multimap\!\langle {}^{\text{dyn}}_{\text{srch}(\vee)} \rangle\!\longrightarrow e'}{(\boldsymbol{o}_\wedge, \boldsymbol{o}_\vee); \boldsymbol{V} \vdash p\texttt{(}\boldsymbol{X}\texttt{)}\ \texttt{-->}\ e \multimap\!\langle {}^{\text{dyn}}_{\text{srch}(\vee)} \rangle\!\longrightarrow p_\vee(\boldsymbol{C}_\wedge, \boldsymbol{C}_\vee, \boldsymbol{X})\ \texttt{=}\ e'}\ p\texttt{(}\boldsymbol{X}\texttt{)}\ \texttt{-->}\ e\texttt{.} \in R(\mathcal{M})$$

où les vecteurs de variables $\boldsymbol{C}_\wedge = v(\dim(\boldsymbol{o}_\wedge))$ et $\boldsymbol{C}_\vee = v(\dim(\boldsymbol{o}_\vee))$, avec $v(n) = [V_1, \ldots, V_n]$, permettent de passer les coûts d'une déclaration du code intermédiaire à une sous-déclaration. La fonction $u(\boldsymbol{c}, \boldsymbol{o}, p\texttt{(}\boldsymbol{X}\texttt{)})$ calcule le le nouveau vecteur de coûts $\boldsymbol{c}'$, en mettant à jour les composantes qui concernent le prédicat $p\texttt{(}\boldsymbol{X}\texttt{)}$:

$$u(\overrightarrow{c_i}, \overrightarrow{e_i\ \texttt{for}\ p_i\texttt{(}\boldsymbol{Y}_i\texttt{)}}, p\texttt{(}\boldsymbol{X}\texttt{)}) = \overrightarrow{c_i'}$$

où :

$$c_i' = \begin{cases} \sigma(e_i) & \text{si } \sigma(p_i\texttt{(}\boldsymbol{Y}_i\texttt{)}) = p\texttt{(}\boldsymbol{X}\texttt{)} \\ c_i & \text{sinon} \end{cases}$$

Les appels au prédicat $p\texttt{(}\boldsymbol{X}\texttt{)}$ se font sur une de ces deux définitions, selon le type de la couche courante O.

$$\frac{}{(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash p\texttt{(}\boldsymbol{X}\texttt{)} \multimap\!\!\left\langle {}^{\text{dyn}}_{\text{list}(O)} \right\rangle\!\!\longrightarrow p_O\texttt{(}\boldsymbol{c}_\wedge\texttt{, }\boldsymbol{c}_\vee\texttt{, }\boldsymbol{X}\texttt{)}} \; p\texttt{(}\boldsymbol{X}\texttt{) --> }e\texttt{.} \in R(\mathcal{M})$$

Conjonction et disjonction

Les transformations de conjonctions et disjonctions $\multimap\!\!\left\langle {}^{\text{dyn}}_{\text{list}(\wedge)} \right\rangle\!\!\longrightarrow$ et $\multimap\!\!\left\langle {}^{\text{dyn}}_{\text{list}(\vee)} \right\rangle\!\!\longrightarrow$ sont définies de manière duale. Considérons ici uniquement le cas de les transformations de conjonctions.

$\multimap\!\!\left\langle {}^{\text{dyn}}_{\text{list}(\wedge)} \right\rangle\!\!\longrightarrow$ produit du code qui concatène les listes produites par les transformations des opérandes de la conjonction a `and` b.

$$\frac{(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash a \multimap\!\!\left\langle {}^{\text{dyn}}_{\text{list}(\wedge)} \right\rangle\!\!\longrightarrow a' \qquad (\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash b \multimap\!\!\left\langle {}^{\text{dyn}}_{\text{list}(\wedge)} \right\rangle\!\!\longrightarrow b'}{(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash a \texttt{ and } b \multimap\!\!\left\langle {}^{\text{dyn}}_{\text{list}(\wedge)} \right\rangle\!\!\longrightarrow \texttt{append(}a'\texttt{, }b'\texttt{)}}$$

Lors d'un changement d'opérateur au sein d'une couche, en l'occurence une disjonction a `or` b dans une couche $\wedge$, un changement de couche s'impose. La disjonction est un nouvel élément à ordonner pour la couche courante, un noeud enfant de la couche $\wedge$, et le prédicat q est introduit pour se charger de la construction de la sous couche $\vee$. La construction syntaxique `delay(`$p\texttt{(}\boldsymbol{X}\texttt{)}$`)` est introduite pour dénoter symboliquement le terme $p\texttt{(}\boldsymbol{X}\texttt{)}$ plutôt que l'application de la déclaration $p\texttt{(}\boldsymbol{X}\texttt{)}$.

$$(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash a \texttt{ or } b \multimap\!\!\left\langle {}^{\text{dyn}}_{\text{list}(\wedge)} \right\rangle\!\!\longrightarrow \texttt{[\{ costs = }\boldsymbol{c}_\wedge\texttt{,}$$
$$\texttt{predicate = delay(}q\texttt{(}\boldsymbol{c}_\wedge\texttt{, }\boldsymbol{c}_\vee\texttt{, }\boldsymbol{V}\texttt{)) \}]}$$

où q applique la transformation récursivement à la sous-couche $\vee$ (toutes les variables sont fraîches vis-à-vis de $\boldsymbol{V}$) :

$$q\texttt{(}\boldsymbol{C}_\wedge\texttt{, }\boldsymbol{C}_\vee\texttt{, }\boldsymbol{V}\texttt{) = }e\texttt{.} \qquad (\boldsymbol{C}_\wedge, \boldsymbol{C}_\vee); \boldsymbol{V} \vdash a \texttt{ or } b \multimap\!\!\left\langle {}^{\text{dyn}}_{\text{srch}(\vee)} \right\rangle\!\!\longrightarrow e$$

Implication

Le sous-arbre de la partie gauche b d'une implication a `implies` b est exécuté si la partie droite a est vraie.

$$\frac{\boldsymbol{V} \vdash a \multimap\!\langle \text{dyn} \rangle\!\longrightarrow a' \qquad (\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash b \multimap\!\!\left\langle {}^{\text{dyn}}_{\text{list}(O)} \right\rangle\!\!\longrightarrow b'}{(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash a \texttt{ implies } b \multimap\!\!\left\langle {}^{\text{dyn}}_{\text{list}(O)} \right\rangle\!\!\longrightarrow \texttt{filter(}\boldsymbol{c}_O\texttt{, }a'\texttt{, }b'\texttt{)}}$$

où

$$\texttt{filter(}\boldsymbol{c}\texttt{, }g\texttt{, }d\texttt{)} = \begin{cases} d & \text{si } g \text{ est vraie} \\ \texttt{[\{ costs = }\boldsymbol{c}\texttt{,} & \text{sinon} \\ \quad \texttt{predicate = delay(true) \}]} & \end{cases}$$

Contraintes et sous directives de recherche

Une contrainte ou une sous directive de recherche e est un noeud enfant d'une couche. La transformation produit donc pour chacune une liste singleton associant leur coût avec un nouveau prédicat q.

$$(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash e \longrightarrow\!\!\left\langle {}^{\text{dyn}}_{\text{list}(O)} \right\rangle\!\!\longrightarrow \texttt{[\{ costs = } \boldsymbol{c}_O \texttt{,}\ \texttt{predicate = delay(}q\texttt{(}\boldsymbol{V}\texttt{)) \}]}$$

où q applique récursivement la transformation (toutes les variables sont libres vis-à-vis de $\boldsymbol{V}$) :

$$q\texttt{(}\boldsymbol{V}\texttt{) = } e'\texttt{.} \qquad\qquad \boldsymbol{V} \vdash e \longrightarrow\!\!\langle \text{dyn} \rangle\!\!\longrightarrow e'$$

Liaison `let`

$$\frac{\boldsymbol{V} \vdash v \longrightarrow\!\!\langle \text{dyn} \rangle\!\!\longrightarrow v' \qquad (\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \cdot Y \vdash e[X := Y] \longrightarrow\!\!\left\langle {}^{\text{dyn}}_{\text{list}(O)} \right\rangle\!\!\longrightarrow e'}{(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash \texttt{let(}X\ \texttt{:=}\ v\ \texttt{in}\ e\texttt{)} \longrightarrow\!\!\left\langle {}^{\text{dyn}}_{\text{list}(O)} \right\rangle\!\!\longrightarrow \texttt{let(}Y\ \texttt{:=}\ v'\texttt{,}\ e'\texttt{)}}$$

où Y est une variable fraîche.

Combinateurs

Les combinateurs sont traduits par des définitions récursives. De plus, il est possible que l'appel récursif se fasse dans une autre couche que celle de l'appel du combinateur. C'est pourquoi la compilation des combinateurs fait usage d'un symbole spécial `rec` pour appréhender la récursion de manière compatible avec les changements de couche.

$$\frac{\boldsymbol{V} \vdash \texttt{reverse(}l\texttt{)} \longrightarrow\!\!\langle \text{dyn} \rangle\!\!\longrightarrow l'}{(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \vdash \texttt{foldl(}A\ \texttt{from}\ i\texttt{,}\ X\ \texttt{in}\ l\texttt{,}\ e\texttt{)} \longrightarrow\!\!\left\langle {}^{\text{dyn}}_{\text{list}(O)} \right\rangle\!\!\longrightarrow q_O\texttt{(}l'\texttt{,}\ \boldsymbol{c}_\wedge\texttt{,}\ \boldsymbol{c}_\vee\texttt{,}\ \boldsymbol{V}\texttt{)}}$$

où q_O est un nouveau symbole de prédicat défini comme suit (toutes les variables sont fraîches vis-à-vis de $\boldsymbol{V}$) :

$$\begin{array}{rl} q_O\texttt{([],}\boldsymbol{C}_\wedge\texttt{,}\boldsymbol{C}_\vee\texttt{,}\boldsymbol{V}\texttt{) = } i'\texttt{.} & (\boldsymbol{C}_\wedge, \boldsymbol{C}_\vee); \boldsymbol{V} \vdash i \longrightarrow\!\!\left\langle {}^{\text{dyn}}_{\text{list}(O)} \right\rangle\!\!\longrightarrow i' \\ q_O\texttt{([}H\ \texttt{|}\ T\texttt{],}\boldsymbol{C}_\wedge\texttt{,}\boldsymbol{C}_\vee\texttt{,}\boldsymbol{V}\texttt{) = } e'\texttt{.} & (\boldsymbol{C}_\wedge, \boldsymbol{C}_\vee); \boldsymbol{V} \cdot H \vdash \\ & e[A := \texttt{rec(}q\texttt{,}\ T\texttt{,}\ \boldsymbol{V}\texttt{)}, X := H] \longrightarrow\!\!\left\langle {}^{\text{dyn}}_{\text{list}(O)} \right\rangle\!\!\longrightarrow e' \end{array}$$

et `rec` est traduit en l'appel récursif à q_O :

$$(\boldsymbol{c}_\wedge, \boldsymbol{c}_\vee); \boldsymbol{V} \cdot H \vdash \texttt{rec(}q\texttt{,}\ T\texttt{,}\ \boldsymbol{V}\texttt{)} \longrightarrow\!\!\left\langle {}^{\text{dyn}}_{\text{list}(O)} \right\rangle\!\!\longrightarrow q_O\texttt{(}T\texttt{,}\ \boldsymbol{c}_\wedge\texttt{,}\ \boldsymbol{c}_\vee\texttt{,}\ \boldsymbol{V}\texttt{)}$$

6.2.3 Complexité

Propriété 1. *Pour tout modèle* **Cream** $\mathcal{M}$*, la taille du programme généré par* $\longrightarrow\!\langle \text{dyn} \rangle\!\longrightarrow$ *est linéaire en la taille de* $\mathcal{M}$*. Il existe* $\mathbf{O}(|D(m)| \cdot s)$ *définitions du code intermédiaire (*p_d*,* $p_\vee$ *et* $p_\wedge$*), où* $|D(m)|$ *est le nombre de définitions de* $\mathcal{M}$*, et* s *est le nombre de prédicats* `search`*. Chaque définition du code intermédiaire, et donc du code généré, est de taille linéaire en la taille de la déclaration* **Cream** *originelle ; définitions auxiliaires dédiées aux* `foldl` *et aux sous-couches comprises.*

Preuve 5. *Les transformations* $\longrightarrow\!\langle \text{dyn} \rangle\!\longrightarrow$ *et* $\longrightarrow\!\langle {}^{\text{dyn}}_{\text{list}(\cdot)} \rangle\!\longrightarrow$ *sont des transformations inductives pour lesquelles chaque étape compose linéairement le résultat des sous transformations, soit par définitions auxiliaires, soit par expressions « en place ». Par conséquent, il existe un facteur constant entre la taille des définitions générées et la taille d'une déclaration* **Cream** *originelle. Plus précisément, pour chaque déclaration* **Cream** $p(\boldsymbol{X})$*, sont générées une définition* p_d*, et deux définitions* $p_\vee$ *et* $p_\wedge$ *par sous-directive de recherche* `search`*.* □

Ce résultat de complexité est à mettre en rapport avec la transformation du mode statique $\longrightarrow\!\langle \text{stc} \rangle\!\longrightarrow$ dans laquelle le développement complet des déclarations mène à une taille du code généré exponentielle en la taille du modèle dans le pire des cas.

Exemple 18. *Considérons la transformation de la version dynamique du modèle* **Cream** *du problème des n-reines. Pour une description des heuristiques du modèle employées, voir la section 7.1. Par la suite, le modèle est représenté à gauche et sa transformation en code intermédiaire par* $\longrightarrow\!\langle \text{dyn} \rangle\!\longrightarrow$ *à droite.*

But et déclarations d'objets

Le but du modèle est compilée par $\longrightarrow\!\langle \text{dyn} \rangle\!\longrightarrow$ *en le code intermédiaire suivant où chaque appel de déclaration du but du modèle est associé un appel de définition du code intermédiaire :*

```
? dynamic(let(N := 4, B := board(N),
    queens_constraints(B, N) and
    queens_labeling(variables(B)))).
```

```
goal = let(N := 4, B := board(N),
  queens_constraints(B, N) and
  queens_labeling(rcp_variables(B))).
```

La déclaration `q/1` *est transformée en une déclaration du code intermédiaire dans laquelle la variable libre porte l'information de sa provenance (premier argument de* `rcp_var/2`*) et sa position syntaxique dans le corps de la déclaration (deuxième argument de* `rcp_var/2`*).*

```
q(I) = {row:_, column:I}.
```

```
q(I) = rcp_rec([row(rcp_var(q(I),1)), column(I)]).
```

La déclaration de l'échiquier `board/1` *est transformée en une définition principale et une définition récursive auxiliaire* `board_foldl_1/3` *pour le* `map` *:*

```
board(N) =
  map(I in [1 .. N],
      q(I)).
```

```
board_foldl_1([], I_board_foldl_1, _) = I_board_foldl_1.
board_foldl_1([I_1 | Tail_1], I_board_foldl_1, []) =
  board_foldl_1(Tail_1, append(I_board_foldl_1, q(I_1)), []).

board(N) = board_foldl_1(rcp_range(1, N), [], []).
```

Déclarations de règles relatives aux contraintes

La contrainte `all_different/1` *est une contrainte globale intégrée à* ***Cream*** *et les contraintes relatives aux diagonales sont posées par la traduction de l'imbrication des quantifications universelles* `safe_foldl_2/3`.

```
safe(L) -->
  all_different(L) and
  forall(Q in L,
    forall(R in L,
      let(I := Q:column,
          J := R:column,
        I < J implies
        Q:row # J - I + R:row and
        Q:row # I - J + R:row))).

queens_constraints(B, N) -->
  domain(B, 1, N) and safe(B).
```

```
safe(L) =
  all_different(rcp_variables(L)) and
  safe_foldl_2(L, 1, []).


queens_constraints(B, N) =
  domain(rcp_variables(B), 1, N) and
  safe(B).
```

La transformation du quantificateur le plus extérieur (dont la variable liée est `Q`*) donne la définition récursive* `safe_foldl_2/3` *qui itère sur les éléments de la liste représentant l'échiquier. À chaque itération,* `safe_foldl_2/3` *combine par conjonction (avec comme élément initial* `1`*) les contraintes posées par la définition* `safe_foldl_3/3` *:*

```
safe_foldl_2([], I_safe_foldl_2, _) = I_safe_foldl_2.
safe_foldl_2([Q_2 | Tail_2], I_safe_foldl_2, []) =
  safe_foldl_2(Tail_2, (I_safe_foldl_2 and safe_foldl_3(L, 1, Q_2)), []).
```

La définition récursive `safe_foldl_3/3` *est la traduction du quantificateur universel le plus profond et itère aussi sur la liste représentant l'échiquier. Elle pose effectivement les contraintes de non-capture entre une reine particulière, représentée par la variable* `Q_2` *de l'environnement, et l'ensemble des reines contenues dans la liste dont l'élément courant est la variable* `R_3` *.*

```
safe_foldl_3([], I_safe_foldl_3, _) = I_safe_foldl_3.
safe_foldl_3([R_3 | Tail_3], I_safe_foldl_3, [Q_2]) =
  safe_foldl_3(Tail_3,
               (I_safe_foldl_3 and
                let(I := rcp_att(Q_2, column),
                    J := rcp_att(R_3, column),
                  I < J implies
                  rcp_att(Q_2, row) # J - I + rcp_att(R_3, row) and
                  rcp_att(Q_2, row) # I - J + rcp_att(R_3, row))),
                  [Q_2]).
```

Déclarations des heuristiques

Dans le modèle **Cream**, *une heuristique conjonctive (*`ff`*) et une heuristique disjonctive (*`mo`*) paramètrent le prédicat* `search` *comme présenté ci-dessous.*

```
heuristic ff =
  conjunctive(least(domain_size(Var))
              for queens_labeling_var(Var)).
heuristic mo =
  disjunctive(least(abs((domain_max(Var) - domain_min(Var))/2 - Val))
              for queens_labeling_val(Var, Val)).

queens_labeling(Vars) -->
  search(ff and mo, forall(Var in Vars, queens_labeling_var(Var))).
```

Quantification universelle et filtrage de la règle `queens_labeling_var/1`

La compilation du quantificateur universel `forall` *par* $\longrightarrow\!\!\!\!\!\!\left\langle \begin{smallmatrix}\text{dyn}\\ \text{list}(\wedge)\end{smallmatrix} \right\rangle\!\!\!\!\!\!\longrightarrow$ *produit un appel à la procédure récursive* `srch_foldl_1_and/1`*. Cette procédure est considérée par convention dans une couche* $\wedge$ *et les coûts correspondant aux critères des heuristiques sont initialisés à* `max_integer` *(le plus grand coût possible).*

```
queens_labeling(Vars) =
  iter_predicates_and(srch_foldl_1_and(Vars, [max_integer], [max_integer], [])).
```

`srch_foldl_1_and/1` *génère une liste de couples (*`costs`, `predicate`*) à partir de laquelle la procédure* `iter_predicates_and/1` *posera itérativement le but associé au couple de plus petit coût. À noter que la procédure transporte les vecteurs de coûts relatifs aux heuristiques conjonctives et disjonctives par les variables* `CostsC` *et* `CostsD`*, respectivement.*

```
srch_foldl_1_and([], CostsC, _, _) =
  [rcp_rec([costs(CostsC), predicate(delay(true))])].
srch_foldl_1_and([Var | Tail], CostsC, CostsD, []) =
  append(queens_labeling_var_and(CostsC, CostsD, [Var]),
         srch_foldl_1_and(Tail, CostsC, CostsD, [])).
```

Chaque élément de la liste produite est le résultat d'un appel à la définition du code intermédiaire `queens_labeling_var_and/3`*, générée par la compilation de la règle du modèle* `queens_labeling_var/1`*, qui définit un sous arbre des affectations possibles d'une variable du problème.*

```
queens_labeling_var(Var) -->
  exists(Val in [domain_min(Var) .. domain_max(Var)],
         queens_labeling_val(Var, Val)).
```

La règle `queens_labeling_var/1` *est membre d'une couche* $\wedge$ *et filtrée par l'heuristique* `ff`*. Le coût relatif à l'heuristique conjonctive est donc mis à jour pour la définition du code généré et devient* `domain_size(Var)`*.*

```
queens_labeling_var_and(_, CostsD, [Var]) =
  srch_foldl_2_and(rcp_range([domain_min(Var)..domain_max(Var)]),
                   domain_size(Var), CostsD, [Var]).
```

Quantification existentielle et filtrage de la règle `queens_labeling_val/2`

La compilation du quantificateur existentiel **`exists`** *par* $\longrightarrow\!\!\langle {}^{\text{dyn}}_{\text{list}(\wedge)} \rangle\!\!\longrightarrow$ *produit un appel à la procédure récursive* `srch_foldl_2_and/3` *dans laquelle s'opère le changement de couche* $\wedge$ *en couche* $\vee$*.*

```
srch_foldl_2_and([], CostsC, CostsD, _) =
  [rcp_rec([costs(CostsC), predicate(delay(true))])].
srch_foldl_2_and([Val | Tail], CostsC, CostsD, [Var]) =
  [rcp_rec([costs(CostsC),
            predicate(delay(srch_subterm_1(Tail,CostsC,CostsD,[Var,Val])))])].
```

Le changement de couche est matérialisé par l'appel à la définition `srch_subterm_1/4`*.*

```
srch_subterm_1(Tail, CostsC, CostsD, [Var, Val]) =
  iter_predicates_or(append(queens_labeling_val_or(CostsC, CostsD, [Var, Val]),
                            srch_foldl_2_or(Tail, CostsC, CostsD, Var))).
```

Cette définition fait elle-même appel à la procédure récursive `srch_foldl_2_or` *qui génère la liste des appels affectations possibles et dont chaque élément est le produit d'un appel à* `queens_labeling_val_or/3`*. Dans le cas présent, la procédure* **`iter_predicates_or/2`** *appliquera récursivement le choix non déterministe du prédicat de ladite liste associé au plus petit coût.*

```
srch_foldl_2_or([], _, CostsD, _) =
  [rcp_rec([costs(CostsD), predicate(delay(true))])].
srch_foldl_2_or([Var | Tail], Costs_I, Costs_J, [Var, Val]) =
  append(queens_labeling_var_or(CostsC, CostsD, [Var, Val]),
         srch_foldl_1_or(Tail, CostsC, CostsD, [Var, Val])).
```

Le prédicat `queens_labeling_val_or/3` *est le résultat de la compilation de* `queens_labeling_val/2` *par* $\longrightarrow\!\!\langle {}^{\text{dyn}}_{\text{list}(\vee)} \rangle\!\!\longrightarrow$ *et correspond à la sélection d'une valeur pour* `Var`*.*

```
queens_labeling_val(Var, Val) --> Var = Val.
```

`queens_labeling_val_or/3` *renvoit un couple dont le coût est celui relatif à l'heuristique disjonctive (*`abs(Var-Val)/2`*) et le prédicat associé est l'affectation elle-même (*`Var=Val`*).*

```
queens_labeling_val_or(_, _, [Var, Val]) =
  rcp_rec([costs([abs(Var - Val)/2]), predicate(delay(Var = Val))]).
```

Cet exemple a illustré le schéma de compilation par génération de code procédural et la linéarité de cette transformation. Pour chaque règle et quantificateur sont produites deux définitions du code intermédiaire et à chaque changement de couche est produite une définition du code intermédiaire.

Chapitre 7

Heuristiques de recherche déclaratives

Sommaire

Ajouter à un langage de modélisation les moyens d'exprimer des heuristiques de recherche est indispensable pour résoudre un problème combinatoire efficacement voire effectivement. La PPC offre un langage riche pour formuler les problèmes combinatoires, mais dans la plupart des cas concrets, une fois modélisé, le problème n'est pas résolu par une stratégie de recherche standard. Alors, sans les moyens de définir et expérimenter rapidement une stratégie, les ingénieurs non-experts du domaine ne vont pas plus loin avec la PPC.

Les langages de modélisation OPL et Comet sont les seuls de l'état de l'art à proposer une composante recherche. Dans ces langages, les stratégies de branchement sont exprimées par des imbrications d'itérateurs `forall` pour le séquencement et `tryall` pour le choix non déterministe de contraintes. Ils offrent en outre des structures de contrôle comme les conditionnelles ou les boucles et donc une composante de programmation de la recherche. Les heuristiques sont définies comme des ordres lexicographiques sur les listes qui paramètrent ces deux itérateurs.

Une originalité de `Cream` est de proposer une composante recherche purement déclarative. Les stratégies de branchement sont exprimés par des formules logiques quelconques dans la portée du prédicat `search`. Dans ces formules, la conjonction est interprétée comme l'opérateur de séquence et la disjonction comme l'opérateur de choix non déterministe. Les quantificateurs universels `forall` et existentiels `exists` jouent donc des rôles similaires aux itérateurs `forall` et `tryall` d'OPL et Comet. Il est possible en `Cream` de définir simplement des stratégies de branchement plus complexes que des entrelacements de quantifications universelles et existentielles avec les combinateurs génériques `fold`. En `Cream`, les heuristiques ne sont pas définies à chaque niveau d'itérateur sur les listes qui les paramètrent mais par filtrage ou *pattern-matching* sur les têtes de règles présentes dans les sous-formules conjonctives et disjonctives du modèle.

L'avantage de cette approche est de permettre au modélisateur de définir indépendemment et de manière déclarative d'une part la stratégie de branchement et d'autre part les heuristiques qui le réordonne à un haut niveau de discours. Les conséquences pratiques sont la rapidité de développement et de modification des stratégies de recherche et leur lisibilité même pour les non-programmeurs.

Dans un langage de programmation logique avec contraintes comme `SICStus-Prolog` [C+07], la notion d'heuristique de sélection de variables et de valeurs recouvre à la fois la notion d'ordonnancement et la notion de stratégie de branchement. L'ensemble des heuristiques standards au sens de `SICStus-Prolog` est représenté par les options du prédicat `labeling/2`. La table suivante montre pour chaque option comment s'exprime l'équivalent `Cream`.

Option de `labeling/2` SICStus-Prolog	Équivalent `Cream`
`min`	`conjunctive(least(domain_min(X)) for labeling_var(X)`
`max`	`conjunctive(greatest(domain_max(X)) for labeling_var(X)`
`ff`	cf. Sec. 7.2
`ffc`	`conjunctive(least(domain_size(X)) for labeling_var(X) and conjunctive(greatest(degree(X)) for labeling_var(X)`
`enum`	arbre d'affectations type Fig. 4.6, cf. Sec. 7.2
`bisect`	`dichotomic_split/1`, cf. Chap. 9
`step`	sur le modèle de `interval_split/1`, cf. Chap. 9
`up`	`disjunctive(least(V) for labeling_val(_, V)`
`down`	`disjunctive(greatest(V) for labeling_val(_, V)`

TABLE 7.1: Heuristiques de sélection de variables et de sélection de valeurs standards `SICStus-Prolog` et équivalents `Cream`.

Dans la table 7.1, on considère une définition `Cream` du *labeling* sur la base du modèle présenté dans la section 7.2. On suppose que le langage est enrichi de la fonction prédéfinie `degree/1` qui prend une variable à domaine fini `X` en argument et dénote le nombre de contraintes actives du modèle dans lesquelles `X` apparaît.

Dans ce chapitre, nous présentons des stratégies de recherches `Cream` pour trois problèmes. Un premier problème simple d'ordonnancement de tâches disjonctives illustre la capacité de `Cream` à spécifier des heuristiques de façon déclarative sur des formules logiques. Ensuite, le problème des n-reines montre que les heuristiques traditionnelles de sélection de variables et de valeurs sont respectivement des cas particuliers d'heuristiques conjonctives et disjonctives. Enfin nous renvoyons à un problème de placement issu de l'industrie pour montrer l'efficacité de l'approche sur des problèmes réels.

7.1 Heuristiques pour les formules conjonctives et disjonctives

Considérons le modèle `Cream` du problème du pont (*Bridge Problem* [Van99], p. 209) consistant à trouver un ordonnancement des tâches qui minimise la durée de construction

d'un pont à 5 segments. Le projet implique 46 tâches et un ensemble de contraintes sur ces tâches. En plus des contraintes habituelles de précédence, le problème fait intervenir des contraintes de ressources. La plupart des tâches requièrent une ressource (une grue par exemple) et les tâches qui demandent la même ressource ne peuvent se chevaucher dans le temps.

Les déclarations d'objets

Les tâches sont définies comme des couples (date de début, durée). La fonction `end/1` calcule la somme des composantes et dénote donc la date de fin d'une tâche.

```
m1 = {start : _, duration : 16}.
m2 = {start : _, duration : 8}.
(...)
end(Task) = Task:start + Task:duration.
```

La borne supérieure de la durée effective de construction du pont et le domaine des tâches sont définis.

```
(...)
maxDuration = sum(map(T in tasks, T:duration)).
tasks_domain --> domain(tasks, 0, maxDuration).
```

Le but

Le but du modèle `Cream` consiste à poser les contraintes de précédence, de distance et de disjonction et à demander la minimisation de la date de fin des travaux par une recherche *branch and bound* (cf. Sec. 4.1).

```
? tasks_domain and precedences and distances and disjunctives and
  minimize(disjunctives, stop:start).
```

Heuristiques et déclarations règles relatives à la recherche

L'arbre de recherche exploré à chaque itération de l'optimisation est induit par la formule `disjunctives`, dans la portée du prédicat de `minimize`, qui met en disjonction les tâches qui requièrent la même ressource. La formule `disjunctives` est définie par les règles suivantes :

```
precedes(T1, T2) --> end(T1) =< T2:start.
disjuncts(T1, T2) --> precedes(T1, T2) or precedes(T2, T1).

disjunctives -->
   forall(Task in resource,
      forall(T1 in Task,
         forall(T2 in Task,
            T1:uid < T2:uid implies disjuncts(T1, T2)))).
```

Supposons que l'on veuille que la recherche essaye les paires de tâches disjonctives dans l'ordre décroissant des durées. L'heuristique et le but s'écrivent alors ainsi :

```
heuristic h =
  conjunctive(greatest(T1:duration + T2:duration) for disjuncts(T1, T2)).

? tasks_domain and precedences and distances and
  minimize(search(h, disjunctives), stop:start).
```

L'heuristique `h` définit un ordre sur les termes en conjonction issus de `disjunctives` et qui s'unifient avec `disjuncts(T1, T2)`, c'est-à-dire les contraintes disjonctives. Donc, dans la conjonction produite par `disjunctives`, les sous-formules (contraintes) issues de la déclaration de règle `disjuncts(T1, T2)` seront essayées lors de la recherche dans l'ordre décroissant (**greatest**) des sommes de durées `T1:duration+T2:duration`.

Il est aussi possible d'exprimer un ordre sur les termes de chaque disjonction, par exemple celui qui préfère accomplir la tâche de plus grande durée avant l'autre. Cela consiste à ajouter l'heuristique disjonctive suivante portant sur les termes introduits par la règle `precedes(T1, T2)`, et `h` devient alors :

```
heuristic h =
  conjunctive(greatest(T1:duration + T2:duration) for disjuncts(T1, T2)) and
  disjunctive(greatest(T1:duration) for precedes(T1, T2)).
```

Le résultat (partiel) de compilation vers `SICStus-Prolog` donne le code suivant où le point-virgule dénote l'opérateur de choix non déterministe, et la virgule l'opérateur de séquence :

```
(...)
minimize((((M6+20#=<M1 ; M1+16#=<M6), (P1+20#=<P2 ; P2+13#=<P1),
           (M6+20#=<M2 ; M2+8#=<M6), (M6+20#=<M3 ; M3+8#=<M6),
           (M6+20#=<M4 ; M4+8#=<M6), (M6+20#=<M5 ; M5+8#=<M6),
           (V1+15#=<V2 ; V2+10#=<V1), (T1+12#=<T2 ; T2+12#=<T1),
           (...)
           (B4+1#=<B6 ; B6+1#=<B4), (B5+1#=<B6 ; B6+1#=<B5)
          ),
         labeling([up], [Stop])), Stop).
```

On peut constater que les contraintes disjonctives sont posées dans un ordre non croissant de somme des durées des tâches ($20+16 \geq 20+13\ldots$). Pour chaque contrainte disjonctive, la tâche de plus grande durée est essayée d'abord ($20 \geq 16$, $20 \geq 13$, ...).

L'heuristique conjonctive laisse des cas d'égalité que l'on peut par exemple départager en ajoutant une heuristique conjonctive dynamique qui préfère essayer les paires qui peuvent débuter au plus tôt. Les deux heuristiques conjonctives forment alors un vecteur de critères :

```
heuristic h =
  conjunctive(greatest(T1:duration + T2:duration) for disjuncts(T1, T2)) and
  conjunctive(least(domain_min(T1:start)+domain_min(T2:start)) for disjuncts(T1, T2)).
```

7.2 Heuristiques pour les affectations de variables

En `Cream`, les heuristiques pour le choix de variables et de valeurs dans un *labeling* (cf. Sec. 4.3) sont exprimées naturellement comme des heuristiques sur une formule qui représente l'arbre de recherche des affectations. Pour illustrer ces propos, considérons la version dynamique du modèle `Cream` du problème des n-reines (cf. Sec. 2.2) pour lequel la recherche est guidée par des heuristiques de choix de variables et de valeurs. Le code généré par la compilation de ce modèle dynamique `Cream` est présenté dans la section 6.2.

Le but

Le but du modèle consiste à poser les contraintes et à demander la recherche par affectations. Le prédicat dynamique impose au compilateur d'appliquer le schéma par génération de procédures.

```
? dynamic(let(N := 4, B := board(N),
    queens_constraints(B, N) and queens_labeling(variables(B)))).
```

Heuristiques et déclarations de règles relatives à la recherche

D'un point de vue logique, l'arbre des affectations correspond à la formule qui énonce que pour toute variable V du problème, il existe une affectation de V à une valeur dans son domaine (qui satisfait toutes les contraintes). La partie recherche s'exprime avec les déclarations de règles suivantes :

```
queens_labeling(Vars) -->
  search(ff, forall(Var in Vars, queens_labeling_var(Var))).

queens_labeling_var(Var) -->
  exists(Val in [domain_min(Var) .. domain_max(Var)], queens_labeling_val(Var, Val)).

queens_labeling_val(Var, Val) -->
  Var = Val.
```

où `queens_labeling_var(Var)` définit le sous-arbre des affectations possibles d'une variable et `queens_labeling_val(Var, Val)` son affectation à une valeur particulière.

Une bonne heuristique conjonctive pour le problème des n-reines est une instance du principe *first-fail* (cf. Sec. 4.3) : on préfère les variables dont la taille de domaine est la plus petite.

L'heuristique dynamique `Cream` (des indexicaux sont en jeu) correspondante filtre les termes des sous-formules conjonctives qui sont introduits par la règle de tête `queens_labeling_var(Var)`. Le critère d'ordre de l'heuristique a pour effet d'appliquer les appels à `queens_labeling_var(Var)` dans l'ordre croissant (`least`) des tailles de domaine de la variable `Var`.

```
heuristic ff =
  conjunctive(least(domain_size(Var)) for queens_labeling_var(Var)).
```

On peut combiner avec cette heuristique conjonctive l'heuristique disjonctive *middle out*, introduite dans un contexte statique à la fin de la section 5.1, qui ordonne les valeurs d'un domaine du milieu vers les bornes.

Dans le contexte dynamique de cet exemple, l'heuristique peut se baser sur les indexicaux et profiter de la propagation. L'heuristique disjonctive et la recherche `Cream` correspondantes s'écrivent de la manière suivante :

```
heuristic mo =
  disjunctive(least(abs((domain_max(Var) - domain_min(Var))/2 - Val))
              for queens_labeling_val(Var, Val)).

queens_labeling(Vars) -->
  search(ff and mo, forall(Var in Vars, queens_labeling_var(Var))).
```

7.3 Heuristiques pour des problèmes de placement

Nous renvoyons au chapitre 9 pour d'autres exemples de spécifications d'heuristiques de placement dans `Cream`. Ces heuristiques ont permis de résoudre effectivement des problèmes de placement réels issus de l'industrie avant le terme du projet européen Net-WMS, ce qui ne fût pas le cas des autres partenaires académiques qui travaillaient avec `SICStus-Prolog` et `Choco-Java`. D'autre part, les ingénieurs experts en placement ont pu lire ces heuristiques et y reconnaître certains principes utilisés en colisage.

Troisième partie

Évaluation

Sommaire

Chapitre 8

Bibliothèque pour la modélisation de connaissances en placement

Sommaire

Nous présentons ici la bibliothèque de modélisation des connaissances en placement `PKML` (pour *Packing Knowledge Modelling Library*) [FM09]. C'est à notre connaissance la première bibliothèque d'un langage de modélisation qui offre les définitions nécessaires à la modélisation des problèmes de placement de dimensions quelconques.

`PKML` illustre l'expressivité de `Cream` et a permis de résoudre des problèmes de placement réels issus du projet européen Net-WMS qui traite des problèmes de placement de taille réelle en logistique et dans l'industrie automobile (cf. Sec. 9.2 et Sec. 9.3). La bibliothèque a participé à la résolution effective de ces problèmes que les autres partenaires du projet travaillant avec `SICStus-Prolog` ou `Choco-Java` n'ont pas eu le temps de résoudre en prenant en compte toutes les contraintes en jeu. `PKML` repose sur le formalisme de règles, le mécanisme de réification implicite et le système de modules de `Cream`.

La bibliothèque est construite comme une hiérarchie de modules. Le module de base introduit les règles qui définissent les relations d'intervalles d'Allen [All91]. Sur ce premier module sont construites les règles qui définissent les relations du *Region Connection Calculus* [RCC92] entre orthotopes (ou boîtes) de dimensions quelconques. Le module de plus haut niveau s'attache à définir, sur la base des deux premiers, les problèmes de *Bin Packing* purs et un ensemble de règles spécifiques qui concernent la répartition du poids et la stabilité de placements en 3 dimensions. De plus, la bibliothèque fait usage

de la contrainte géométrique `geost` [BCP+07] intégrée à `Cream`.

Contraintes géométriques au-dessus de `geost`

Un objectif du projet européen a été réalisé avec le développement de la contrainte géométrique `geost`. Cette contrainte globale très efficace est dédiée aux problèmes de placement en dimensions quelconques. Elle est implantée en `SICStus-Prolog` et en `Choco-Java` et intégrée dans `Cream` sous le nom de `non_overlapping/2`. Elle prend en charge essentiellement les contraintes de non-enchevêtrement d'orthotopes de dimensions quelconques et ne fournit pas de support pour d'autres contraintes qui doivent être définies en `Cream` en l'occurrence.

Par exemple, le problème de chargement de containers (cf. Sec. 9.3) consiste en un problème de placement *Bin Packing* en 3 dimensions enrichi de plusieurs contraintes spécifiques de répartition de poids et de stabilité :

```
container_loading_constraints(Items, Bin, BinSize, Dims) -->
  domains(Items, BinSize, Dims) and
  containmentAE(Items, [Bin], Dims) and
  non_overlapping(Items, Dims) and
  gravity(Items) and
  blocked(Items) and
  stack_height(Items) and
  stack_area(Items) and
  stack_alignment(Items) and
  stack_support_area(Items, 100) and
  stack_weight_sum(Items) and
  weight_balancing(Items, Bin, 1, 10).
```

Par ailleurs, nous avons participé à l'évolution de `geost` avec règles [CBM08]. Dans cette version, `geost` est paramétrée par un langage de règles qui constitue un sous-ensemble strict des règles `PKML` restreintes aux contraintes linéaires. Il a été montré que ce langage plus simple que `PKML` est compilable efficacement vers les indexicaux k-dimensionnels du noyau de `geost`.

Extensibilité du langage et réification

La réification est un mécanisme très utile pour composer arbitrairement des contraintes, mais dans un langage comme `SICStus-Prolog` par exemple, il est nécessaire de manipuler les variables associées à chaque contrainte réifiée. `Cream` offre une réification implicite qui permet de composer arbitrairement des règles sans avoir à manipuler explicitement ces variables réifiées. Considérons par exemple la définition d'une contrainte disjonctive de précédence entre tâches. En `SICStus-Prolog` (à gauche), la réification de la contrainte est explicitée par la variable `R` alors qu'elle est implicite en `Cream` (à droite).

```
precedes(T1, T2, R) :-
  end(T1, ET1), origin(T2, OT2),
  (ET1 #< OT2) #<=> R.
```

```
precedes(T1, T2) -->
  end(T1) < origin(T2).
```

8.1 Formes et objets k-dimensionnels

8.1.1 Assemblages de formes k-dimensionnelles

Les formes définies dans `PKML` sont des assemblages d'orthotopes et vivent dans $\mathbb{Z}^k$. Un orthotope est une généralisation du rectangle que l'on peut définir comme un produit cartésien d'intervalles.

Un *point* dans cet espace est représenté par une liste de ses k coordonnées entières `[i1,...,ik]`. Ces coordonnées peuvent être des variables ou des valeurs constantes entières.

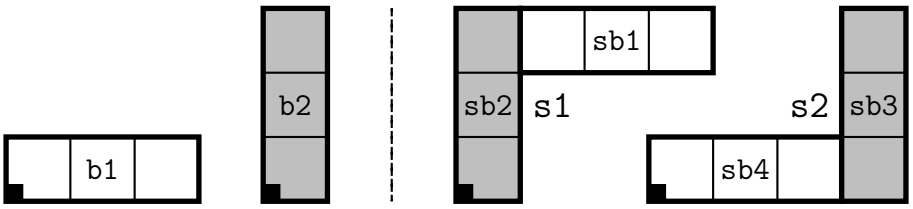
Une boîte ou *box* est un orthotope dans $\mathbb{Z}^k$, représenté par un enregistrement contenant un attribut taille ou *size* indiquant la liste des longueurs de la boîte dans chaque dimension.

Une boîte déportée ou *shifted box* est représentée par un enregistrement contenant un attribut *box* et un attribut *offset* donnant les coordonnées locales (à une forme) de la boîte dans chaque dimension.

Dans `PKML`, une forme ou *shape* est un *assemblage rigide de boîtes déportées*.

Une forme est alors représentée par un enregistrement contenant un attribut *sboxes* pour la liste des boîtes déportées constituant la forme. Par exemple :

```
% Soit b1 et b2 deux boîtes.
b1 = {size = [3, 1]}.
b2 = {size = [1, 3]}.

% Soit sb1, sb2, sb3 et sb4 quatre boîtes déportées.
sb1 = {box = b2, offset = [1, 2]}.
sb2 = {box = b1, offset = [0, 0]}.
sb3 = {box = b2, offset = [3, 0]}.
sb4 = {box = b1, offset = [0, 0]}.

% Soit s1, s2 deux formes.
s1 = {sboxes = [sb1, sb2]}.
s2 = {sboxes = [sb3, sb4]}.
```



Les déclarations suivantes définissent respectivement : la construction d'une boîte et son volume ; et la construction d'une boîte déportée et sa taille dans une dimension donnée.

```
make_box(L) = {size = L}.
box_volume(B) = prod(B:size).
make_sbox(B, O) = {box = B, offset = O}.
sbox_size(SB, D) = nth(D, SB:box:size).
```

Le constructeur d'une forme quelconque (un assemblage de boîtes) et d'une forme composée d'une seule boîte, l'origine locale, la fin et la taille de la forme dans une dimension donnée, et le volume de la forme (en supposant que les boîtes de l'assemblage de s'enchevêtrent pas) :

```
make_shape(SBs) = {sboxes = SBs}.
make_shape_box(B) = make_shape([make_sbox(B, map(_ in B:size, 0))]).

shape_origin(S, D) = min(map(SB in S:sboxes, sbox_offset(SB, D))).
shape_end(S, D) = max(map(SB in S:sboxes, sbox_end(SB, D))).
shape_size(S, D) = shape_end(S, D) - shape_origin(S, D).
shape_volume(S) = sum(map(SB in S:sboxes, box_volume(SB:box))).
```

8.1.2 Alternatives d'assemblages de formes

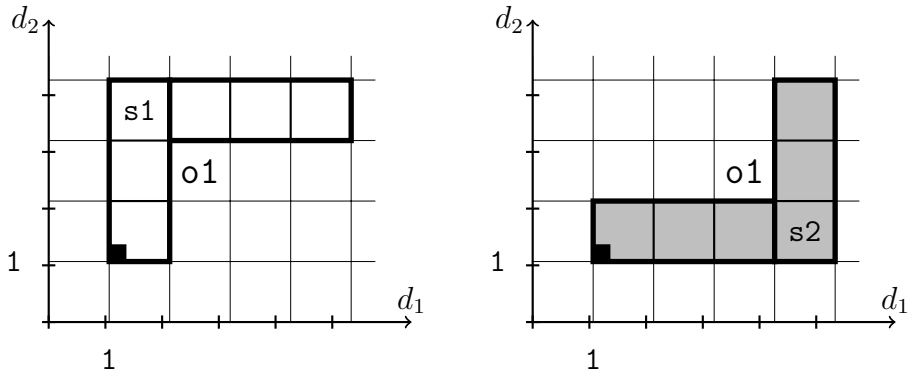
Un objet PKML ou *object*, désignant un container ou un carton, est un enregistrement portant un attribut *shapes* indiquant un *ensemble de formes alternatives*, un point *origin* et un attribut *shape_index* représentant la forme courante de l'objet. Des attributs optionnels tels que le poids pourraient aussi apparaître dans l'enregistrement. On appelle *polymorphe* un objet qui peut prendre plus d'une forme.

Les formes alternatives d'un objet peuvent représenter des rotations discrètes d'une forme, comme par exemple les formes s1 et s2 de l'objet o1 dans la figure ci-dessous, ou des formes complètement différentes dans un problème de configuration. Nous ne distinguons pas entre les traits d'un objet à placer et ceux d'un contenant, puisque le contenant d'un niveau peut devenir un objet à placer au niveau supérieur (cf. sections 9.2 et 9.3).

```
% Soit o1 l'objet polymorphe de forme s1 ou s2
o1 = {shapes = [s1, s2], shape_index = S, origin = [_, _]}.
```

```
? let(Origin := o1:origin,
    o1:shape_index = 1 and
    nth(1, Origin) = 1 and
    nth(2, Origin) = 1).
```

```
? let(Origin := o1:origin,
    o1:shape_index = 2 and
    nth(1, Origin) = 1 and
    nth(2, Origin) = 1).
```



Il est à noter que si les tailles des boîtes composant une forme sont connues, les déclarations de taille et de volume ont pour valeur des entiers fixés ; alors que si les tailles sont inconnues, c'est-à-dire des variables, les expressions ont pour valeur des termes non clos. Ces termes sont nécessaires en PKML pour définir, à l'aide de variables réifiées, la fin, le bout dans une dimension donnée ou le volume d'un objet polymorphe, comme suit :

```
make_object(SL, OL, S) = {shapes=SL, shape_index=S, origin=OL}.
make_object_shape(S, L) = {shapes=[S], shape_index=1, origin=L}.

origin(O, D) = nth(D, O:origin).
end(O, D) = origin(O, D) +
            sum(map(S in [1 .. length(O:shapes)],
                (O:shape_index = S) * shape_end(nth(S, O:shapes), D))).
size(O, D) = sum(map(S in [1 .. length(O:shapes)],
                  (O:shape_index = S) * shape_size(nth(S, O:shapes), D))).
area(O, D1, D2) = size(O, D1) * size(O, D2).
volume(O) = sum(map(S in [1 .. length(O:shapes)],
                 (O:shape_index = S) * shape_volume(nth(S, O:shapes)))).
```

8.2 Relations et règles de placement génériques

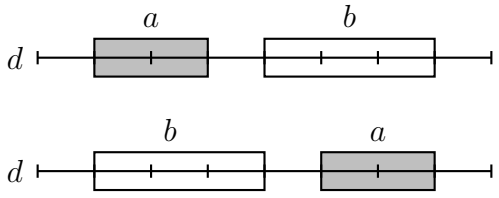
La bibliothèque PKML repose sur les relations d'intervalles d'Allen [All91] en une dimension, et les relations topologiques du Region Connection Calculus (RCC) [RCC92] en dimensions supérieures. Dans les deux cas, les relations présentent deux bonnes propriétés : elles sont complètes et mutuellement exclusives. Toute paire de polytope entretient une des relations mais pas deux à la fois. Toutes ces relations sont prédéfinies entre orthotopes (entiers) dans dans deux bibliothèques [FM08]. Des relations supplémentaires sont ajoutées pour des raisons de commodité ou d'efficacité,

8.2.1 Relations d'Allen

Les relations les plus simples définies dans PKML sont des relations entre orthotopes uni-dimensionnels, donc des intervalles. Il existe 13 relations d'Allen (6 relations avec leur symétrique plus la relation d'égalité) dont nous décrivons un sous-ensemble ici.

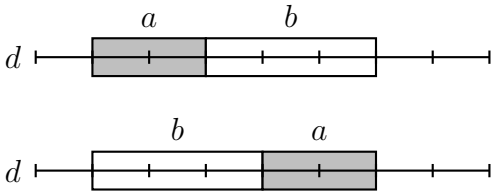
precedes/3, preceded_by/3 : un intervalle a précède un intervalle b dans une dimension d lorsque a se termine strictement avant le début de b.

```
precedes(A, B, D) -->
  end(A, D) < origin(B, D).

preceded_by(A, B, D) -->
  end(B, D) < origin(A, D).
```



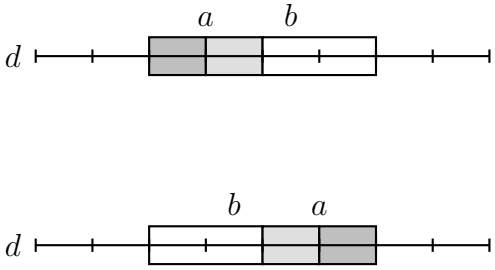
meets/3, met_by/3 : un intervalle a rencontre un intervalle b dans la dimension d lorsque a se termine avant le début b et le touche.

```
meets(A, B, D) -->
  end(A, D) = origin(B, D).

met_by(A, B, D) -->
  end(B, D) = origin(A, D).
```



overlaps/3, overlapped_by/3 : un intervalle a chevauche un intervalle b lorsque a commence strictement avant b et se termine strictement après le début de b et strictement avant la fin de b.

```
overlaps(A, B, D) -->
  origin(A, D) < origin(B, D) and
  end(A, D) < end(B, D) and
  origin(B, D) < end(A, D).

overlapped_by(A, B, D) -->
  origin(B, D) < origin(A, D) and
  origin(A, D) < end(B, D) and
  end(A, D) > end(B, D).
```



De façon générale, deux intervalles a et b se chevauchent lorsque leur intersection est non vide. Par définition, cette notion de chevauchement n'est pas définie par une relation d'Allen unique. Puisqu'elle est nécessaire dans tout problème de placement, bibliothèque introduit la relation supplémentaire **overlaps_sym/3** :

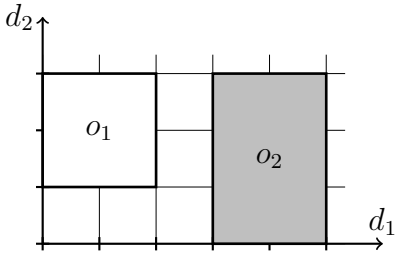
```
overlaps_sym(A, B, D) -->
  end(A, D) > origin(B, D) and end(B, D) > origin(A, D).
```

8.2.2 Relations du Region Connection Calculus (RCC)

Les relation de RCC sont définies sur les relations uni-dimensionnelles d'Allen.

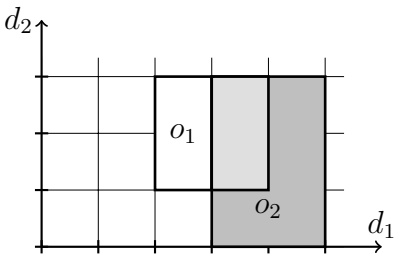
disjoint/3 : deux orthotopes sont disjoints s'il existe au moins une dimension dans laquelle la projection de l'un précède la projection de l'autre.

```
disjoint(O1, O2, Dims) -->
  exists(D in Dims,
    precedes(O1, O2, D) or
    preceded_by(O1, O2, D)).
```



overlap/3 : deux orthotopes se chevauchent ou s'enchevêtrent si leurs projections se chevauchent dans toutes les dimensions.

```
overlap(O1, O2, Dims) -->
  forall(D in Dims,
    overlaps_sym(O1, O2, D)).
```



8.2.3 Règles de placement

Ces relations génériques entre orthotopes sont utilisées dans PKML pour définir des règles de placement génériques pour les problèmes *Bin Packing* comme suit :

```
non_overlapping_bin(Items, Dims) -->
  forall(O1 in Items,
    forall(O2 in Items,
      O1:uid < O2:uid implies
      not overlap(O1, O2, Dims))).
```

```
containmentAE(Items, Bins, Dims) -->
  forall(I in Items,
    exists(B in Bins,

      rcc::contains_meets(B, I, Dims))).
```

Les règles définissent respectivement le non-chevauchement d'un ensemble d'objets sur un nombre de dimensions donné, et l'inclusion de tous les objets dans le contenant.

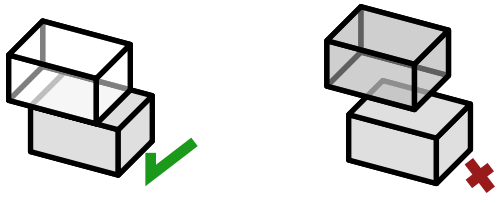
```
bin_packing_binary(Items, Bins, Dims) -->
  containmentAE(Items, Bins, Dims) and non_overlapping_bin(Items, Dims) and
  labeling(Items).
```

La règle qui définit le problème Bin Packing demande, pour un ensemble d'objets et de contenants définis sur un nombre de dimension donnés, s'il existe une solution à la conjonction des deux contraintes précédentes. Dans un souci d'efficacité, la contrainte géométrique **geost** est préférée en pratique à la version binaire de la contrainte de non-chevauchement que l'on vient de présenter.

Le bibliothèque définit aussi des relations dans l'espace naturel à trois dimensions.

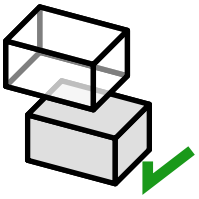
on_top/2 : une boîte o_1 est sur une boîte o_2 si les deux boîtes se chevauchent sur le plan du « sol » (sur le plan décrit pas les axes des deux premières dimensions) et o_1 est touchée par o_2 dans la dimension « verticale » (la troisième dimension).

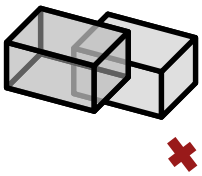
```
on_top(O1, O2) -->
  overlap(O1, O2, [1, 2]) and
  met_by(O1, O2, 3).
```



above/2 : une boîte o_1 est au-dessus d'une boîte o_2 si leurs projections au « sol » se chevauchent et si la projection de o_1 dans la dimension « verticale» est précédée par o_2 ou si o_1 est touchée par o_2 dans cette même dimension.

```
above(O1, O2) -->
  overlap(O1, O2, [1, 2]) and
  (preceded_by(O1, O2, 3) or
   met_by(O1, O2, 3)).
```





8.3 Règles de placement spécifiques

Les règles métier de placement existent et sont définies dans PKML pour prendre compte des règles de bon sens ou les spécifications industrielles qui dépassent la portée des problèmes Bin Packing purs [CD85].

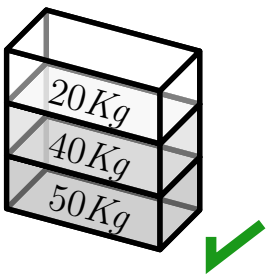
8.3.1 Règles relatives au poids

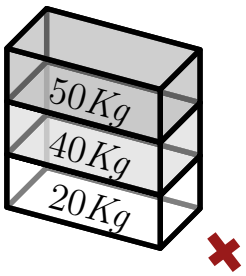
Par exemple, les règles suivantes expriment des contraintes de poids pour un placement admissible en logistique.

Les orthotopes considérés sont munis de deux propriétés supplémentaires : un poids `weight` et un facteur `piling`, dont le produit avec `weight` exprime le poids qu'un objet peut supporter (sa gerbabilité).

stack_weight/1 : Quelque soit la paire d'objets $(o_1, o_2) \in Items \times Items$, si o_1 est au-dessus de o_2 alors o_1 est plus léger que o_2.

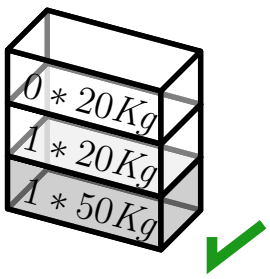
```
stack_weight(Items) -->
  forall(O1 in Items,
    forall(O2 in Items,
      above(O1, O2) implies lighter(O1, O2))).
```

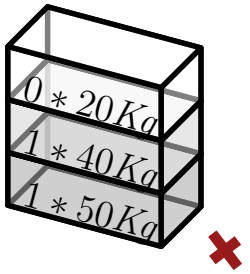




stack_weight_sum/1 : Pour tout $o_1 \in Items$, la somme des poids des objets de l'ensemble $\{o_2 \mid \texttt{above}(o_2, o_1), o_2 \in Items\}$ est inférieure à la capacité de support de o_1.

```
stack_weight_sum(Items) -->
  forall(O1 in Items,
    sum(map(O2 in Items, above(O2, O1) * O2:weight)) =< O1:piling * O1:weight).
```

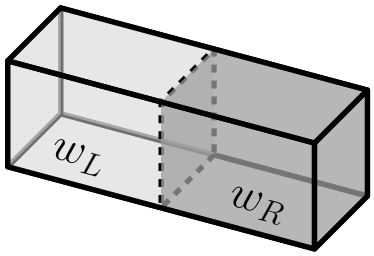




weight_balancing/4 : Soit $L = \{o \mid \texttt{stand\_in\_first\_half}(o, bin, d), o \in Items\}$ l'ensemble des objets de *Items* dont les projections dans la première moitié (ou partie « gauche ») de la projection dans la dimension d du contenant *bin*. Soit $R = \{o \mid \texttt{stand\_in\_second\_half}(o, bin, d), o \in Items\}$ l'ensemble des objets inclus dans la seconde moitié (ou partie « droite »). Soit w_L et w_R les sommes des poids des objets dans la partie gauche et dans la partie droite, respectivement.

La règle énonce que le rapport de poids $max(w_L, w_R)/min(w_L, w_R)$ ne doit pas excéder un certain entier naturel *ratio*.

```
stand_in_first_half(Item, Bin, D) --> end(Item, D) =< size(Bin, D) / 2.
stand_in_second_half(Item, Bin, D) --> origin(Item, D) >= size(Bin, D) / 2.

weight_balancing(Items, Bin, D, Ratio) -->
  let(WL = sum(map(O in Items, O:weight * stand_in_first_half(O, Bin, D))),
      WR = sum(map(O in Items, O:weight * stand_in_second_half(O, Bin, D))),
    100 * max(WL, WR) =< (100 + Ratio) * min(WL, WR)).
```



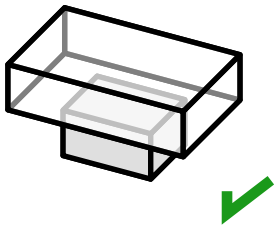
$$\frac{max(w_L,\ w_R)}{min(w_L,\ w_R)} \leq 1 + \frac{ratio}{100}$$

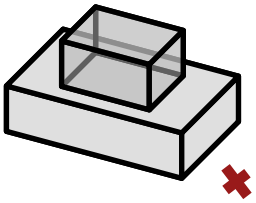
8.3.2 Règles relatives aux longueurs et aux surfaces

Les règles suivantes expriment des contraintes de tailles et de surfaces d'objet appartenant à même une pile.

stack_area/1 : Quelque soit la paire d'objets $(o_1, o_2) \in Items \times Items$, si o_1 repose sur o_2 alors la base de o_1 (sa surface dans le plan décrit par les axes des deux premières dimensions) est supérieure à la base de o_2.

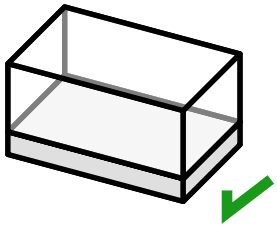
```
stack_area(Items) -->
  forall(O1 in Items,
    forall(O2 in Items,
      on_top(O1, O2) implies larger_area(O1, O2, 1, 2))).
```

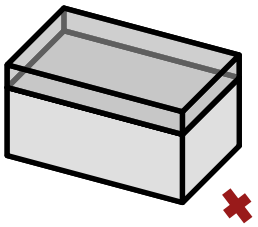




stack_height/1 : Quelque soit la paire d'objets $(o_1, o_2) \in Items \times Items$, si o_1 repose sur o_2 alors la hauteur o_1 (sa taille dans la troisième dimension) est supérieure à la hauteur de o_2.

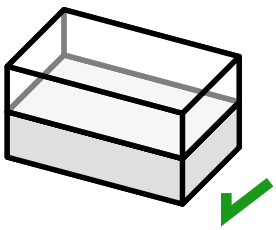
```
stack_height(Items) -->
  forall(O1 in Items,
    forall(O2 in Items,
      on_top(O1, O2) implies larger_size(O1, O2, 3))).
```

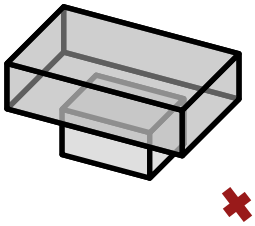


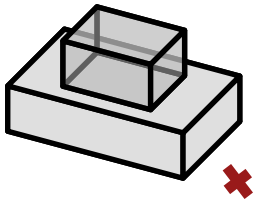


stack_oversize/2 : Quelque soit la paire d'objets $(o_1, o_2) \in Items \times Items$, si leurs projections au sol (plan décrit par les axes des deux premières dimensions) se chevauchent alors la différence (en valeur absolue) entre leurs origines et la différence entre leurs fins dans les deux premières dimensions ne doivent pas être strictement supérieures à l'entier naturel *length*.

```
stack_oversize(Items, Length) -->
  forall(O1 in Items,
    forall(O2 in Items,
      overlap(O1, O2, [1,2]) implies
      forall(D in [1, 2], oversize(O1, O2, D) =< Length))).
```





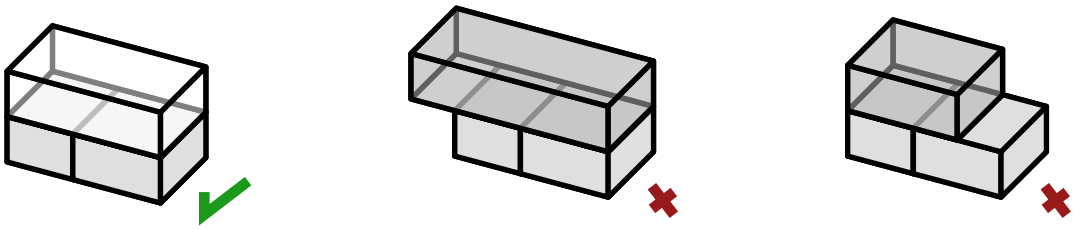


`stack_support_area/2` : Pour tout objet o_1, ou bien il repose au sol (sa coordonnée dans la troisième dimension vaut 0) ; ou bien sa base est recouverte au moins à la hauteur de l'entier naturel *percent* par l'ensemble des objets sur lesquels o_1 repose.

```
stack_support_area(Items, Percent) -->
  forall(O1 in Items,
    origin(O1, 3) = 0 or
    sum(map(O2 in Items, on_top(O1, O2) * overlap_area(O1, O2, 1, 2)))
    >=
    (area(O1, 1, 2) * Percent) / 100).
```

`stack_alignment/1` : Quelque soit la paire d'objets $(o_1, o_2) \in \mathit{Items} \times \mathit{Items}$, si les projections au sol de o_1 et o_2 se chevauchent, alors o_1 et o_2 sont alignés à l'origine ou alignés à la fin dans la première dimension (par convention la plus longue).

```
stack_alignment(Items) -->
  forall(O1 in Items,
    forall(O2 in Items,
      overlap(O1, O2, [1, 2])
      implies
      origin(O1, 1) = origin(O2, 1) or end(O1, 1) = end(O2, 1))).
```



Dans le contexte de la bibliothèque `PKML`, la proposition 4 a pour conséquence :

Proposition 6. *Les modèles `PKML` contenant des listes d'au plus l éléments génèrent des programmes de contraintes de taille $O(l^4)$ en présence à la fois de d'alternatives de formes et d'assemblages de boîtes, $O(l^3)$ en présence d'une des deux caractéristiques uniquement, et $O(l^2)$ en présence de boîtes simples.*

Chapitre 9

Modèles et résolution de problèmes de placement

Sommaire

Ce chapitre présente trois problèmes de placement académique et issus de l'industrie de difficultés croissantes.

Ces problèmes illustrent l'expressivité et l'efficacité du langage ainsi que sa capacité à exprimer de façon déclarative les stratégies de branchement et les heuristiques d'ordonnancement. Ce trait original de `Cream` nous a permis de construire et d'expérimenter rapidement des heuristiques menant à la résolution effective de problèmes de placement réels issus de l'industrie (cf. Sec. 9.3). Cela avant le terme du projet européen NetWMS, ce qui ne fût pas le cas des autres partenaires académiques qui travaillaient avec `SICStus-Prolog` et `Choco-Java`.

Dans les modèles qui suivent, la syntaxe des `fold` est légèrement différente que celle présentée jusque là. En particulier, les combinateurs de listes sont de la forme plus standard `foldl(`X `in` l`,` p`,` i`,` e`)` où

- X est une variable liée dans e prenant à chaque itération comme valeur l'élément suivant de l ;
- p est un symbole de prédicat binaire défini par ailleurs et qui attend en premier argument (*fold left*) l'accumulateur et en second argument l'expression $e[X]$;
- et i est l'élément initial pour l'accumulateur.

9.1 Problème de placement optimal de carrés dans un rectangle

9.1.1 Définition

Le problème de placement optimal de carrés dans un rectangle (*Optimal Rectangle Packing*) [Kor04] est un problème académique qui consiste à trouver le plus petit rectangle (qui ne soit pas un carré) contenant n carrés de tailles $s_i = i$, $1 \leq i \leq n$.

9.1.2 Modèle et évaluation

Le modèle `Cream` est basé sur la modélisation et la procédure de recherche de Simonis et O'Sullivan (S&O) [SO08], une approche par décomposition, et implanté en `SICStus-Prolog`. Les résultats du programme `SICStus-Prolog` de S&O ont été confrontés au benchmark de Korf [Kor04] et ont améliorés les meilleurs temps de résolution connus de facteurs compris entre 100 et 300. Nous comparons à la fin de cette section les performances du programme de S&O avec celles du programme généré à partir du modèle `Cream`.

L'idée de la stratégie de branchement est d'énumérer les rectangles candidats par ordre d'aires ascendant et, pour chaque rectangle, tenter d'y placer les carrés. Cette stratégie est dynamique au sens où les problèmes de placement des n carrés successifs dépendent de l'instanciation donnée par l'énumération des rectangles candidats. Par construction, au premier placement réussit correspond le plus petit rectangle englobant. L'efficacité du modèle réside dans la décomposition du problème et dans l'association des contraintes globales de non-enchevêtrement bi-dimensionnel (`disjoint2/3`) et de cumul unidimensionnel (`cumulative/4`) avec une recherche basée sur le découpage par intervalles ; découpage des domaines de toutes les coordonnées dans la première dimension d'abord puis découpage dans la seconde.

Imports et données

```
import('Cream/lib/pkml/pkml').
import('Cream/lib/search/domain_splitting').
% Constructeur du rectangle.
% Il porte sa propre aire pour faciliter l'énumération.
make_object_shape_area(S, OL, A) = {shapes:[S], shape_index:1, origin:OL, area:A}.
bin = make_object_shape_area(make_shape_size([_, _]), [1, 1], _).
% Constructeur de formes carrées.
% Les carrés sont ordonnés du plus grand au plus petit.
square(S) = make_shape_size([S, S]).
squares(N) = map(Size in reverse([2 .. N]), square(Size)).
% Constructeur des carré à placer.
% Les objets à placer sont des carrés munis de coordonnées.
item(S) = make_object_shape(S, [_, _]).
items(N) = map(S in squares(N), item(S)).
```

```
% Taille des objets dans chaque dimension.
w(0) = size(0, 1).
h(0) = size(0, 2).
```

Le but

Le but du modèle montre la décomposition du problème : énumération de rectangles englobants (`solve_bin_subproblem/6`) et résolution du problème de placement des N carrés (`solve_items_subproblem/3`).

```
% N : taille de l'instance
% Items : liste des carrés à placer
% Width : largeur du rectangle (longueur dans la première dimension)
% Height : hauteur du rectangle (longueur dans la seconde dimension)
% Area : aire du rectangle
% LB : borne inférieur de la solution optimale
% UB : borne supérieure de la solution optimale
? let(N := 25,
      Items := items(N),
      Width := size(bin, 1),
      Height := size(bin, 2),
      Area := bin:area,
      LB := sum(map(I in Items, area(I, [1, 2]))) + 1,
      UB := LB + 200,
    solve_bin_subproblem(Width, Height, Area, LB, UB, N) and
    solve_items_subproblem(Items, Width, Height)).
```

(1) Énumération des rectangles

Il s'agit de poser les domaines des variables représentant largeur (`W`) et hauteur (`H`) des rectangles candidats, puis de poser les contraintes sur ces variables. Ces contraintes mettent en relation largeur, hauteur et aire (`A`) ; imposent que la hauteur soit supérieure à la largeur ; et considèrent les plus grands carrés qui ne peuvent pas être empilés les uns sur les autres et qui doivent donc tenir dans la dimension longue.

```
bin_domain(W, H, A, L, U, N) -->
  domain([W, H], N, L) and
  domain([A], L, U).

bin_constraints(W, H, A, N) -->
  let(K := (W + 1)/2,
      A = W * H and
      H >= W and
    (W >= 2 * N - 1 or H >= (N * N + N - (K - 1) * (K - 1) - (K - 1)) / 2)).
```

La stratégie de branchement énumère d'abord les aires puis la largeur dans l'ordre croissant.

```
bin_search -->
  variable_ordering([is(^:area), is(w(^)), is(h(^))]) and labeling(bin).
```

(2) Placement des carrés

De la même façon, le sous-problème de placement des carrés pose les domaines des coordonnées, puis les contraintes et enfin définit la stratégie de branchement. Ici, (`lower_quadrant/3`) brise une symétrie du problème en imposant que le plus grand carré soit nécessairement placé dans le quart inférieur gauche du rectangle candidat.

```
items_domain(Items, W, H) -->
  forall(It in Items,
    let(X := x(It),
        Y := y(It),
        S := w(It),
      domain(X, 1, W - S + 1) and
      domain(Y, 1, H - S + 1) )) and
      lower_quadrant(Items, W, H).

% Bris de symétrie : le premier carré est placé nécessairement
% dans le quart inférieur gauche du rectangle englobant.
lower_quadrant(Items, W, H) -->
  let(FstIt := nth(1, Items),
       X := x(FstIt),
       Y := y(FstIt),
       S := w(FstIt),
    domain(X, 1, (W - S + 2) / 2) and
    domain(Y, 1, (H + 1) / 2)).

% Contraintes globales portant sur les carrés.
% disjoint2/3 : non-envechêtrement bi-dimensionnel
% cumulative/4 : plafonnement du cumul des carrés dans chaque dimension
items_constraints(Items, W, H) -->
  let(Xs := map(It in Items, x(It)),
      Ys := map(It in Items, y(It)),
      Ss := map(It in Items, w(It)),
    disjoint2(Xs, Ys, Ss) and
    cumulative(Xs, Ss, Ss, H) and
    cumulative(Ys, Ss, Ss, W)).
```

La stratégie de branchement procède par découpage d'intervalles dont les primitives `interval_split/3` et `dichotomic_split/1` sont définies dans la bibliothèque. L'application et la combinaison des ces primitives se présentent ainsi :

1. pour tous les carrés, découper le domaine des coordonnées dans la première dimension, sauf les 6 plus petits ; puis, pour tous les carrés, fixer les coordonnées dans la même dimension ;
2. pour tous les carrés, découper le domaine des coordonnées dans la seconde dimension ; puis, pour tous les carrés, fixer les coordonnées dans la même dimension.

Le découpage s'effectue par intervalles de longueur le tiers du carré considéré et selon des origines ascendantes. Les coordonnées sont ensuite fixées par une recherche dichotomique standard sur leur domaine.

```
% XSs : liste de couples (X_i, S_i)
% YSs : liste de couples (Y_i, S_i) avec
% X_i coordonnée du carré i dans la première dimension,
% Y_i coordonnée du carré i dans la seconde dimension,
% S_i taille du carré i.
items_search(Items, W, H) -->
  let(XSs := map(It in Items, {coord : x(It), siz : w(It)}),
      YSs := map(It in Items, {coord : y(It), siz : h(It)}),
      Min := 1,
      MaxX := W + 1,
      MaxY := H + 1,
    dynamic(
      search(
        forall(XS in XSs,
          XS:siz > 6 implies
          interval_split(XS:coord, Min, MaxX, max(1, (XS:siz*3)/10))) and
        forall(XS in XSs, dichotomic_split(XS:coord)) and

        forall(YS in YSs,
          interval_split(YS:coord, Min, MaxY, max(1, (YS:siz*3)/10))) and
        forall(YS in YSs, dichotomic_split(YS:coord))))).
```

Le découpage d'un domaine, initialement $[min,\ max]$, par intervalles de longueur l comprend $n = (max - min)/l$ coupes. Comme l'illustre la figure 9.1, à chaque étape $c \in \{1, \ldots, n\}$ le domaine est restreint à l'intervalle $[min + (c-1) * l,\ min + c * l]$.

```
interval_predicate(List, Ctn) -->
  let(Var := nth(1, List),
      Len := nth(2, List),
      Cut := domain_min(X) + Len,
    Var =< Cut or (Var > Cut and Ctn)).

interval_split(X, Min, Max, L) -->
  foldr(C in [1 .. (Max - Min) / L], interval_predicate, true, [X, L]).
```

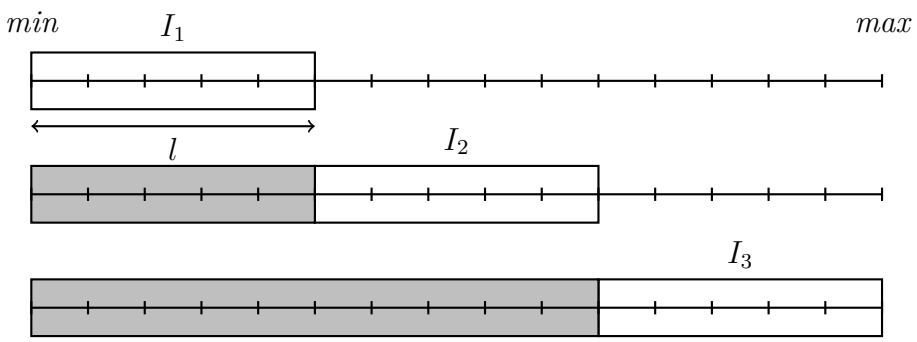


FIGURE 9.1: Découpage d'un domaine $[min,\ max]$ par intervalles I_k de longueur l.

Solution

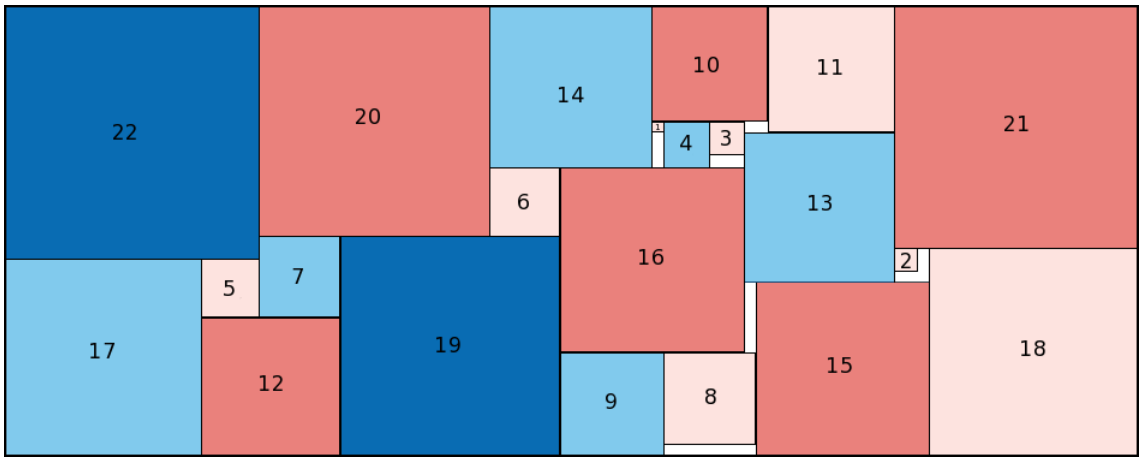


FIGURE 9.2: Une solution de *Optimal Rectangle Packing*, $n = 22$.

La table 9.1 compare les temps de résolution du modèle original avec le code généré Cream. On constate que le code généré est plus lent à résoudre d'un facteur 2 pour la plupart des instances mais qui tend à diminuer avec la taille des instances.

n	Compilation	Résolution	
		Cream	Réf.
18	0.650	17	9
19	0.700	17	8
20	0.780	30	17
21	0.810	100	63
22	0.870	430	297
23	0.930	2700	1939
24	0.980	3900	2887
25	1.060	27020	20713

TABLE 9.1: Temps d'exécution pour *Optimal Rectangle Packing*, en secondes (Linux / Intel Core2 Quad CPU, 2.83GHz).

Conclusion

Nous avons présenté un modèle `Cream` pour le problème de placement optimal de carrés dans un rectangle construit sur la base de la proposition de S&O. `Cream` permet de formuler concisément ce problème de nature dynamique et de le résoudre avec un surcoût limité vis-à-vis du programme de référence.

C'est un modèle qui ne pourrait être formulé dans les langages de modélisation qui adoptent une compilation statique comme Zinc ou Essence. La stratégie de branchement est exprimée dans le modèle `Cream` grâce à deux déclarations de règles définissant des recherches par découpage d'intervalles. Le temps de compilation du modèle `Cream`

ne dépasse pas 1 seconde et les temps d'exécution pour les différentes instances du problème tendent à un surcoût raisonnable d'environ 25% par rapport au programme `SICStus-Prolog` de S&O.

9.2 Problème de chargement de palette

9.2.1 Définition

Le problème de chargement de pallettes (*Pallet Loading*) est un problème industriel de placement tri-dimensionnel qui consiste à placer un ensemble de cartons sur une pallette. Les cartons sont des parrallèlépidèdes rectangles ou cuboïdes, ainsi que l'espace de placement dont la base est une palette. Les cartons peuvent subir une rotation d'un quart de tour autour de l'axe orthogonal à la palette.

Par rapport au problème de placement précédent, trois niveaux de difficulté supplémentaires sont introduits :

- c'est un problème en *3 dimensions*, en fait ce premier problème se réduit en un problème de placement 2+1D, c'est-à-dire par couches d'objets, chaque couche étant solution d'un problème en 2 dimensions ;
- les objets peuvent subir des *rotations*, ce ne sont pas des carrés ;
- une *nouvelle contrainte* sur la surface de la base des boîtes composant chaque couche est ajoutée.

Les palettes sont remplies de préférence avec un ensemble d'objets homogène en terme de référence, idéalement avec une voire deux références différentes. Les solutions doivent respecter les contraintes suivantes :

- Si plusieurs références de cartons sont à placer sur une palette, les objets de plus grande base doivent être placés sur les objets de base plus petite.

Chaque palette est destinée à être placée à son tour dans un container (cf. section suivante : Problème *Container Loading*).

9.2.2 Modèle et évaluation

Imports et données

Une instance du problème *Pallet Loading* implique typiquement 3 modules qui définissent :

- les données propres à l'instance (définitions du contenant et des contenus),
- la stratégie de branchement utilisée pour la résolution,
- et le modèle commun à toutes les instance de *Pallet Loading* avec placement « par couches ».

```
import('Cream/examples/packing/pallet_loading_1.data').
import('Cream/examples/packing/pallet_loading_1.strategy').
import('Cream/examples/packing/pallet_loading_layer.model').
```

Le module concernant les données du problème définit :

- la pallette : ses dimensions réelles et réduites ;
- pour chaque référence de carton : les dimensions réelles et réduites ainsi que la cardinalité.

```
% Les dimensions du problème
dimensions = [1, 2, 3].

% Réduction des longueurs par dimension
% selon le plus grand commun diviseur
canonical_size(RefSize) = map(D in dimensions, nth(D, RefSize) / gcd(D)).

% Définition de la palette
pf82ref_size = [2240, 2240, 992].
pf82_size = canonical_size(pf82ref_size).
bin_data = {ref_size = pf82ref_size, size = pf82_size}.

% Définitions des cartons
c11ref_size = [600, 400, 200].
c11_size = canonical_size(c11ref_size).
items_c11 = {ref_size = c11ref_size, size = c11_size, card = 80}.
items_data = [items_c11].
```

Le modèle importe la bibliothèque `PKML` afin de profiter des définitions de constructeurs, fonctions et autres règles impliquées dans les problèmes de placement.

Afin de profiter d'une éventuelle factorisation des longueurs dans une instance de problème, les longueurs d'une référence de carton i dans la dimension d sont divisées par le plus grand commun diviseur de toutes les longueurs potentiellement impliquées [1] dans d. La taille réduite d'une référence i dans la dimension d est donc $c_i^d = r_i^d / pgcd(R_d)$, où R_d est l'ensemble des longueurs impliquées dans la dimension d.

```
import('Cream/lib/pkml/pkml').

gcd(I) = _.

% Dans chaque dimension d, les longueurs des objets du problème
% sont divisées par leur plus grand commun diviseur pgcd(d).
reduced_sizes(BinRefSize, ItemsRefSize, Dimensions) -->
  let(BinWidthRef := nth(1, BinRefSize),
      BinHeightRef := nth(2, BinRefSize),
      BinLengthRef := nth(3, BinRefSize),
      ItemsWidthRef := map(IRS in ItemsRefSize, nth(1, IRS)),
      ItemsHeightRef := map(IRS in ItemsRefSize, nth(2, IRS)),
      ItemsLengthRef := map(IRS in ItemsRefSize, nth(3, IRS)),
    gcd([BinWidthRef] ++ ItemsWidthRef ++ ItemsHeightRef, gcd(1)) and
    gcd([BinHeightRef] ++ ItemsWidthRef ++ ItemsHeightRef, gcd(2)) and
    gcd([BinLengthRef] ++ ItemsLengthRef, gcd(3))).
```

1. potentiellement car les objets sont polymorphiques (une rotation autorisée).

Le but

Le but demande simplement de placer les cartons sur la pallette après avoir si possible réduit les longueurs.

```
? dynamic(
    let(Dimensions := dimensions,
        BinData := bin_data,
        BinRefSize := BinData:ref_size,
        ItemsData := items_data,
        ItemsRefSize := map(ItemsData in ItemsData, ItemsData:ref_size),
      reduced_sizes(BinRefSize, ItemsRefSize, Dimensions) and
      pallet_loading(BinData, ItemsData, Dimensions))).
```

Les contraintes

Placer les cartons sur la pallette, c'est les faire tenir dans l'espace de la pallette (`containmentAE/3`) sans qu'ils ne s'enchevêtrent (`non_overlapping/2`). Cette contrainte est définie par la contrainte globale `geost`.

Les contraintes `containmentAE/3` et `domains/3` ne peuvent être confondues car les objets sont polymorphes. Les domaines doivent être décidés statiquement, donc `domain/3` ne peut que considérer qu'une sous-approximation des longueurs dans les dimensions qui sont permutées par la rotation ; et `containmentAE/3` s'assure grâce à des contraintes réifiées, une fois l'orientation décidée, que les objets tiennent bien dans la pallette.

De plus, puisque les cartons d'une même références forment une classe d'équivalence pour le placement (ils sont rigoureusement identiques et sont placés, pour chaque coordonnées, des plus petites valeurs de domaines aux plus grandes), la contrainte globale `lexicographic/1` permet de briser la symétrie de permutation des cartons d'une même référence.

Enfin, on demande (`stack_area/1`) que les cartons de plus grande base soient empilés sur les cartons de plus petite base.

```
% Bris de symétrie : pour tout ensemble d'objets de même référence K,
% les vecteurs coordonnées {(x_i, y_i, z_i) | i ∈ {1,...,card(K)}} respectent
% l'ordre lexicographique.
lexicographical(ListOfItemsGroups, Dims) -->
  forall(Items in ListOfItemsGroups,
    lexicographic(map(I in Items, map(D in Dims, nth(D, I:origin))))).

pallet_loading_constraints(Items, ItemsReferences, Bin, BinSize, Dims) -->
  domains(Items, BinSize, Dims) and
  containmentAE(Items, [Bin], Dims) and
  non_overlapping(Items, Dims) and
  lexicographical(ItemsReferences, [3, 1, 2])and
  stack_area(Items).
```

La stratégie de recherche

Les cartons sont placés par couches de même référence `load_layers/5` et l'heuristique demande de placer les références dans l'ordre croissant de leur base.

```
h_layer =
  conjunctive(least(nth(1, ItemsSize) * nth(2, ItemsSize))
              for load_layers(_, _, _, ItemsSize, _)).
```

La stratégie de branchement est basée sur un placement par couches : calculer le nombre de couches par référence et placer d'abord les références dont la base est la plus petite.

```
pallet_loading_search(ItemsData, ItemsReferences, Bin, BinSize, Dims) -->
  layers_card(ItemsData, BinSize) and
  search(h_layer,
    forall(ID in [1 .. length(ItemsData)],
      let(ItemData := nth(ID, ItemsData),
          Items := nth(ID, ItemsReferences),
          ItemsCard := ItemData:card,
          ItemsSize := ItemData:size,
          LayersCard := layers_card(ID),
          ItemsPerLayerCard := ItemsCard / LayersCard,
        load_layers(LayersCard, ItemsPerLayerCard, Items, ItemsSize, BinSize)))).
```

La stratégie de branchement détermine d'abord pour tous les cartons leur coordonnée sur l'axe vertical (orthogonal à la pallette) avec la plus petite valeur disponible.

```
import('Cream/lib/search/domain_splitting').

search_strategy(Items, Dimensions, UpperBounds, IntervalLengths) -->
  forall(I in Items, origin(I, 3) = domain_min(origin(I, 3))) and
  interval_dichotomic_split_interleaved(Items, Dimensions,
                                        UpperBounds,
                                        IntervalLengths).
```

Ensuite, il s'agit de résoudre un problème de placement bi-dimensionnel par une recherche par découpage d'intervalles semblable à celle utilisée pout *Optimal Rectangle Packing* (cf. section 9.1)

```
interval_dichotomic_split_interleaved(Items, Dimensions,
                                      UpperBoundsList,
                                      IntervalLengthList) -->
  forall(I in Items,
    forall(D in Dimensions,
      let(UpperBound := nth(D, UpperBoundsList),
          IntervalLength := nth(D, IntervalLengthList),
          OriginID := origin(I, D),
        interval_split(OriginID, 0, UpperBound, IntervalLength) and
        dichotomic_split(OriginID)))).
```

Solutions

Les figures 9.3 et 9.4, issues de `CLPGUI` [FSC04], montrent respectivement une solution d'une instance de *Pallet Loading* impliquant 80 objets d'une même référence (*i.e.*, de même caractéristiques), et une solution d'une instance impliquant 40 objets et deux références.

FIGURE 9.3: Une solution de *Pallet Loading* ; 80 objets polymorphes, 1 référence.

FIGURE 9.4: Une solution de *Pallet Loading* ; 40 objets polymorphes, 2 références.

Conclusion

Le problème 2+1D de chargement de palettes monoréférences a pu être résolu car le jeu de boîtes à placer permet une division en couches, mais ce n'est pas le cas de tous les problèmes de chargement de pallettes [ACC+10].

En effet, certaines instances présentent une grande hétérogénéité en terme de références d'objets à placer ce qui empêche de trouver une structure exploitable par une heuristique et nécessite d'explorer la combinatoire de toutes les rotations. De plus, la contrainte globale `geost` effectue une propagation trop faible dans le contexte de problèmes avec objets polymorphiques. En particulier `geost` n'inclut pas de raisonnement sur les volumes des objets à placer.

9.3 Problème de chargement de container

9.3.1 Définition

Le problème de chargement de container (*Container Loading*) est un problème industriel de placement tri-dimensionnel, non réductible à un problème 2+1D, qui consiste à placer un ensemble de palettes dans un container qui pourra être destiné au transport routier, ferroviaire ou maritime.

Les palettes peuvent subir une rotation d'un quart de tour autour de l'axe orthogonal à la base du container.

De plus, les palettes sont munies d'un attribut de poids et de facteur de gerbabilité dont le produit indique la capacité d'un objet à supporter du poids.

Enfin, les solutions doivent respecter 7 contraintes spécifiques portant sur la stabilité et la répartition du poids localement à une pile et globalement dans le container.

9.3.2 Modèle et évaluation

Imports et données

Une instance de *Container Loading* se découpe en trois modules : les données, la stratégie de recherche et le modèle à proprement parler.

```
import('Cream/examples/packing/container_loading_1.data').
import('Cream/examples/packing/container_loading_1.strategy').
import('Cream/examples/packing/container_loading.model').
```

Les données brutes sont transformées pour produire des enregistrements du type qui suit. Chaque référence d'objet a une certaine taille (réduite si possible), un facteur de gerbabilité, un poids et une cardinalité.

```
% taille originale
one_ref_size = [224, 170, 121].
% taille réduite
one_size = canonical_size(one_ref_size).
% propriétés de la référence "one"
items_one = {ref_size = one_ref_size,
             size = one_size,
             piling = 1,
             weight = 220,
             card = 4}.
(...)
% ensemble des références de l'instance
items_data = [items_one, items_two, ..., items_n].
```

Le but

Avant de résoudre le problème *Container Loading* on essaye de réduire les tailles de l'instance si possible (`reduced_sizes/3`), puis on pose les contraintes et on lance la recherche (`container_loading/3`).

```
% Dims : nombre de dimensions du problème
% BinData : données du contenant
% ItemsData : données des références à placer
? dynamic(
    let(Dims := dimensions,
        BinData := bin_data,
        BinRefSize := BinData:ref_size,
        ItemsData := items_data,
        ItemsRefSize := map(ItemsData in ItemsData, ItemsData:ref_size),
      reduced_sizes(BinRefSize, ItemsRefSize, Dims) and
      container_loading(BinData, ItemsData, Dims))).
```

Les contraintes

Après les domaines des variables, les contraintes génériques correspondant au problème pur de *Bin Packing* sont posées. Elles comprennent la contrainte globale `geost`.

À ces contraintes génériques sont ajoutées un ensemble de contraintes spécifiques au problème container loading dont une grande partie a été décrite dans la section 8.3. Elles assurent que dans toute solution :

- les objets sont stables et ne glissent pas, c'est-à-dire que chaque objet ait un support dans toutes les dimensions (`gravity(Items)` et `blocked(Items)`) ;
- les objets de plus petite hauteur sont placés sous les autres (`stack_height(Items)`) ;
- les objets de plus petite base sont placés sous les autres (`stack_area(Items)`) ;
- les contours des piles sont «lisses» (`stack_alignment(Items)`) ;
- les objets sont supportés sur 100% de la surface de leur base (`stack_support_area(Items, 100)`) ;
- les objets n'ont pas à supporter plus que leur gerbabilité (`stack_weight_sum(Items)`)
- et enfin que le poids des objets soit bien réparti globalement dans la longueur de contenant (`weight_balancing(Items, Bin, 1, 10)`) .

```
import('Cream/lib/pkml/pkml').

container_loading_constraints(Items, Bin, BinSize, Dims) -->
  domains(Items, BinSize, Dims) and

  containmentAE(Items, [Bin], Dims) and
  non_overlapping(Items, Dims) and

  gravity(Items) and
  blocked(Items) and
  stack_height(Items) and
  stack_area(Items) and
  stack_alignment(Items) and
  stack_support_area(Items, 100) and
  stack_weight_sum(Items) and
  weight_balancing(Items, Bin, 1, 10).
```

La stratégie de recherche

Après avoir décider des orientations, la stratégie de branchement se décompose en deux :

1. placer grossièrement tous les objets (`place_item_coarsely/2`) par découpage de leur domaine ;
2. décider précisément du placement (`place_item/1`).

L'heuristique principale (l'heuristique sur les conjonctions), dissociée de la stratégie de branchement, choisit de placer d'abord les objets les moins lourds et les moins capable de supporter (de gerbabilité minimale).

```
h_cont_load =
  conjunctive([
    least(Item:piling * Item:weight) for place_item_coarsely(Item, _),
    least(Item:piling * Item:weight) for place_item(Item),
    least(domain_size(Var)) for dichotomic_split(Var)]).
(...)
search(h_cont_load,
    forall(Item in Items, place_item_coarsely(Item, BinSize)) and
    forall(Item in Items, place_item(Item))).
```

```
place_item_coarsely(Item, BinSize) -->
  interval_split(Item, 2, 0, nth(2, BinSize), size(Item, 2) - 1) and
  interval_split_out_middle(Item, 1, 0, nth(1, BinSize), size(Item, 1) / 2) and
  interval_split_reverse(Item, 3, 0, nth(3, BinSize), size(Item, 3) - 1).

place_item(Item) -->
  forall(D in dimensions, dichotomic_split(origin(Item, D))).
```

Puisque l'heuristique choisit de placer d'abord les objets les moins lourds et les moins capable de supporter, il s'agit de les placer de haut en bas, grâce à la stratégie suivante appliquée sur la dimension de l'axe vertical.

`interval_split_reverse/5` : découpe le domaine de la coordonnée de l'objet *item* dans la dimension *dim* par intervalles de longueur *len*, de la borne supérieure *max* vers la borne inférieure *min*, et en prenant en compte la taille de l'objet *s*.

```
interval_reverse_predicate(Rec, Ctn) -->
  let(Var := Rec:var,
      Len := Rec:len,
      Cut := domain_max(Var) - Len,
    Var >= Cut or (Var < Cut and Ctn)).

interval_split_reverse(Item, Dim, Min, Max, Len) -->
  foldr(C in [1 .. ((Max - size(Item, Dim)) - Min) / Len],
        interval_reverse_predicate, true,
        {var = end(Item, Dim), len = Len}).
```

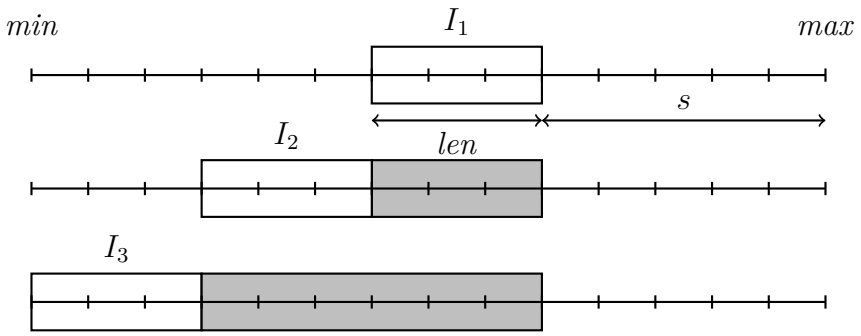


FIGURE 9.5: Découpage d'un domaine $[min,\ max]$ par intervalles I_i de longueur *len* de la borne supérieure vers la borne inférieure.

Pour ce qui est de la première dimension, la dimension des plus grandes longueurs, la stratégie consiste à ventiler les objets aux extrémités et finir le placement au centre du contenant, grâce à la règle suivante.

`interval_split_out_middle/5` : Le découpage d'un domaine, initialement $[min,\ max]$, d'un objet de taille s des bornes vers le milieu et par intervalles de longueur len comprend $n = ((max - s) - min)/(2 * len)$ coupes. Comme l'illustre la figure 9.6, à chaque étape $c \in \{0, \ldots, n-1\}$ le domaine est restreint à l'intervalle $[min + (c-1) * len,\ (min + c * len) - s]$.

```
interval_split_out_middle(Item, Dim, Min, Max, Len) -->
  exists(I in [0 .. ((Max - size(Item, Dim))- Min) / (2 * Len))],
    let(LB := I * Len,
        UB := (I + 1) * Len,
      domain(origin(Item, Dim), Min + LB, Min + UB) or
      domain(end(Item, Dim), Max - UB, Max - LB))).
```

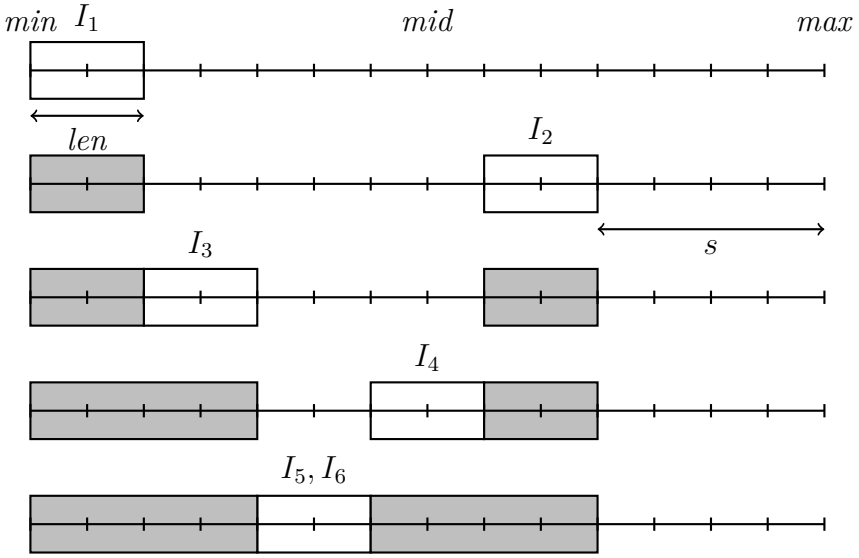


FIGURE 9.6: Découpage d'un domaine $[min,\ max]$ par intervalles I_i de longueur l des bornes du domaine vers milieu mid, en considérant la taille d'objet s.

Solution

La figure 9.7, montre une solution d'une instance de *Container Loading*.

L'exécution du programme généré `SICStus-Prolog`, sur une machine Linux dotée de procésseurs Intel Core2 Quad CPU cadencé à 2.83GHz, résout le problème en 8 secondes et 489 *backtracks*.

Conclusion

Le problème de placement en 3 dimensions avec rotations et contraintes de stabilité et de poids a été résolu en `Cream` dans un temps raisonnable grâce à la combinaison des contraintes réifiées à la contrainte globale `geost`, qui a pris en charge les contraintes

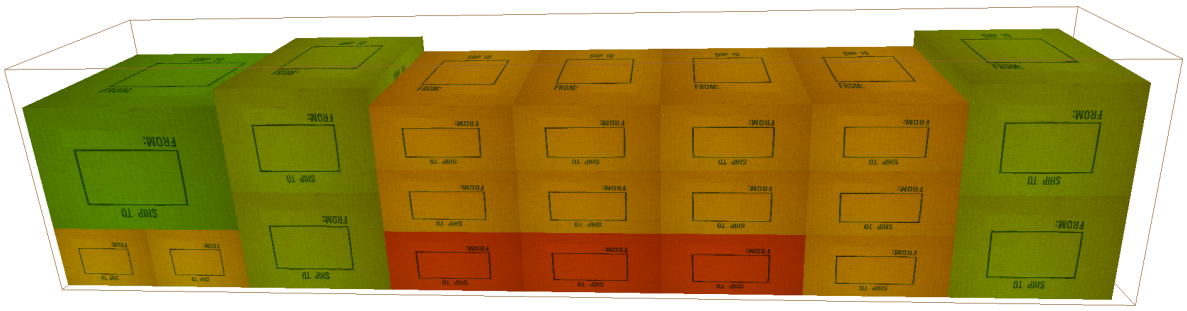


FIGURE 9.7: Une solution de *Container Loading* ; 19 objets polymorphes, 5 références. Plus un objet est lourd et peut supporter de poids, plus il est rouge foncé. Inversement, moins un objet est lourd et peut supporter de poids, plus il est vert clair.

de non-enchevêtrement, et à une heuristique de recherche performante. Il s'avère que l'heuristique présentée adopte une stratégie similaire à celle utilisée par les ingénieurs spécialisés en colisage et qu'elle s'exprime en `Cream` dans des termes intuitifs pour ces derniers.

Parmi les partenaires du projet Net-WMS dont est issu ce problème de chargement de palettes, nous avons été les seuls à résoudre ce problème en prenant en compte toutes les contraintes. Cela a été possible d'une part grâce à la bibliothèque `PKML` et d'autre part grâce à la rapidité d'expérimentation des heuristiques exprimées par *pattern-matching* sur les têtes de règles qu'offre `Cream`.

Cependant, la stratégie présentée n'est pas suffisamment robuste pour résoudre toutes les instances de cette classe de problème. Pour ces problèmes avec rotations en effet, deux difficultés se combinent : d'une part l'algorithme de filtrage de la contrainte `geost` n'est actuellement pas suffisant pour propager correctement les contraintes d'occupation volumique en présence de rotations, et d'autre part la stratégie de placement nécessite d'effectuer les choix d'orientation des objets trop tôt et n'est pas robuste pour cette raison.

Chapitre 10

Conclusion

`Cream` est un langage de modélisation à base de règles pour la programmation par contraintes. Il a été conçu pour permettre à l'homme de métier profane en PPC et qui n'est pas nécessairement programmeur d'exprimer ses connaissances et des exigences industrielles sur des problèmes d'optimisation combinatoire grâce à la définition de règles. Les règles `Cream` sont déclaratives et compositionnelles, ce qui autorise l'énoncé d'une connaissance complexe par fragments.

Un point original de `Cream` réside dans le fait que non seulement les arbres mais aussi les heuristiques d'ordonnancement de la recherche peuvent être spécifiés déclarativement. Dans le langage, une stratégie de branchement est définie par une formule logique du modèle. Les heuristiques qui guident le parcours de l'arbre de recherche induit sont déclaratives, c'est-à-dire dissociées de la stratégie de branchement, compositionnelles et exploitent directement la structure du modèle. Elles sont définies comme des ordres de préférence sur les sous-formules conjonctives et disjonctives, par filtrage des têtes des déclarations de règle. Par opposition aux autres langages de modélisation dans lesquels les stratégies de recherche doivent encore être programmées en s'appuyant sur des listes ou des tableaux et des structures de contrôles, comme dans OPL et Comet par exemple.

La transformation des modèles `Cream` en programmes de contraintes sur domaines finis a été décrite formellement pour deux schémas de compilation complémentaires. Le schéma de compilation fondamental, dit statique, qui procède par expansion des règles du modèle comme dans Zinc par exemple, est formalisé par un système de réécriture de termes défini inductivement et muni d'un mécanisme d'évaluation partielle. Ce schéma produit des programmes de contraintes « plats » optimisés. La confluence et la correction de la transformation vis-à-vis d'une sémantique déclarative sont prouvées, ainsi qu'une borne de complexité sur la taille des programmes générés. De plus, la terminaison de la compilation et la terminaison de l'exécution des modèles `Cream` sont garanties. L'obtention de tels résultats reflète la simplicité des choix de conception de `Cream`, comme par exemple l'absence de récursion remplacée par des constructeurs et itérateurs sur listes généraux.

Cependant, cette borne de complexité montre une explosion potentielle de la taille des contraintes générées. Dans de tels cas, le schéma de compilation dynamique par génération de procédures, similaire à celui d'OPL et de Comet, peut être appliqué.

Dans `Cream`, ce schéma produit des programmes de contraintes structurés incluant de définitions (récursives) de clauses telles qu'un programmeur les auraient écrites dans un langage de programmation logique par contraintes. Cette stratégie engendre un facteur constant de surcoût à l'exécution.

Enfin, l'expressivité du langage et l'efficacité des programmes générés ont été évaluées avec succès sur un problème académique et sur des problèmes issus de la logistique dans l'industrie automobile. Les modèles considérés reposent sur une bibliothèque dédiée à la modélisation des problèmes de placement multi-dimensionnel. La bibliothèque inclut les définitions nécessaires à l'expression de problèmes de placement purs de dimension quelconque. Elle offre aussi des définitions relatives à des exigences sur le poids, la surface, le volume, la stabilité, et autres règles de placement spécifiques. L'évaluation du modèle du problème académique *Optimal Rectangle Packing* permet de constater que les stratégies de recherche dynamiques efficaces s'expriment de façon concise et déclarative. L'évaluation du langage est d'autant plus riche qu'elle a compris des problèmes de placement réels non-purs tri-dimensionnels et impliquant des objets polymorphiques. Elle a montré qu'un langage de modélisation à base de règles donne les moyens d'exprimer et de composer les contraintes spécifiques à la logistique dans l'industrie automobile. De plus, les heuristiques déclaratives ont permis d'expérimenter rapidement et sans peine de nombreuses idées de stratégies jusqu'à résoudre effectivement les problèmes. Cependant, il est nécessaire d'améliorer les stratégies pour les problèmes de placement avec rotations.

Perspectives

Des travaux complémentaires doivent être accomplis sur le système de module du langage dans la perspective de développer des bibliothèques de modèles réutilisables dans une hiérarchie de modèles et pour des applications spécifiques.

Un système de type pour `Cream` est en cours de développement par Thierry Martinez [MMF10] afin d'aider encore à l'écriture des modèles. Pour le moment la compilation ne garantit pas le bon typage des programmes générés. En particulier, une définition d'objet est implicitement entendue comme un constructeur ou une fonction, or l'appel à une telle définition directement sous un opérateur logique n'est pas une expression valide. D'un autre côté, les définitions de règles sont comprises comme des définitions de prédicats et donc de type booléen. Il serait alors confortable qu'un système de type impose cette sémantique.

Par ailleurs, l'approche de spécification d'heuristiques d'ordre par filtrage proposée devrait être applicable à d'autre langages de modélisation qui permettent des définitions, tel que ZINC [dlBMRW06] par exemple. Il serait également intéressant de tenter d'adapter l'expression du contrôle par filtrage sur les membres gauches de règles au langage Prolog pour lequel on recourt toujours à l'écriture d'un méta-interpréteur en l'absence d'un langage d'expression du contrôle.

De plus, l'aspect déclaratif du langage de définition d'heuristique de `Cream` en fait un bon candidat pour un apprentissage automatique. Les critères dynamiques dépendraient alors de profils d'exécution.

Enfin, une extension naturelle du langage serait d'intégrer la spécification de procédures de recherche dans le langage, actuellement limitées à la recherche en profondeur

d'abord et par séparation et évaluation.

Bibliographie

[ACC+10] Abder Aggoun, Mats Carlsson, Mickaël Collardey, Francesca Di Lucchio, François Fages, Philippe Gravez, and Renaud Deligny. Industrial prototypes. deliverable D9.1, FP6 Strep project Net-WMS, February 2010.

[All91] J. Allen. Time and time again : The many ways to represent time. *International Journal of Intelligent System*, 6(4), 1991.

[AS99] Krzysztof R. Apt and Andrea Schaerf. The alma project, or how first-order logic can help us in imperative programming. In *Correct System Design*, pages 89–113, 1999.

[AW07] Krzysztof R. Apt and Mark Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, New York, NY, USA, 2007.

[BCP+07] N. Beldiceanu, M. Carlsson, E. Poder, R. Sadek, and C. Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In C. Bessière, editor, *Proc. CP'2007*, volume 4741 of *LNCS*, pages 180–194. Springer, 2007. Also available as SICS Technical Report T2007 :08, `http://www.sics.se/libindex.html`.

[BPN01] Ph. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling : Applying Constraint Programming to Scheduling Problems*. Kluwer Academic Publishers, 2001.

[C+07] M. Carlsson et al. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, release 4 edition, 2007. ISBN 91-630-3648-7.

[CBM08] Mats Carlsson, Nicolas Beldiceanu, and Julien Martin. A geometric constraint over k-dimensional objects and shapes subject to business rules. In Peter J. Stuckey, editor, *Proceedings of CP'08*, volume 5202 of *LNCS*, pages 220–234. Springer, 2008.

[CD85] H. Carpenter and W. Dowsland. Practical consideration of the pallet loading problem. *Journal of the Operations Research Society*, 36 :489–497, 1985.

[CJCM08] François Clautiaux, Antoine Jouglet, Jacques Carlier, and Aziz Moukrim. A new constraint programming approach for the orthogonal packing problem. *Comput. Oper. Res.*, 35(3) :944–959, 2008.

[CKPR72] A. Colmerauer, Henry Kanoui, Robert Pasero, and Philippe Roussel. Un système de communication en français, rapport préliminaire de fin de contrat iria, groupe intelligence artificielle. Technical report, Technical Report, Faculté des Sciences de Luminy, Université Aix-Marseille II, 1972.

[Col84] A. Colmerauer. Equations and inequations on finite and infinite trees. In *Proc. of the International conference on fifth generation computer systems FGCS'84*, pages 85–99. ICOT, 1984.

[Col87] Alain Colmerauer. Opening the prolog iii universe. *BYTE*, 12(9) :177–182, 1987.

[Col96] A. Colmerauer. Specification of Prolog IV. Technical report, LIM Technical Report, 1996.

[DC01] Daniel Diaz and Philipe Codognet. Design and implementation of the GNU Prolog system. *Journal of Functional and Logic Programming*, 6, October 2001.

[Der79] Nachum Dershowitz. Orderings for term-rewriting systems. In *SFCS '79 : Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, pages 123–131. IEEE Computer Society, 1979.

[dlBMRW06] Maria Garcia de la Banda, Kim Marriott, Reza Rafeh, and Mark Wallace. The modelling language Zinc. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming* CP*'06)*, pages 700–705. Springer-Verlag, 2006.

[DP85] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of A*. *J. ACM*, 32(3) :505–536, July 1985.

[FAB+07] François Fages, Abder Aggoun, Nicolas Beldiceanu, Mats Carlsson, Filipe Carvalho, Philippe Gravez, András Kovács, and Julien Martin. State-of-the-art of enabling technologies for packing and planning in future wms. deliverable D3.1, FP6 Strep project Net-WMS, 2007.

[FHJ+08] Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martinez-Hernandez, and Ian Miguel. Essence : A constraint language for specifying combinatorial problems. *Constraints*, 13 :268–306, 2008.

[FM08] François Fages and Julien Martin. From rules to constraint programs with the Rules2CP modelling language. INRIA Research Report RR-6495, Institut National de Recherche en Informatique, April 2008.

[FM09] François Fages and Julien Martin. From rules to constraint programs with the Rules2CP modelling language. In *Recent Advances in Constraints, Revised Selected Papers of the 13th Annual ERCIM International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP'08*, volume 5655 of *Lecture Notes in Artificial Intelligence*, pages 66–83. Springer-Verlag, 2009.

[For82] Charles Forgy. Rete : A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligences*, 19(1) :17–37, 1982.

[FSC04] François Fages, Sylvain Soliman, and Rémi Coolen. CLPGUI : a generic graphical user interface for constraint logic programming. *Journal of Constraints, Special Issue on User-Interaction in Constraint Satisfaction*, 9(4) :241–262, October 2004.

[GJM06] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion : A fast scalable constraint solver. In *Proceedings of the 17th European Conference on Artificial Intelligence, ECAI*, pages 98–102, 2006.

[HF06] Rémy Haemmerlé and François Fages. Modules for Prolog revisited. In *Proceedings of International Conference on Logic Programming ICLP 2006*, number 4079 in Lecture Notes in Computer Science, pages 41–55. Springer-Verlag, 2006.

[HF07] Rémy Haemmerlé and François Fages. Abstract critical pairs and confluence of arbitrary binary relations. In *Proceedings of th 18th International Conference on Rewriting Techniques and Applications, RTA'07*, number 4533 in Lecture Notes in Computer Science. Springer-Verlag, 2007.

[HG95] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *IJCAI'95 : Proceedings of the 14th international joint conference on Artificial intelligence*, pages 607–613, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[Hof92] Dieter Hofbauer. Termination proofs by multiset path orderings imply primitive recursive derivation lengths. *Theoretical Computer Science*, 105(1) :129–140, 1992.

[HPP00] Pascal Van Hentenryck, Laurent Perron, and Jean-Francois Puget. Search and strategies in opl. *ACM Transactions on Compututational Logic*, 1(2) :285–320, 2000.

[IBM] IBM. *IBM ILOG JRules.* http ://www.ibm.com/developerworks/websphere/zones/brms/.

[ILO] ILOG. *ILOG Solver.* http ://www.ilog.com/products/solver/.

[JBo] JBoss. *JBoss Drools.* http ://www.jboss.org/drools/documentation.html.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming : a survey. *Journal of Logic Programming*, 19/20 :503–581, May 1994.

[Jun98] Ulrich Junker. Constrained-based problem decomposition for a key configuration problem. In *CP*, pages 265–279, 1998.

[Kor85] Richard E. Korf. Depth-first iterative-deepening : an optimal admissible tree search. *Artif. Intell.*, 27(1) :97–109, 1985.

[Kor96] Richard E. Korf. Improved limited discrepancy search. In *AAAI/IAAI, Vol. 1*, pages 286–291, 1996.

[Kor04] Richard E. Korf. Optimal rectangle packing : New results. In *ICAPS*, pages 142–149, 2004.

[Kow74] R. Kowalski. Predicate logic as programming language. In *IFIP Congress*, pages 569–574, 1974.

[Lab00] F. Laburthe. CHOCO : Implementing a CP kernel. In *Proceedings of Techniques for Implementing Constraint Programming Systems (TRICS)*, pages 71–85, 2000.

[LD60] A. H. Land and A. G Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3) :497–520, 1960.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1) :99–118, 1977.

[MF85] Alan K. Mackworth and Eugene C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artif. Intell.*, 25(1) :65–74, 1985.

[MH05] Laurent Michel and Pascal Van Hentenryck. The comet programming language and system. In *CP*, pages 881–881, 2005.

[MMF09] Julien Martin, Thierry Martinez, and François Fages. On the specification of search tree heuristics by pattern-matching in a rule-based modelling language. In *Proceedings of the Eighth International Workshop on Constraint Modelling and Reformulation, associated to CP'09*, pages 73–86, 2009.

[MMF10] Julien Martin, Thierry Martinez, and François Fages. Static expansion vs procedural code generation in rule-based modeling languages. In preparation, 2010.

[NSB+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc : Towards a standard CP modelling language. In *CP*, pages 529–543, 2007.

[RCC92] D.A. Randell, Z. Cui, and A.G. Cohn. A spatial logic based on regions and connection. In B. Nebel, C. Rich, and W. R. Swartout, editors, *Proc. of 2nd International Conference on Knowledge Representation and reasoning KR'92*, pages 165–176. Morgan Kaufmann, 1992.

[RdlBMW07] Reza Rafeh, Maria Garcia de la Banda, Kim Marriott, and Mark Wallace. From Zinc to design model. In *Proceedings of PADL'07*, pages 215–229. Springer-Verlag, 2007.

[RMdlB+08] Reza Rafeh, Kim Marriott, Maria Garcia de la Banda, Nicholas Nethercote, and Mark Wallace. Adding search to zinc. In *CP*, pages 624–629, 2008.

[Ros73] B.K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, 20 :160–187, 1973.

[SLM] C. Schulte, Lagerkvist, and G. M., Tack. *Gecode.* http ://www.gecode.org/.

[SO08] Helmut Simonis and Barry O'Sullivan. Using global constraints for rectangle packing. In *Proceedings of the first Workshop on Bin Packing and Placement Constraints BPPC'08, associated to CPAIOR'08*, May 2008.

[Sot09] Ricardo Soto. *Langages et transformation de modèles en programmation par contraintes.* PhD thesis, Université de Nantes, 2009.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2) :146–160, 1972.

[Ter03] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 2003.

[Van89] Pascal Van Hentenryck. *Constraint satisfaction in Logic Programming.* MIT Press, 1989.

[Van99] Pascal Van Hentenryck. *The OPL Optimization programming Language.* MIT Press, 1999.

[vdKFLS10] Roman van der Krogt, Jacob Feldman, James Little, and David Stynes. An integrated business rules and constraints approach to data centre capacity management. In *Proceedings of the 16th Conference on Principles and Practice of Constraint Programming CP 2010*, Saint-Andrews, Scotland, 2010.

[VG06] Vincent Vidal and Hector Geffner. Branching and pruning : An optimal temporal pocl planner based on constraint programming. *Artificial Intelligence*, 170(3) :298–335, 2006.

[VHDT92] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artif. Intell.*, 57(2-3) :291–321, 1992.

[VKK91] Nageshwara Rao Vempaty, Vipin Kumar, and Richard E. Korf. Depth-first versus best-first search. In *AAAI*, pages 434–440, 1991.

[Wal96] Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1/2) :139–168, 1996.

[ZK93] Weixiong Zhang and Richard E. Korf. Depth-first vs. best-first search : New results. In *AAAI*, pages 769–775, 1993.