Approche, en termes de squelettes de preuve, de la sémantique et du diagnostic déclaratif d'erreur des programmes logiques avec contraintes

Thèse

présentée pour l'obtention du titre de

Docteur de l'Université d'Orléans

spécialité Informatique

par

Alexandre Tessier

soutenue le 6 janvier 1997

Composition du jury présidé par François Fages

Rapporteurs :	François Fages	Chargé de recherche - CNRS, ENS Paris
	Jean-François Pique	Professeur - Université de Marseille
Examinateurs :	Pierre Deransart	Directeur de recherche - INRIA Rocquencourt
	Gérard Ferrand	Professeur - Université d'Orléans
	François Le Berre	Professeur - Université d'Orléans
	Jan Małuszyński	Professeur - Linköping University

Thèse réalisée au Laboratoire d'Informatique Fondamentale d'Orléans sous la direction de Gérard Ferrand.

Résumé

Cette thèse propose une reformulation complète de la sémantique des programmes logiques avec contraintes dans la lignée de la vision grammaticale de la programmation logique [P. Deransart et J. Maluszynski 1993]. La sémantique généralise les sémantiques classiques [J. Jaffar et J-L. Lassez 1987] basées sur une interprétation ou une théorie pour la sémantique du langage des contraintes. De plus, elle tient compte de l'incomplétude des solveurs de contraintes. Les résultats connus sont retrouvés. Cette reformulation est particulièrement bien adaptée pour étudier le diagnostic déclaratif d'erreur [E. Y. Shapiro 1982] des programmes logiques avec contraintes. Diverses notions d'erreurs sont définies et des algorithmes de diagnostics sont proposés. Ils sont comparés avec certaines techniques élaborées pour la programmation logique pure.

Mots Clés

Programmation logique avec contraintes, sémantique opérationnelle, sémantique déclarative, squelettes, arbre SLD, définitions inductives, arbres de preuve, diagnostic déclaratif d'erreur, correction partielle, débogage, validation.

Abstract

This thesis proposes a full reformulation of the Constraint Logic Program semantics following the Grammatical View of Logic Programming [P. Deransart et J. Maluszynski 1993]. Our semantics genralizes classical semantics [J. Jaffar et J-L. Lassez 1987] based on an interpretation or a theory for the contraint language semantics. Moreover, it takes into account incompleteness of the contraint solvers. Known results are revisited. This reformulation is particularly adapted to study Declarative Error Diagnosis [E. Y. Shapiro 1982] of Constraint Logic Programs. Several notions of errors are defined and diagnosis algorithms are proposed. They are compared with some technics developped for pure Logic Programming.

Key Words

Constraint logic programming, operational semantics, declarative semantics, skeleton, SLD tree, inductive definitions, proof trees, declarative error diagnosis, partial correctness, debugging, validation.

Remerciements

Je remercie François Fages, Chargé de Recherche à l'ENS de Paris, et Jean-françois Pique, Professeur à l'Université de la Méditerranée, qui ont accepté de rapporter cette thèse.

Je remercie Jan Małuszyński, Professeur à L'Université de Linköping, d'avoir accepté de participer au jury. Ses travaux antérieurs sont à l'origine de nouveaux résultats présentés dans cette thèse.

Je suis très sensible à l'intérêt que Pierre Deransart, Directeur de Recherche à l'INRIA, a porté à ce travail. Cette thèse est au centre du projet ESPRIT DiSCiPL dont il est l'instigateur. Ses travaux avec Gérard Ferrand et Jan Małuszyński ont largement contribué à mes recherches. Enfin, je le remercie pour avoir accepté de faire partie du jury.

J'exprime ma plus profonde gratitude à Gérard Ferrand, Professeur à l'Université d'Orléans, qui a dirigé cette thèse. Je tiens à le remercier pour sa grande rigueur scientifique et ses conseils éclairés qui m'ont fourni une aide décisive. Je lui suis particulièrement reconnaissant pour sa disponibilité malgré les lourdes tâches administratives de directeur du Laboratoire d'Informatique Fondamentale d'Orléans qu'il assure. Nos discussions ont été déterminantes pour ce travail.

Je tiens à remercier tout particulièrement François Le Berre, Professeur à l'Université d'Orléans, qui a très largement contribué au travail de reformulation de la sémantique des programmes logiques avec contraintes.

Je remercie Michel Bergère pour la relecture du manuscrit dans l'étape finale.

Merci à tous les collègues du Laboratoire d'Informatique Fondamental d'Orléans pour l'ambiance sympathique dans laquelle j'ai travaillé durant ces années. Merci à Frédéric Benhamou, pour m'avoir fait profité de son expérience grâce aux nombreux échanges lors des pauses café et durant les nuits passées à travailler au laboratoire.

Table des matières

Table des figures

Table des définitions

Table des exemples

Introduction

On n'écrit pas un programme seulement pour qu'il produise des résultats. On souhaite de plus que ces résultats soient justes. Il arrive qu'un calcul fournisse un résultat *non attendu*. Il s'agit par conséquent d'un calcul qui termine et le résultat non attendu est *symptôme* d'une erreur dans le programme. L'absence de symptôme est la *correction partielle* du programme. Cette correction n'est que partielle car elle assure que les résultats fournis par les calculs finis sont justes. Le problème de non terminaison n'est pas abordé ici.

Pour juger si un résultat est juste, on doit disposer d'une sémantique attendue pour le programme. Le concept d'*oracle* formalise cette notion de jugement : l'oracle permet d'interpréter un résultat, même complexe, pour indiquer s'il est attendu relativement à la sémantique attendue.

On peut distinguer deux activités duales :

1. la preuve de correction partielle ;
2. le diagnostic d'erreur.

La notion commune à ces deux activités est l'*erreur*. Une erreur est une partie de programme cause de l'apparition d'un symptôme à la fin d'un calcul.

La preuve de correction partielle cherche à montrer l'absence d'erreur alors que le diagnostic est la recherche d'une erreur quand un symptôme est observé.

Le théorème fondamental nécessaire pour le diagnostic d'erreur est : s'il existe un symptôme alors il existe une erreur. Sa contraposé (pas d'erreur $\Rightarrow$ pas de symptôme) est le cœur des méthodes de preuve de correction partielle. Bien entendu, ce résultat n'est obtenu que si les notions de symptôme et d'erreur ont été correctement définies.

En général la réciproque du théorème est fausse (i.e. il existe une erreur $\not\Rightarrow$ il existe un symptôme). En effet, une erreur peut ne pas se manifester par un symptôme : elle n'intervient dans aucun calcul fini. Malgré tout, l'absence d'erreur est plus intéressante que l'absence de symptôme : prenons un programme erroné sans symptôme, une extension du programme peut faire surgir des symptômes dûs à des erreurs du programme d'origine. D'où l'intérêt des méthodes de preuve de correction partielle quand elles peuvent être mises en œuvre.

Un algorithme de diagnostic cherche à localiser une erreur associée à une notion de symptôme minimal (intuitivement le résultat non attendu d'un calcul minimal).

À cette fin, l'algorithme interroge l'oracle sur des résultats de calculs pour déterminer s'ils sont symptômes ou non (i.e. s'ils sont attendus). L'oracle a donc la compétence d'indiquer si pour tout calcul fini le résultat est attendu.

Ceci pose le *problème de présentation* : le résultat doit être présenté de manière à ce que l'oracle (par exemple le programmeur) puisse juger s'il est attendu. Mais ce problème dépasse le cadre du diagnostic. En effet, les systèmes doivent eux mêmes présenter les résultats de manière exploitable, c'est-à-dire compréhensible.

Dans le cadre du diagnostic d'erreur, nous considérons donc en permanence deux sémantiques pour un programme : sa sémantique *de fait* (ce qui est calculé) et sa sémantique *attendue* (ce qui devrait être calculé).

Notons que cette distinction est directement lié à l'activité de programmation. Comme nous l'avons précisé, concevoir un programme ne se résume pas à écrire un programme qui s'exécute, on souhaite écrire un programme juste. Par conséquent avant même l'écriture d'un programme sa sémantique attendue doit être connue du programmeur. Bien programmer suppose savoir distinguer la sémantique de fait de la sémantique attendue.

Pour les langages déclaratifs, l'accent est mis sur l'existence d'une *sémantique déclarative*, c'est-à-dire une sémantique indépendante du modèle d'exécution.

Il est naturel que la sémantique attendue soit de même nature.

C'est le concept de *diagnostic déclaratif d'erreur*, déclaratif signifiant que l'oracle juge un résultat indépendamment du comportement opérationnel du système qui exécute le programme. Pour un langage déclaratif il est essentiel de considérer une notion d'erreur déclarative.

Le cadre de cette thèse est celui de la Programmation Logique avec Contraintes (PLC). Un des avantages des langages de PLC est leur sémantique déclarative dont l'utilisation est aussi confirmé par des applications industrielles récentes.

Il serait incohérent pour ces langages de n'utiliser que des outils de mise au point de bas niveau : il est important de se doter d'outils qui ne font appel qu'à une connaissance déclarative des propriétés attendues du programme. De plus, le comportement opérationnel des systèmes de PLC est relativement complexe.

Les langages de PLC sont vus comme des langages du paradigme de programmation relationnelle (comme il existe la programmation impérative ou la programmation fonctionnelle).

Un système de PLC est paramétré par une structure qui fournit la sémantique des contraintes (i.e. des relations pré-définis). Par exemple une interprétation du langage des contraintes.

Un programme est la définition de nouvelles relations qui étendent les relations pré-définies. Sa sémantique attendue pourra par exemple être définie comme une interprétation du langage du programme. Cette interprétation prolonge l'interprétation du langage des contraintes.

En fait, le programme cherche à axiomatiser l'interprétation attendue, c'est-à-dire qu'on veut que l'interprétation attendue soit un modèle du programme (en réalité un modèle du complété du programme).

La particularité du paradigme de programmation relationnelle est l'indéterminisme.

On peut considérer deux niveaux de résultats calculés pour un but $\leftarrow a$:

1. *Le calcul d'une réponse*, le sens donné à une (contrainte) réponse C est $a \leftarrow C$ (le calcul de la réponse est fini).

   Une réponse est un symptôme si $a \leftarrow C$ est faux dans l'interprétation attendue (c'est à dire qu'il existe une valuation v qui satisfait C dans l'interprétation du langage des contraintes mais ne satisfait pas a dans l'interprétation attendue) ;

2. *Le calcul de l'ensemble des réponses* (le calcul de cet ensemble est fini), le sens donné à l'ensemble fini de réponse $\{C_1, \ldots, C_n\}$ est $a \leftrightarrow C_1 \vee \cdots \vee C_n$.

   L'implication $a \leftarrow C_1 \vee \cdots \vee C_n$ est vraie si pour tout $i = 1 \ldots, n$ l'implication $a \leftarrow C_i$ est vraie. C'est un cas particulier du point précédent (quand le calcul de l'ensemble est fini). On s'intéresse donc seulement à l'implication $a \rightarrow C_1 \vee \cdots \vee C_n$ pour ce type de calcul (le cas $n = 0$ correspond à ce qu'on appelle couramment l'échec fini).

L'ensemble des réponses est un symptôme si $a \rightarrow C_1 \vee \cdots \vee C_n$ est faux dans l'interprétation attendue (c'est à dire s'il existe une valuation v qui satisfait a dans l'interprétation attendue mais qui ne satisfait aucun des C_i dans l'interprétation du langage des contraintes).

Ces deux niveaux de réponses correspondent à deux sémantiques *de fait* du programme : le premier niveau détermine la *sémantique positive* et le second niveau la *sémantique négative* du programme.

On distingue deux niveaux de réponse : les réponses positives (appelées simplement réponses) et les réponses négatives (appelées $\vee$-réponses).

Nous avons motivé le travail dans le cadre d'une interprétation attendue, mais de manière plus générale, on diagnostique des erreurs à partir de propriétés attendues qui ne constituent pas nécessairement une spécification complète du programme et qui ne sont pas nécessairement exprimées par une interprétation (partielle) attendue.

Un programme logique avec contraintes peut être vu comme un programme logique avec des relations pré-définies et pré-interprétées dans un domaine particulier (différent du domaine de Herbrand), mais c'est un idéal que les systèmes n'atteignent pas toujours, les *solveurs de contraintes* sont souvent incomplets.

D'où la nécessité d'une reformulation des deux niveaux de calcul dans un cadre qui sépare l'aspect programmation de l'aspect satisfaction de contraintes et qui formule une sémantique déclarative des programmes.

Ce cadre est adapté à l'étude du diagnostic déclaratif d'erreur. Il est basé sur la notion fondamentale et centrale de *squelette*. Le squelette regroupe dans une structure arborescente toute l'information déclarative (utile pour le diagnostic) issue d'un calcul. C'est à partir des squelettes que sont parfaitement définis les deux niveaux de réponses associés aux deux niveaux de calculs : le squelette représente un état de calcul.

Le chapitre 1 rappelle les définitions utiles dans cette thèse. Il est constitué de trois sections : les rappels sur les arbres, sur les définitions inductives et sur les langages de programmation logique avec contraintes.

Le chapitre 2 démarre la reformulation de la sémantique des programmes. Il présente la sémantique opérationnelle. On y trouve les définitions de squelettes, état de calcul, dérivation SLD, arbre SLD, réponse et $\vee$-réponse. La notion habituelle de test de satisfiabilité d'une contrainte est abstraite par un *critère de rejet*, ainsi on rend compte de l'incomplétude éventuelle du solveur de contraintes.

Le chapitre 3 présente les types de sémantiques déclaratives habituellement étudiés : la sémantique déclarative selon une interprétation du langage des contraintes et la sémantique déclarative selon une théorie dans le langage des contraintes. Les résultats connus sont retrouvés dans notre cadre. Cette partie explique clairement la terminologie "sémantique positive" et "sémantique négative". Pour continuer à donner une sémantique de nature déclarative quand le solveur de contraintes est incomplet, on étend le critère de rejet à une relation de couverture. La notion de couverture d'une contrainte par un ensemble (éventuellement infini) de contraintes (réponses) abstrait la notion de couverture dans une interprétation ou une théorie.

Le chapitre 4 termine cette thèse par l'étude du diagnostic déclaratif d'erreur. Il met complètement à profit la reformulation des deux chapitres précédents. On présente un premier schéma général pour le diagnostic basé sur les définitions inductives. Dans ce cadre, on rappelle rapidement les travaux précédents sur le diagnostic déclaratif d'erreur. La section 4.3 fournit un nouveau schéma basé sur une relation bien fondée, pour l'étude des algorithmes de diagnostic d'incorrection, fondé sur la constatation que ces algorithmes ne cherchent que des éléments minimaux parmi les

symptômes. Dans ce cadre nous étudions le diagnostic d'incorrection positive et le diagnostic d'incorrection négative. Le premier correspond au problème de réponse fausse et considère le niveau de calcul "positif" (calcul d'une réponse). Le second correspond au problème de réponse manquante et considère le niveau de calcul "négatif" (calcul de l'ensemble des réponses). Le nouveau schéma montre tout son intérêt pour l'étude des symptômes liés au second niveau de calcul : il permet de comprendre des algorithmes qui étaient connus dans le cadre plus simple de la programmation logique, mais qui n'étaient pas expliqués formellement. En fait ces algorithmes sont des algorithmes de recherche d'incorrection (non des optimisations d'algorithmes de recherche d'insuffisance, une autre notion de diagnostic pour les réponses manquantes) sur une structure complexe qui devient simple grâce au second schéma et à notre définition des arbres SLD. De plus on donne une famille beaucoup plus large d'algorithmes (que celle connue pour la programmation logique) qui permet des optimisations auparavant impossibles. Enfin, on esquisse une autre méthode, plus classique, pour le problème des réponses manquantes : le diagnostic d'insuffisance.

> "Above all, the book conveys the excitement of using Prolog — the thrill of declarative programming. As the authors put it "declarative programming clears the mind". Declarative programming enables one to concentrate on the essentials of a problem, without getting bogged down in too much operational detail. Programming should be an intellectually rewarding activity. Prolog helps to make it so. Prolog is indeed, as the authors contend, a tool for thinking."
>
> Foreword by David H. D. Warren, The Art of Prolog [90], page xi.

C'est encore plus vrai en programmation logique avec contraintes !

Chapitre 1

Préliminaires

Nous présentons dans ce chapitre les préliminaires utiles à cette thèse.

Nous commençons par définir les arbres dans la section 1.1. La notion d'arbre est au cœur de notre travail. Notre définition est équivalente à celle que l'on trouve habituellement dans la littérature.

Dans la section 1.2 nous rappelons les principes généraux des définitions inductives. Les définitions inductives seront largement utilisées dans les chapitres 3 et 4.

Enfin, dans la section 1.3 nous présentons la syntaxe que nous utiliserons pour les programmes logiques avec contraintes et nous introduisons quelques notations et la terminologie correspondante.

1.1 Arbres

Définition 1.1.1 Arbre
Un arbre *est une paire* (E, R)*, où* E *est l'ensemble des* nœuds*, appelé* domaine de l'arbre*, et* R *est une relation binaire sur* E*, appelée* relation de parenté*, tels qu'il existe un unique élément* $r \in E$*, appelé la* racine *de l'arbre et notée* $root((E, R))$*,* E, R, r *vérifient :*

- *pour tout* $e \in E$ *:* $(r, e) \in R^*$ *(*R^* *est la clôture réflexive transitive de* R*, cf. annexe A) ;*
- $(r, r) \notin R$ *;*
- *pour tout* $e \in E \setminus \{r\}$*, il existe un unique* $e' \in E$*, appelé le* père *de* e*, tel que* $(e', e) \in R$ *;*

Soit (E, R) un arbre. Si $(e, e_1) \in R$ alors e_1 est appelé un *fils* de e. De plus, si $(e, e_2) \in R$, $e_1 \neq e_2$, alors e_1 est appelé un *frère* de e_2. Une *feuille* e est un nœud qui n'a aucun fils (pour tout $e' \in E : (e, e') \notin R$). Un *descendant* de e est un nœud e' tel que $(e, e') \in R^+$. Une *branche* de l'arbre est une suite de nœuds telle que :

- le premier nœud de la suite est la racine ;
- tout nœud de la suite, excepté le premier, est un fils du nœud qui le précède dans la suite ;
- la suite est infinie ou son dernier nœud est une feuille.

L'arbre (E, R) est *bien fondé* s'il ne contient pas de branches infinies (en termes plus formels, si R^{-1} est bien fondée). De plus, si E est fini, sa *profondeur* est la longueur de sa plus longue branche moins 1.

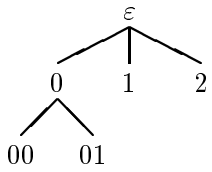


Figure 1.1 : Représentation graphique d'un arbre (orienté)

Un *arbre orienté* est un arbre (E, R) muni d'une famille d'ordres (stricts) $\{<_e\}_{e\in E}$, i.e. un triplet $(E, R, \{<_e\}_{e\in E})$, tel que, pour tout nœud e: $<_e$ est un ordre total (strict) sur l'ensemble des fils de e.

Exemple 1.1.1 Arbre
Soit $E = \{\varepsilon, 0, 00, 1, 2, 01\}$, $R = \{(\varepsilon, 0), (\varepsilon, 1), (0, 00), (0, 01), (\varepsilon, 2)\}$ et $<_\varepsilon = \{(0, 1), (0, 2), (1, 2)\}$, $<_0 = \{(00, 01)\}$ et $<_{00} = <_1 = <_{01} = <_2 = \emptyset$.
(E, R) est un arbre (bien fondé et même fini), ε est sa racine, 0 est un fils de ε, 00 est un frère de 01, 01 est une feuille, 00 est un descendant de ε, et ε 0 01 est une branche. Sa profondeur est 2.
$(E, R, \{<_e\}_{e\in E})$ est un arbre orienté.
L'arbre (E, R) (ou l'arbre $(E, R, \{<_e\}_{e\in E})$) sera représenté comme sur la figure 1.1.
Parfois nous représenterons les arbres par des triangles comme dans la figure 1.3.

Un *arbre étiqueté* est un quadruplet (E, R, F, L), où (E, R) est un arbre, F est son ensemble d'*étiquettes* et L est sa *fonction d'étiquetage* de E dans F. On dit que (E, R, F, L) est *étiqueté* par F et qu'il est *enraciné* par f si f est l'étiquette de sa racine.

Dans l'utilisation des arbres orientés étiquetés, on s'intéresse souvent plus à la relation de parenté, aux ordres entre les fils et à la fonction d'étiquetage qu'à la nature même des nœuds. Pour ces arbres, sont définis des domaines d'arbre appelés *domaine d'arbre standards.*

Définition 1.1.2 Domaine d'arbre standard
Un domaine d'arbre standard *est un ensemble E de suites finies d'entiers, notées comme des mots sur $\mathbb{N}$ (voir section A.2), i.e. E est un langage*[1] *sur $\mathbb{N}$, tel que, pour tout $N \in \mathbb{N}^*$, pour tout $(m, n) \in \mathbb{N}^2$:*

- *$\varepsilon \in E$;*
- *si $N \cdot n \in E$ alors $N \in E$;*
- *si $N \cdot n \in E$ et $m < n$ alors $N \cdot m \in E$.*

Exemple 1.1.2 Domaine d'arbre standard
Le domaine de l'arbre de l'exemple 1.1.1 est un domaine d'arbre standard (voir figure 1.1).

L'arbre orienté sur le domaine d'arbre standard E est l'arbre $(E, R, \{<_N\}_{N\in E})$ défini par :

- ε est sa racine ;

[1] Ne pas confondre la notation $\mathbb{N}^*$ qui désigne l'ensemble des suites finies d'entiers avec la même notation utilisée, parfois, pour désigner l'ensemble des entiers strictement positifs.

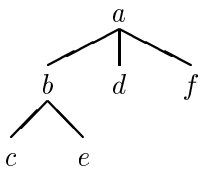


Figure 1.2 : Représentation d'un arbre orienté étiqueté

- pour tout nœuds N et N' de E : $(N, N') \in R$ si et seulement si il existe $n \in \mathbb{N}$ tel que $N' = N \cdot n$;
- l'ensemble d'ordre $\{<_N\}_{N \in E}$ est défini par : pour tout $N \in E$, soit $N \cdot m$ et $N \cdot n$ deux fils de N, $N \cdot m <_N N \cdot n$ si $m < n$.

  Remarque. Si N est un nœud d'un arbre sur un domaine d'arbre standard alors les fils de N sont les nœuds de la forme $N \cdot n$ avec $n \in \mathbb{N}$. Si $N \cdot n_1$ est un nœud de l'arbre ($n_1 \in \mathbb{N}$) alors les frères de $N \cdot n_1$ sont les nœuds de la forme $N \cdot n_2$ avec $n_2 \in \mathbb{N}$, $n_2 \neq n_1$. ◇

La *profondeur* d'un arbre sur un domaine d'arbre standard fini E correspond à la longueur du mot le plus long de E.

Un arbre orienté sur un domaine d'arbre standard E est, par définition, entièrement caractérisé par E. Dans la suite, on omet R et $\{<_N\}_{N \in E}$ et on le note E.

Naturellement, on peut définir des arbres qui sont à la fois orientés et étiquetés, mais ici nous n'avons besoin que de la définition suivante :

Définition 1.1.3 Arbre orienté étiqueté
Un arbre orienté étiqueté *est un triplet (E, L, F), où E est un domaine d'arbre standard, F est un ensemble d'étiquettes et L est une fonction d'étiquetage de E dans F.*

Exemple 1.1.3 Arbre orienté étiqueté
Reprenons l'arbre orienté $(E, R, \{<_e\}_{e \in E})$ de l'exemple 1.1.1. Soit $F = \{a, b, c, d, e, f\}$ et $L : E \to F$ définie par : $L(\varepsilon) = a$, $L(0) = b$, $L(1) = d$, $L(2) = f$, $L(00) = c$ et $L(01) = e$.
(E, F, L) est un arbre orienté étiqueté. Un arbre orienté étiqueté est représenté comme un arbre excepté, qu'on indique l'étiquette à la place du nœud (voir figure 1.2).

Afin d'alléger le texte, on utilise les notations suivantes : si T est un arbre orienté étiqueté, on note

- dom_T son domaine d'arbre standard ;
- lab_T sa fonction d'étiquetage.

Soit $T = (dom_T, lab_T, F)$ et $T'' = (dom_{T''}, lab_{T''}, F)$ deux arbres orientés étiquetés par F. La *greffe* de T'' sur le nœud $N \in dom_T$ dans T est l'arbre orienté étiqueté $T' = (dom_{T'}, lab_{T'}, F)$, noté $graft(T, N, T'')$, défini de la manière suivante : (voir figure 1.3)

- $dom_{T'} = \{N' \in dom_T \mid N \text{ n'est pas un préfixe de } N'\} \cup \{N \cdot N'' \mid N'' \in dom_{T''}\}$;
- pour tout $N' \in dom_{T'}$, si $N' = N \cdot N''$ alors $lab_{T'}(N') = lab_{T''}(N'')$ sinon $lab_{T'}(N') = lab_T(N')$ (en particulier, l'étiquette de N est celle de la racine de T'').

On dit que T'' est l'arbre *enraciné* en N dans $graft(T, N, T'')$.

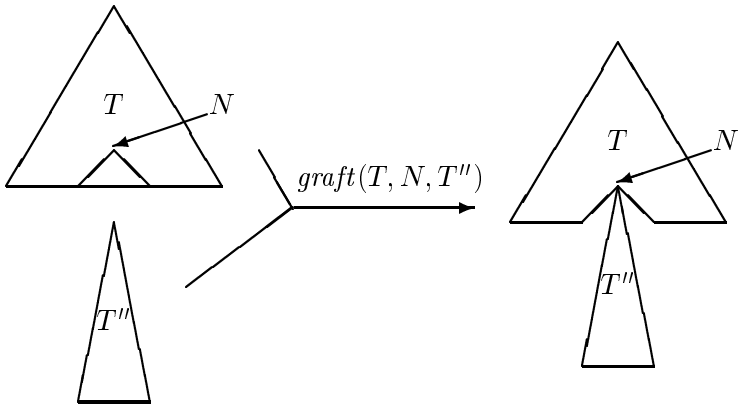


Figure 1.3 : $graft(T, N, T'')$

1.2 Définitions inductives

Nous présentons dans cette section le cadre des définitions inductives [1]. Ce cadre sera largement utilisé dans le chapitre 3 pour définir des sémantiques déclaratives et dans le chapitre 4 pour définir un schéma général de diagnostic déclaratif d'erreur.

Soit E un ensemble. Une *règle* sur l'ensemble E est une paire notée $x \leftarrow X$, où $X \subseteq E$ est l'ensemble des *prémisses* et $x \in E$ est la *conclusion*. Une règle est *finitaire* quand l'ensemble de ses prémisses est fini.

Soit Φ un ensemble de règles sur E. L'*opérateur associé* à Φ est l'opérateur $T_\Phi : 2^E \rightarrow 2^E$ tel que $T_\Phi(I) = \{x \in E \mid \exists X \subseteq I, x \leftarrow X \in \Phi\}$. T_Φ est *monotone*[2] (pour l'inclusion sur 2^E).

Preuve. T_Φ est monotone:
Soient $I \subseteq J$ deux sous-ensembles de E; si $x \in T_\Phi(I)$ alors il existe $X \subseteq I$ tel que $x \leftarrow X \in \Phi$, d'où $x \in T_\Phi(J)$. ∎

Réciproquement, on peut montrer que tout opérateur monotone sur 2^E est l'opérateur associé d'un système de règles sur E.

Définition 1.2.1 Ensemble clos par un opérateur monotone
On dit que $I \subseteq E$ est clos[3] *par T_Φ si $T_\Phi(I) \subseteq I$.*

De tels ensembles existent toujours, e.g. l'ensemble des conclusions des règles de Φ.

Définition 1.2.2 Ensemble défini inductivement par un opérateur monotone
*L'*ensemble défini inductivement *par Φ (ou par T_Φ) est $ind(\Phi) = \bigcap\{I \in 2^E \mid T_\Phi(I) \subseteq I\}$.*

L'intersection d'une famille d'ensembles clos par T_Φ est close par T_Φ. En fait, $ind(\Phi)$ est le plus petit ensemble clos par T_Φ. Donc, si I est clos par T_Φ alors $ind(\Phi) \subseteq I$. C'est le principe général de preuve par induction: pour montrer que $ind(\Phi) \subseteq I$, il suffit de montrer que I est clos par T_Φ.

[2] Nous employons ici le terme opérateur monotone pour désigner un opérateur croissant. Cette terminologie issue de l'anglais est de plus en plus utilisée; à l'origine, un opérateur monotone désigne un opérateur soit croissant soit décroissant.

[3] Un ensemble clos par T_Φ est aussi appelé un *pré-point fixe* de T_Φ.

Preuve. L'intersection d'une famille d'ensembles clos par T_Φ est close par T_Φ :
soit $\{I_j\}_{j\in J}$ une famille d'ensembles clos par T_Φ, soit $x \in T_\Phi(\bigcap_{j\in J} I_j)$, il existe $x \leftarrow X \in \Phi$ tel que $X \subseteq \bigcap_{j\in J} I_j$, donc $x \in I_j$, pour tout $j \in J$, par conséquent $x \in \bigcap_{j\in J} I_j$. ■

$ind(\Phi)$ est aussi le plus petit I tel que $T_\Phi(I) = I$, i.e. le plus petit *point fixe* de T_Φ, noté $pppf(T_\Phi)$.

Preuve. $ind(\Phi)$ est le plus petit point fixe de T_Φ :
$ind(\Phi)$ est plus petit que tous les points fixes de T_Φ puisque les points fixes de T_Φ sont clos par T_Φ. $T_\Phi(ind(\Phi))$ est clos par T_Φ ($ind(\Phi)$ est clos par T_Φ et T_Φ est monotone) donc $ind(\Phi) \subseteq T_\Phi(ind(\Phi))$. $ind(\Phi)$ est un point fixe de T_Φ. Nous avons utilisé le principe de preuve par induction. ■

On retrouve, un cas particulier du théorème de Knaster-Tarski [57] mais la démonstration est ici triviale car $(2^E, \subseteq)$ est un treillis complet.

Définition 1.2.3 Opérateur monotone compact
Un opérateur $T_\Phi : 2^E \to 2^E$ est compact *si, pour tout $I \subseteq E$, pour tout $x \in T_\Phi(I)$, il existe une partie finie $I_f \subseteq I$ telle que $x \in T_\Phi(I_f)$.*

Lemme 1.2.4 *Si les règles de Φ sont finitaires alors l'opérateur T_Φ est compact.*

Preuve. L'ensemble des prémisses d'une règle est fini. ■

"*A note on terminology.* Monotone, compact operators are generally called continuous operators in the literature, because in the right topology they are continuous functions. In mathematical logic, the *Compactness Theorem* says: Any statement that is a logical consequence of a set of statements S is a logical consequence of some finite subset S_0 of S. The compactness of the operators [...] can be seen as a special case of this result of logic. Since the role of finiteness is central to our work here, while topological notions are peripheral, we have choosen to refer to compact, rather than continuous operators."

Melvin Fitting, [45], page 62.

Il n'est rien à ajouter à cette citation de Fitting qui explique pourquoi nous ne parlerons pas de treillis, partie dirigée, continuité, etc.

Le plus petit point fixe de T_Φ peut s'obtenir à partir des itérations ascendantes (des puissances ordinales[4]) de T_Φ :

- $T_\Phi \uparrow 0 = \emptyset$;
- $T_\Phi \uparrow \alpha + 1 = T_\Phi(T_\Phi \uparrow \alpha)$ pour tout ordinal successeur $\alpha + 1$;
- $T_\Phi \uparrow \beta = \bigcup_{\alpha<\beta} T_\Phi \uparrow \alpha$ pour tout ordinal limite β.

D'après le théorème de Knaster-Tarski [57], il existe un ordinal α, appelé *ordinal de fermeture* de T_Φ, tel que $pppf(T_\Phi) = T_\Phi \uparrow \alpha$. De plus, si T_Φ est compact alors $pppf(T_\Phi) = T_\Phi \uparrow \omega$ (ω est le premier ordinal limite).

[4]Nous ne redéfinissons pas les ordinaux, qui ne sont qu'une notion intermédiaire peu utilisée dans ce travail.

Preuve. Si T_Φ est compact alors $pppf(T_\Phi) = T_\Phi \uparrow \omega$:

- $T_\Phi \uparrow \omega$ est clos par T_Φ : Soit $x \in T_\Phi(T_\Phi \uparrow \omega)$. Comme T_Φ est compact, il existe un ensemble fini $I_f \subseteq T_\Phi \uparrow \omega$ tel que $x \in T_\Phi(I_f)$. Comme $T_\Phi \uparrow n' \subseteq T_\Phi \uparrow n'+1$, pour tout $n' \in \mathbb{N}$, alors il existe $n \in \mathbb{N}$ tel que $I_f \subseteq T_\Phi \uparrow n$. Comme T_Φ est monotone, $x \in T_\Phi \uparrow n+1 \subseteq T_\Phi \uparrow \omega$. Donc $T_\Phi \uparrow \omega$ est clos par T_Φ.
- $T_\Phi \uparrow \omega$ est le plus petit ensemble clos par T_Φ : Soit I clos par T_Φ. $T_\Phi \uparrow 0 = \emptyset \subseteq I$, si $T_\Phi \uparrow n \subseteq I$ alors $T_\Phi \uparrow n+1 \subseteq T_\Phi(I) \subseteq I$. Donc $T_\Phi \uparrow \omega \subseteq I$.

Donc $T_\Phi \uparrow \omega = ind(\Phi) = pppf(T_\Phi)$. ∎

Du point de vue dual,

Définition 1.2.5 Ensemble supporté par un opérateur monotone
On dit que $I \subseteq E$ est supporté[5] *par T_Φ si $T_\Phi(I) \supseteq I$.*

Définition 1.2.6 Ensemble défini co-inductivement par un opérateur monotone
*L'*ensemble défini co-inductivement *par Φ (ou par T_Φ) est $coind(\Phi) = \bigcup\{I \in 2^E \mid T_\Phi(I) \supseteq I\}$.*

L'union d'une famille d'ensembles supportés par T_Φ est supportée par T_Φ. $coind(\Phi)$ est le plus grand ensemble supporté par T_Φ. Donc, si I est supporté par T_Φ alors $coind(\Phi) \supseteq I$.

Preuve. L'union d'une famille d'ensembles supportés par T_Φ est supportée par T_Φ :
soit $\{I_j\}_{j\in J}$ une famille d'ensembles supportés par T_Φ, soit $x \in \bigcup_{j\in J} I_j$, il existe $j_0 \in J$ tel que $x \in I_{j_0}$ et il existe $x \leftarrow X \in \Phi$ tels que $X \subseteq I_{j_0}$, donc $X \subseteq \bigcup_{j\in J} I_j$, soit $x \in T_\Phi(\bigcup_{j\in J} I_j)$. ∎

$coind(\Phi)$ est aussi le plus grand point fixe de T_Φ, noté $pgpf(T_\Phi)$.

Preuve. $coind(\Phi)$ est le plus grand point fixe de T_Φ :
$coind(\Phi)$ est plus grand que tous les points fixes de T_Φ puisque les points fixes de T_Φ sont supportés par T_Φ. $T_\Phi(coind(\Phi))$ est supporté par T_Φ ($coind(\Phi)$ est supporté par T_Φ et T_Φ est monotone) donc $coind(\Phi) \supseteq T_\Phi(coind(\Phi))$. $coind(\Phi)$ est un point fixe de T_Φ. ∎

Le plus grand point fixe de T_Φ s'obtient à partir des itérations descendantes de T_Φ :

- $T_\Phi \downarrow 0 = E$;
- $T_\Phi \downarrow \alpha+1 = T_\Phi(T_\Phi \downarrow \alpha)$ pour tout ordinal successeur $\alpha+1$;
- $T_\Phi \downarrow \beta = \bigcap_{\alpha<\beta} T_\phi \downarrow \alpha$ pour tout ordinal limite β.

D'après le théorème de Knaster-Tarski [57], il existe un ordinal α tel que $pgpf(T_\Phi) = T_\Phi \downarrow \alpha$. Notons que T_Φ compact n'implique pas $pgpf(T_\Phi) = T_\Phi \downarrow \omega$ (figure 1.4).

[5] Un ensemble supporté par T est aussi appelé un *post-point fixe* de T.

$$\emptyset = T \uparrow 0 \subseteq T \uparrow 1 \subseteq \cdots \subseteq T \uparrow \alpha = \mathit{pppf}(T) \subseteq \mathit{pgpf}(T) = T \downarrow \beta \subseteq \cdots \subseteq T \downarrow 1 \subseteq T \downarrow 0 = E$$

Si T est compact alors $\alpha = \omega$ suffit.

Figure 1.4 : Itérations d'un opérateur monotone T sur $(2^E, \subseteq)$.

1.2.1 Arbres de preuve

On suppose que toutes les règles de Φ sont finitaires (donc T_Φ est compact).

Un *arbre de preuve* pour Φ est un arbre fini sur un domaine d'arbre standard, étiqueté par E, tel que si x est l'étiquette d'un nœud et X est l'ensemble des étiquettes de ses fils alors $x \leftarrow X \in \Phi$. L'ensemble des étiquettes des racines d'arbres de preuve est exactement $\mathit{ind}(\Phi)$, i.e. $x \in \mathit{ind}(\Phi)$ si et seulement si x enracine un arbre de preuve.

Preuve. $\mathit{ind}(\Phi) = \{x \in E \mid x \textit{ enracine un arbre de preuve}\}$:
une simple preuve par induction sur $\mathbb{N}$ montre le résultat plus fort : pour tout entier strictement positif n, $x \in T_\Phi \uparrow n$ si et seulement si x enracine un arbre de preuve de profondeur inférieure ou égale à $n-1$. Comme $\mathit{ind}(\Phi) = T_\Phi \uparrow \omega$, l'égalité est évidente. ■

Un ∞-*arbre de preuve* se définit comme un arbre de preuve excepté qu'il peut être éventuellement infini. L'ensemble des étiquettes des racines d'∞-arbres de preuve est exactement $\mathit{coind}(\Phi)$, i.e. $x \in \mathit{coind}(\Phi)$ si et seulement si x enracine un ∞-arbre de preuve.

Preuve. $\mathit{coind}(\Phi) = \{x \in E \mid x \textit{ enracine un } \infty\textit{-arbre de preuve}\}$:

$\subseteq$ Soit $x \in \mathit{pgpf}(T_\Phi)$. Alors $x \in T_\Phi(\mathit{pgpf}(T_\Phi))$, donc il existe $x \leftarrow X \in \Phi$ telle que $X \in \mathit{pgpf}(T_\Phi)$. Par itération sur les éléments de X, on définit un ∞-arbre de preuve enraciné par x.

$\supseteq$ Supposons qu'il existe x enracinant un ∞-arbre de preuve. Soit X l'ensemble des étiquettes des nœuds de l'∞-arbre de preuve, on constate que $X \subseteq T_\Phi(X)$ donc $X \subseteq \mathit{pgpf}(T_\Phi)$. Or $x \in X$ donc $x \in \mathit{pgpf}(T_\Phi)$.

■

Nous considérons parfois des règles d'une nature différente : leurs prémisses sont des suites finies d'éléments de E (notées comme des mots sur E). Les notions d'opérateur associé, d'ensemble clos ou supporté, sont définies à partir du système de règles où les suites finies sont remplacées par l'ensemble de leurs éléments. Les résultats précédents restent valides. Seules les définitions d'arbres de preuve et d'∞-arbres de preuve sont modifiées. Un arbre de preuve, pour un tel système de règles, est un arbre fini *orienté* étiqueté par E tel que :

- si x est l'étiquette d'un nœud N ;
- si N a n fils ;
- si x_i est l'étiquette de $N \cdot (i-1)$, $i = 1, \ldots, n$;

alors $x \leftarrow x_1 \cdots x_n$ est une règle du système.

Exemple 1.2.1 Arbre de preuve
Soit $E = \{a, b, c, d, e, f, g, h, i\}$ et

$$\Phi = \left\{ \begin{array}{lll} a \leftarrow bdf, & c \leftarrow h, & h \leftarrow i, \\ a \leftarrow g, & d \leftarrow \varepsilon, & h \leftarrow g, \\ b \leftarrow ce, & e \leftarrow \varepsilon, & i \leftarrow h \\ c \leftarrow \varepsilon, & f \leftarrow \varepsilon, & i \leftarrow abh \end{array} \right\}$$

L'arbre de la figure 1.2 est un arbre de preuve.
$ind(\Phi) = \{a, b, c, d, e, f\}$ et $coind(\Phi) = \{a, b, c, d, e, f, h, i\}$.

La différence avec les arbres précédents réside essentiellement dans le fait que l'ordre sur les prémisses permet d'orienter les fils d'un nœud, donc d'orienter les arbres de preuve.

De plus, cette définition prend en compte la multiplicité d'un élément en partie droite d'une règle. Ainsi deux prémisses identiques d'une règle peuvent avoir deux preuves différentes enracinées en deux nœuds frères d'un arbre de preuve. Notons que nous aurions pu définir les règles comme des paires constituées d'un élément de E et d'un multi-ensemble d'éléments de E.

On définit aussi, de manière similaire, les ∞-arbres de preuve (orientés) pour de tels systèmes de règles. On montre que les racines des arbres de preuve et des ∞-arbres de preuve correspondent respectivement au plus petit ensemble clos et au plus grand ensemble supporté (les preuves sont identiques aux précédentes).

1.3 Programmes logiques avec contraintes : langage, notations, terminologie

Considérons une fois pour toute quatre ensembles disjoints qui déterminent le langage des programmes :

- un ensemble infini dénombrable de *variables* V ;
- un ensemble dénombrable de *symboles de fonction* Σ ;
- un ensemble dénombrable de *symboles de prédicat de contrainte* Π_c ;
- un ensemble dénombrable de *symboles de prédicat de programme* Π_p.

On suppose que chaque symbole est muni de son arité.

On note $var(E)$ l'ensemble des variables ayant une occurrence *libre* dans E, où E est une formule du langage du premier ordre construit sur $(V, \Sigma, \Pi_c \cup \Pi_p)$.

Définition 1.3.1 Renommage de variables
Un renommage de variables *est une injection θ de V dans V (i.e. pour toute paire de variables distinctes x, y de V : $\theta(x) \neq \theta(y)$).*

Un renommage de variables θ s'étend en une fonction, notée également θ et appelée *renommage*, des expressions du langage dans les expressions du langage de la façon usuelle. On utilise la notation habituelle suffixée pour les renommages, c'est-à-dire, si E est une expression alors l'application de θ à E est notée $E\theta$.

Un *atome* est une formule atomique particulière de la forme $p(x_1, \ldots, x_n)$ construite sur $(V, \emptyset, \Pi_p)$, où $x_1, \ldots, x_n$ sont n variables distinctes de V, n est l'arité de $p \in \Pi_p$. Soit ATOM l'ensemble des atomes. Les notations introduites dans la section A.2 sont utilisées pour noter les suites finies d'atomes[6] (en particulier, la suite vide est notée ε).

Soient θ un renommage et $a_1 a_2 \cdots a_n$ une suite finie d'atomes, $(a_1 a_2 \cdots a_n)\theta$ est la suite finie d'atomes $a_1\theta a_2\theta \cdots a_n\theta$. L'ensemble des variables libres d'une suite finie d'atomes $a_1 \cdots a_n$ est $var(a_1 \cdots a_n) = \bigcup_{i=1,\ldots,n} var(a_i)$.

L'ensemble des *contraintes basiques*[7] CONST est un sous-ensemble du langage du premier ordre construit sur (V, Σ, Π_c). On suppose qu'il contient les *contraintes atomiques* (formules atomiques du langage) et qu'il est clos par renommage des variables.

> *Remarque.* Dans un soucis de généralité, nous avons défini CONST comme un sur-ensemble des contraintes atomiques pouvant inclure éventuellement des formules plus générales, contenant par exemple des quantifications universelles, des disjonctions, des négations, etc. ◇

Considérons le plus petit ensemble de formules qui contient CONST et qui est fermé par conjonction. C'est l'ensemble des *conjonctions de contraintes basiques*. Deux conjonctions de contraintes basiques sont équivalentes si elles sont construites à partir du même ensemble de contraintes basiques. Un *store pur* est une classe d'équivalence de conjonction de contraintes basiques selon cette relation d'équivalence. En pratique, un store pur est noté par n'importe quel élément de la classe ou par l'ensemble de ses contraintes basiques[8].

Considérons maintenant l'ensemble des *quantifications existentielles de conjonctions de contraintes basiques*. Deux formules de cet ensemble $\exists x_1 \cdots \exists x_n F$ et $\exists y_1 \cdots \exists y_m G$, où F et G sont des conjonctions de contraintes basiques, sont équivalentes si :

- elles ont le même ensemble de variables libres ;
- il existe un renommage θ, tel que $x\theta = x$ pour toute $x \notin \{y_1, \ldots, y_m\}$ et tel que F et $G\theta$ appartiennent au même store pur.

On utilise les notations suivantes pour désigner les classes d'équivalences de quantification existentielle de conjonction de contraintes basiques (en plus des éléments de la classe eux-mêmes), où F est une conjonction de contraintes basiques et $\tilde{x}$ est un ensemble fini de variables :

- $\exists_{\tilde{x}} F$, où $\tilde{x} = \{x_1, \ldots, x_n\}$, désigne la classe $\exists x_1 \cdots \exists x_n F$;
- $\tilde{\exists} F$ désigne la classe $\exists_{var(F)} F$;
- $\exists_{-\tilde{x}} F$ est la classe $\exists_{var(F)\setminus\tilde{x}} F$;
- $\exists_{-a} F$, où a est un atome, est la classe $\exists_{-var(a)} F$.

On étend la relation d'équivalence sur les quantifications existentielles de conjonction de contraintes basiques au plus petit ensemble contenant CONST fermé par conjonction et quantification existentielle en utilisant les transformations habituelles permettant de réécrire ces formules sous la forme de

[6] Une suite finie d'atomes est notée comme un mot sur l'alphabet ATOM.
[7] La terminologie contrainte basique a été empruntée à Fages [39].
[8] On confond souvent conjonction de contraintes basiques et ensemble fini de contraintes basiques.

quantifications existentielles de conjonction de contraintes basiques. Un *store* est une classe d'équivalence selon cette dernière relation. Nous notons STORE l'ensemble des stores. Remarquons qu'un store pur est un store.

Si θ est un renommage et C est un store pur dont F est un représentant alors $C\theta$ est le store pur dont un représentant est $F\theta$, i.e. on applique θ à chacune des contraintes basiques de F ; $C\theta$ est un store pur car CONST est fermé par renommage des variables. Soit C un store pur, l'ensemble des variables libres de C, noté $var(C)$, est la réunion des variables libres des contraintes basiques qui constituent C.

Si $\exists_{-\tilde{x}}C$ est un store (C est une conjonction de contraintes basiques) alors $(\exists_{-\tilde{x}}C)\theta = \exists_{-\tilde{x}\theta}C\theta$. L'ensemble des variables libres de $\exists_{-\tilde{x}}C$ est $var(\exists_{-\tilde{x}}C) = \tilde{x} \cap var(C)$.

Les stores (purs) se comportent comme les formules qu'ils représentent pour : la conjonction, la quantification existentielle, le renommage, l'ensemble des variables libres. Dans la suite, on confond complètement le store avec un de ses éléments, sachant qu'il est défini à une équivalence près : celle décrite ci-dessus.

> *Remarque.* Les algorithmes de test de satisfaction d'ensemble de contraintes basiques[9] ne peuvent manipuler que des ensembles (ou conjonctions) de contraintes basiques. Les définitions qui précèdent ne servent qu'à exprimer le fait que nous souhaitons considérer certaines formules comme équivalentes, sans toutefois considérer nécessairement équivalentes deux contraintes basiques logiquement équivalentes. En effet, les systèmes de PLC utilisent souvent, pour des raisons d'efficacité, des algorithmes de test de satisfaction de contraintes incomplets. Or, il se trouve que ces algorithmes n'ont pas le même comportement pour des conjonctions de contraintes basiques qui sont pourtant logiquement équivalentes comme le montrent l'exemple 1.3.1 et l'exemple 2.2.3. ◇

Exemple 1.3.1 Solveur incomplet de CLP($\mathcal{R}$)

Le solveur de contraintes de CLP($\mathcal{R}$) fournit trois types de réponses : `yes` le store est satisfiable, `no` le store est insatisfiable, `maybe` il ne peut statuer sur la satisfiabilité du store.
Il répond `yes` pour $x \times x = 1 \wedge x = 1$ et `maybe` pour $x \times x = 1$, néanmoins :

$$\models \exists x(x \times x = 1 \wedge x = 1) \rightarrow \exists x(x \times x = 1)$$

où $\models F$ signifie que F est conséquence logique de la théorie vide, i.e. F est une *tautologie*.
Il répond `no` pour $x = 1 \wedge x = 0$ et `maybe` pour $x \times x = 1 \wedge x \times x = 0$, néanmoins :

$$\models \neg\exists x(x = 1 \wedge x = 0) \rightarrow \neg\exists x(x \times x = 1 \wedge x \times x = 0)$$

> *Remarque.* Les trois ensembles de formules que nous avons considéré,
>
> - l'ensemble des conjonctions de contraintes basiques,
> - l'ensemble des quantifications existentielles de conjonctions de contrainte basiques,
> - le plus petit ensemble contenant CONST fermé par conjonction et quantification existentielle,
>
> correspondent à trois notions essentielles (voir chapitre 2) :
>
> - la fermeture par conjonction est indispensable pour décrire le mécanisme de résolution qui consiste à accumuler des contraintes basiques sous forme d'une conjonction (un store pur) afin de construire une réponse à un but ;

[9] On les appelle plus simplement *solveur de contraintes.*

- les quantifications existentielles de conjonctions de contraintes basiques sont, quant à elles, essentielles pour projeter la conjonction de contraintes basiques, accumulées au cours de la résolution, sur les variables libres du but afin d'abstraire les noms arbitraires des autres variables éventuellement apparues lors de la résolution (on obtient un store) ;
- enfin, la fermeture par conjonction et quantification existentielle est nécessaire pour montrer la "compositionalité ET" [7, 39, 48] de la sémantique, c'est-à-dire que les réponses obtenues pour une suite finie d'atomes ne dépendent que des réponses obtenues pour chacun des atomes.

◇

Définition 1.3.2 Clause
Une clause *est un triplet, noté* $a \leftarrow C \,\square\, A$*, où* a *est un atome,* C *est un store pur et* A *est une suite finie d'atomes.*

On note :

- $head(a \leftarrow C \,\square\, A) = a$, appelé la tête de la clause ;
- $body(a \leftarrow C \,\square\, A) = C \,\square\, A$, appelé le corps de la clause ;
- $store(a \leftarrow C \,\square\, A) = C$, appelé le store (pur) de la clause ;
- $arity(a \leftarrow C \,\square\, A) = n$, où n est la longueur de la suite finie d'atomes A, appelée l'arité de la clause.

Un *fait* est une clause d'arité nulle. On étend à nouveau les renommages et la fonction *var* aux corps de clauses et aux clauses de façon évidente. On appelle *variable existentielle*[10] de la clause $a \leftarrow C \,\square\, A$ tout élément de $var(C \,\square\, A) \setminus var(a)$.

Nous nous limitons à des stores purs dans les corps de clauses car cela correspond mieux à la réalité des systèmes. Il ne s'agit pas d'une restriction puisque les contraintes basiques sont quelconques.

Remarque. Les atomes sont définis comme des relations sur des variables distinctes. Cependant, nous ne perdons rien sur l'expressivité du langage :

$$p(t_1, \ldots, t_n) \leftarrow C \,\square\, p_1(t_{1,1}, \ldots, t_{1,n_1}) \cdots p_m(t_{n,1}, \ldots, t_{m,n_m})$$

s'écrit de manière équivalente

$$\begin{array}{ll} p(x_1, \ldots, x_n) \leftarrow & C \wedge x_1 = t_1 \wedge \cdots \wedge x_n = t_n \wedge \\ & x_{1,1} = t_{1,1} \wedge \cdots \wedge x_{1,n_1} = t_{1,n_1} \wedge \\ & \cdots \\ & x_{m,1} = t_{m,1} \wedge \cdots \wedge x_{m,n_m} = t_{m,n_m} \,\square \\ & p_1(x_{1,1}, \ldots, x_{1,n_1}) \cdots p_m(x_{m,1}, \ldots, x_{m,n_m}) \end{array}$$

où $=$ désigne l'opération réalisée pour identifier les arguments d'appel d'une procédure avec les arguments de la tête d'une clause de la définition de la procédure. Ce prédicat est obligatoirement défini dans tous les systèmes. Il fait toujours partie de l'ensemble

[10] On dit aussi *variable locale*.

Π_c. La seconde syntaxe est certes plus lourde, mais elle correspond mieux à la réalité des transformations faites par le système pour implanter efficacement l'appel de procédure. Ainsi nous isolons parfaitement l'aspect résolution de l'aspect test de satisfaction. Cette restriction, appelée parfois *standardisation* des clauses, est tout à fait habituelle [55, 65, 38, 18, 81], elle simplifie grandement la description de la sémantique opérationnelle.
On peut remarquer, dans le même esprit, que l'ensemble des symboles de fonction n'est pas nécessaire. En effet, comme nous avons supposé que les arguments des atomes sont des variables distinctes (clauses en forme standard), nous pouvons standardiser aussi les contraintes basiques. Par exemple, une contrainte basique peut être vue comme une relation entre ses variables libres en ajoutant autant de prédicats de contraintes que nécessaire à Π_c. Mais cela nous éloignerait de la réalité des systèmes. Une autre façon de procéder est de définir pour chaque symbole de fonction un prédicat de contrainte avec un argument de plus ; c'est le choix fait par PROLOG IV [81, 23, 10]. Dans les deux cas, les symboles de fonctions ne sont qu'une facilité syntaxique de la syntaxe externe pour écrire les contraintes. Ces transformations permettent également de trouver des modèles pour certains solveurs de contraintes incomplets [22]. ◇

Définition 1.3.3 Programme, Nom de clause
Un programme P *est une famille de clauses. L'ensemble des indices de la famille* P *est noté* $cn(P)$.
Un nom de clause *est un élément de* $cn(P)$ *(i.e. un indice de clause).*

La notion de nom de clause permet de différencier deux occurrences d'une même clause dans un programme. Nous verrons lors de la définition de la sémantique opérationnelle la simplicité apportée par cette notion indispensable. Dans la suite, la clause du programme P dont le nom est u sera désignée par $clause_P(u)$, ou plus simplement par $clause(u)$ quand P est fixé.

Définition 1.3.4 Définition d'un prédicat de programme
La définition *du prédicat de programme* p *dans le programme* P *est la sous-famille des clauses de* P *dont la tête a pour symbole de prédicat* p *; cette famille est indicée par un sous-ensemble* $cn(P,p)$ *de* $cn(P)$.
On suppose que pour tout symbole de prédicat $p \in \Pi_p$, $cn(P,p)$ *est fini.*

Remarque. Dans notre cadre, un programme peut être infini. Mais contrairement à beaucoup d'auteurs, nous imposons que la définition d'un prédicat de programme soit finie. En effet, si tel n'est pas le cas alors de nombreuses notions ne sont plus définies ou deviennent mal définies (par exemple : complété du programme, arbre SLD de degré infini, etc.). ◇

Exemple 1.3.2 Fibonacci
Soit FIB le programme :

$0 : fib(x,y) \leftarrow x = 0 \wedge y = 0 \,\square\, \varepsilon$
$1 : fib(x,y) \leftarrow x = 1 \wedge y = 1 \,\square\, \varepsilon$
$2 : fib(x,y) \leftarrow 1 < x \wedge x = x_1 + 1 \wedge x_1 = x_2 + 1 \wedge y = y_1 + y_2 \,\square\, fib(x_1,y_1) fib(x_2,y_2)$
$3 : go1(x,y) \leftarrow x = y \,\square\, fib(x,y)$
$4 : go2(x,y) \leftarrow x = 1 + 1 \,\square\, fib(x,y)$

FIB est un programme sur le langage défini par :

- $\{x, x_1, x_2, y, y_1, y_2\} \subseteq V$;
- $\{0, 1, +\} \subseteq \Sigma$;
- $\{<, =\} \subseteq \Pi_c$;
- $\{fib, go1, go2\} \subseteq \Pi_p$.

L'arité des symboles de Σ, Π_c et Π_p est donnée par le programme.
L'ensemble des contraintes basiques CONST contient : $\{x = a \mid x \in V, a \in term(V, \Sigma)\} \cup \{1 < x \mid x \in V\}$, où $term(V, \Sigma)$ est l'ensemble des termes construits sur (V, Σ).
L'ensemble d'indices (noms de clause) pour FIB est $cn(\text{FIB}) = \{0, 1, 2, 3, 4\}$.
La définition de fib dans FIB est l'ensemble des clauses dont les noms sont $0, 1, 2$.

Définition 1.3.5 But
Un but *est un atome a, mais pour des raisons (pré)historiques, nous avons tenu à l'écrire* $\leftarrow a$.

Nous considérons des buts atomiques plutôt que des buts généraux parce qu'un but général $\leftarrow C \,\square\, A$ définit une nouvelle relation p sur ses variables libres et il est toujours possible de considérer un but atomique $\leftarrow p(x_1, \ldots, x_n)$, où $\{x_1, \ldots, x_n\} = var(C \,\square\, A)$, en ajoutant un nouveau prédicat p à Π_p dont la définition dans le programme contient l'unique clause $p(x_1, \ldots, x_n) \leftarrow C \,\square\, A$. De plus,

1. les réponses à un but général ne sont construites qu'en regroupant les réponses à des buts atomiques (compositionalité ET de la sémantique) ;
2. la sémantique opérationnelle est entièrement caractérisée par les réponses aux buts atomiques [12, 39, 47] ;
3. le programmeur définit souvent une clause du type $go(x_1, \ldots, x_n) \leftarrow goal$, pour faciliter l'utilisation de son programme et ainsi ne poser que des questions atomiques[11]. Même si cela n'est pas le cas, d'un point de vue théorique, on peut toujours considérer qu'un but est une clause du programme dont on omet de préciser la tête parce qu'elle ne dépend que du but. Son symbole de prédicat n'apparaît nul part ailleurs dans le programme. Le programmeur ne précise qu'une partie du programme, l'autre partie est l'ensemble des clauses définies par les buts que l'on peut construire à partir des prédicats définis par le programmeur (autant de prédicats de programme que nécessaire sont ajoutés à Π_p).

En procédant ainsi nous simplifions avantageusement les diverses définitions. De plus, nous retrouvons l'ensemble des résultats connus (y compris le lemme 3.2.33).

Pour finir nous définissons trois objets particulièrement utiles à la description de la sémantique déclarative des programmes.

Définition 1.3.6 Atome contraint, Atome couvert, Couverture locale d'atomes
Un atome contraint *est une paire, notée* $a \leftarrow C$, *où a est un atome et C est un store.*
Un atome couvert *est une paire, notée* $a \rightarrow R$, *où a est un atome et R est un ensemble de stores*[12].
Une couverture locale d'atomes *est un triplet, noté* $C \,\square\, A \rightarrow R$, *où C est un store, A est une suite finie d'atomes et R est un ensemble de stores.*

---

[11] Peut-être parce que les interfaces des systèmes sont généralement peu conviviales.

[12] Nous confondons un ensemble fini de stores avec la disjonction des stores de l'ensemble, comme nous avons confondu un ensemble fini de contraintes basiques avec la conjonction des contraintes basiques de l'ensemble.

Chapitre 2

Sémantique opérationnelle

Ce chapitre décrit la sémantique opérationnelle d'un programme logique avec contraintes. C'est-à-dire, étant donné un programme P, comment construire les réponses à un but $\leftarrow a$.

Il n'est pas question de définir une nouvelle sémantique opérationnelle. Nous ne faisons que reformuler la sémantique habituelle dans un cadre favorable à l'étude du diagnostic déclaratif d'erreur. Nous mettons en avant la notion de *squelette* qui est intrinsèque au programme (indépendante du calcul) et qui rassemble toute l'information déclarative sur un calcul. Les squelettes relient simplement la sémantique opérationnelle (dérivation SLD) et la sémantique déclarative (arbres de preuve). L'effort fourni dans cette reformulation, s'avère rapidement payant pour

- avoir des définitions simples, par exemple les réponses à un but ou l'arbre SLD pour un but selon une règle de calcul ;
- avoir des preuves simples, par exemple la correction/complétude de la résolution SLD ou l'indépendance de la règle de calcul ;
- faire le lien avec les sémantiques déclaratives de type point fixes ou logiques, voir le chapitre 3 ;
- apporter de nouveaux résultats, par exemple une caractérisation des branches non échecs des arbres SLD équitables ;
- diagnostiquer des erreurs dans un programme, voir le chapitre 4.

Pour un langage de programmation, quand une notion de calcul est fixée, la sémantique opérationnelle des programmes peut être décrite par l'ensemble des calculs finis (ou une abstraction de ces calculs).

Il ressort de notre reformulation, dont la motivation essentielle est létude de la correction partielle des programmes (principalement le diagnostic déclaratif d'erreur), que deux niveaux de calcul peuvent être considérés pour les langages du paradigme de programmation relationnelle.

1. Les calculs positifs calculent les réponses positives et déterminent la sémantique positive des programmes. Ce sont les dérivations SLD. Ils correspondent au type de calcul le plus souvent considéré dans la littérature. Une réponse positive est l'état final d'une dérivation SLD fini succès.
2. Les calculs négatifs calculent les réponses négatives et déterminent la sémantique négative des programmes. Ce sont les arbres SLD. Contrairement à l'habitude, la sémantique négative n'est pas déterminée seulement par les échecs finis. Une réponse négative est l'ensemble des feuilles succès d'un arbre SLD fini.

Dans la suite, les réponses positives sont appelées réponses et les réponses négatives sont appelées $\vee$-réponses.

Nous supposons, pour l'ensemble de ce chapitre, qu'un programme P sur le langage $(V, \Sigma, \Pi_c \cup \Pi_p)$ est fixé.

Nous commençons par présenter la sémantique opérationnelle la plus répandue [55] dans la section 2.1 comme une introduction à notre reformulation dans les sections qui suivent.

2.1 Présentation classique

La sémantique opérationnelle des programmes logiques avec contraintes la plus fréquemment proposée est la résolution descendante. Il existe néanmoins des approches ascendantes[55], mais elles ne décrivent pas le comportement de la majorité des systèmes de PLC.

Nous présentons succinctement dans cette section le résolution descendante de l'article de synthèse classique de Jaffar et Maher [55] en l'adaptant à notre cadre. Elle est paramétrée par une relation *consistent* et une fonction *infer* permettant de décrire le fonctionnement de la majorité des solveurs de contraintes. En effet, les solveurs de contraintes sont souvent incomplets[1]. La fonction *infer* permet, à partir du store courant, de distinguer les contraintes *actives* des contraintes *passives* (ou retardées). La relation *consistent* teste la consistance de l'ensemble de contraintes actives. Ainsi, on décrit plus précisément le comportement opérationnel (ce qui est réellement calculé) grâce à la fonction abstraite *infer* et à la relation abstraite *consistent* que si l'on s'intéresse à l'interprétation ou la théorie sous-jacente des contraintes comme dans [54, 65, 47, 97, 39].

La résolution descendante est décrite par un système de transition sur des états $\langle A, C, S\rangle$, où A est un but général (le but courant), i.e. de même nature qu'un corps de clause, C (le store actif courant) et S (le store passif courant) sont des stores purs. Le système de transition est paramétré par une relation *consistent* et une fonction *infer*. Il existe un autre état noté *fail*. Une règle de calcul sélectionne pour chaque état un élément de A.

Remarque.

> "A computation rule is a convenient fiction that abstracts some of the behaviour of a CLP system. To be realistic, a computation rule should also depend on factors other than the state (for example, the history of the computation). We ignore these possibilities for simplicity."
>
> Joxan Jaffar and Michael J. Maher, [55], page 18.

La définition de règle de calcul que nous donnerons par la suite n'ignore pas ces possibilités. Pourtant elle est très simple et définie également comme une fonction sur les états de calculs. Notre notion d'état de calcul contient exactement tout ce qui est nécessaire pour décrire aisément la résolution. ◇

Le but initial est représenté par l'état $\langle G, \emptyset, \emptyset\rangle$.

Soit $\mathcal{D}$ l'interprétation sous-jacente des contraintes (voir section 3.2 pour plus de détails). $\models_{\mathcal{D}} F$, où F est une formule construite sur (V, Σ, Π_c), signifie : pour toute valuation v dans $\mathcal{D}$, $v(F)$ est vraie dans $\mathcal{D}$.

[1] Voir l'exemple 1.3.1 qui montre qu'il n'est pas possible de rendre compte, simplement, de cet incomplétude dans un cadre logique.

La relation *consistent* est telle que $\models_{\mathcal{D}} \tilde{\exists} C$ implique $C \in$ *consistent*. La fonction *infer* est telle que si $\mathit{infer}(C, S) = (C', S')$ alors $\models_{\mathcal{D}} (C \wedge S) \leftrightarrow (C' \wedge S')$.

Les transitions du système de transition sont les suivantes :

- $$\langle \leftarrow C' \square A_1 \cdot a \cdot A_2, C, S \rangle \rightarrow_r \langle \leftarrow C' \wedge C'' \square A_1 \cdot A'' \cdot A_2, C, S \wedge a = a'' \rangle$$

  si a est l'atome sélectionné par la règle de calcul, $a'' \leftarrow C'' \square A''$ est une clause renommée avec de nouvelles variables de la définition du symbole de prédicat de a dans le programme.

- $$\langle \leftarrow C' \square A_1 \cdot a \cdot A_2, C, S \rangle \rightarrow_r \mathit{fail}$$

  si a est l'atome sélectionné par la règle de calcul, et la définition du symbole de prédicat de a dans le programme est vide.

- $$\langle \leftarrow C' \wedge c \square A, C, S \rangle \rightarrow_c \langle \leftarrow C' \square A, C, S \wedge c \rangle$$

  si c est la contrainte basique sélectionnée par la règle de calcul.

- $$\langle G, C, S \rangle \rightarrow_i \langle G, C', S' \rangle$$

  si $\mathit{infer}(C, S) = (C', S')$.

- $$\langle G, C, S \rangle \rightarrow_s \langle G, C, S \rangle$$

  si $C \in$ *consistent*.

- $$\langle G, C, S \rangle \rightarrow_s \mathit{fail}$$

  si $C \notin$ *consistent*.

Une dérivation succès à partir de l'état $\langle G, \emptyset, \emptyset \rangle$ est une dérivation finie ayant comme état final $\langle \leftarrow \emptyset \square \varepsilon, C, S \rangle$. Dans ce cas, (C, S) est un *store réponse* pour le but G.

Une suite finie de transitions $s_0 \rightarrow_{x_0} s_1, \ldots, s_{n-1} \rightarrow_{x_{n-1}} s_n$ est noté en résumé $s_0 \rightarrow_{(x_0 \cdots x_{n-1})} s_n$. On écrit aussi $s \rightarrow_{x|y} s'$ si, soit $s \rightarrow_x s'$, soit $s \rightarrow_y s'$.

Un système de PLC est dit *quick-checking* si sa sémantique opérationnelle peut se décrire en terme de $\rightarrow_{ris}$ et $\rightarrow_{cis}$.

Un système de PLC est dit *progressif* si, pour tout état ayant un but contenant une suite non vide d'atomes, toute dérivation à partir de cet état, soit aboutit à l'état *fail*, soit contient une dérivation $\rightarrow_r$ ou une dérivation $\rightarrow_c$.

Un système de PLC est *idéal* si il est quick-checking, progressif, *infer* est définie par $\mathit{infer}(C, S) = (C \wedge S, \emptyset)$ et $C \in$ *consistent* si et seulement si $\models_{\mathcal{D}} \tilde{\exists} C$.

[55] annonce :

> "all major implemented CLP systems are quick-checking and progressive, but most are not ideal".

En fait, tous les systèmes de PLC ont la propriété suivante, appelée *incrémentalité* de *infer* et *consistent* dans [7] :

- $\mathit{infer}(\emptyset, \emptyset) = (\emptyset, \emptyset)$;

- $\emptyset \in \mathit{consistent}$;
- $\mathit{infer}(\emptyset, S) = (C_1, S_1)$ et $\mathit{infer}(C_1, S_1 \wedge c) = (C_2, S_2)$ et $C_1 \in \mathit{consistent}$ et $C_2 \in \mathit{consistent}$ est équivalent à

  $\mathit{infer}(\emptyset, S \wedge c) = (C'_2, S'_2)$ et $C'_2 \in \mathit{consistent}$,

  où (C'_2, S'_2) est obtenue à partir de (C_2, S_2) en renommant éventuellement les variables exceptées celles de $S \wedge c$.

Lemme 2.1.1 *Si infer et consistent sont incrémentales, et le système est quick-checking alors la sémantique opérationnelle est ET-compositionnelle.*

Preuve. Voir [7]. ■

Ce résultat est essentiel et cette propriété d'incrémentalité est toujours recherchée lors de la définition d'un solveur de contraintes. Comme nous l'avons précisé, les systèmes existants les plus connus vérifient ces propriétés (incrémentalité du solveur et système quick-checking).

Un système de PLC ne fait donc qu'accumuler des contraintes basiques lors de la résolution en testant si le store actif courant appartient à *consistent*.

En fait, la forme de la contrainte réponse ne dépend que des contraintes basiques accumulées et le système dispose d'un *critère de rejet* pour le store courant S qui stoppe la résolution quand $\mathit{infer}(\emptyset, S) = (C', S')$ et $C' \notin \mathit{consistent}$. Ce critère ne dépend que de S. D'autre part le store réponse $\mathit{infer}(\emptyset, S) = (C', S')$ ne dépend aussi que de S (au nom des variables, non présentes dans le but, près). D'un point de vue théorique, pour les objectifs de ce travail, il n'est donc pas utile de modéliser l'information calculée par le solveur de contraintes (approximations, store actif, contraintes basiques retardées, etc.) parce que cette information ne dépend que de S et est disponible à tout instant grâce à *infer*.

Nous nous limitons, ici, à la description de cette sémantique opérationnelle, nous en aborderons une autre : la sémantique généralisée [48], dans la section 3.5.

Nous présentons maintenant notre reformulation en commençant par l'outil de base : le *squelette*.

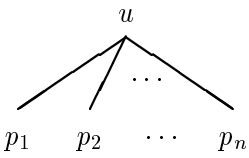
2.2 Squelettes

Définition 2.2.1 Squelettes
Un squelette S *est un arbre orienté étiqueté*[2] *par* $cn(P) \cup \Pi_p$, *tel que pour tout nœud* $N \in dom_S$:

- *si* $lab_S(N) \in \Pi_p$ *alors* N *est une feuille ;*
- *si* $lab_S(N) \in cn(P)$ *et* $clause(lab_S(N)) = a \leftarrow C \,\square\, p_1(x_1^1, \ldots, x_{m_1}^1) \cdots p_n(x_1^n, \ldots, x_{m_n}^n)$ *alors* N *a* n *fils* $N_1, \ldots, N_n$ (N_i *est* $N \cdot (i-1)$), *et chaque fils* N_i *est étiqueté par*
  - *soit* p_i,
  - *soit le nom d'une clause de la définition de* p_i *(un élément de* $cn(P, p_i)$*).*

  Remarque. Les squelettes finis pourraient être définis comme des termes sur un alphabet sorté. ◇

[2] Rappelons que si S est un arbre orienté étiqueté alors son domaine d'arbre est noté dom_S et sa fonction d'étiquetage est notée lab_S.



u est le nom de la clause $a \leftarrow C \,\square\, a_1 a_2 \cdots a_n$
et p_i est le symbole de prédicat de a_i

Figure 2.1 : $sq(u)$

Soit S un squelette et N un nœud de dom_S. Le *symbole de prédicat associé* au nœud N est

- $lab_S(N)$ si $lab_S(N) \in \Pi_p$,
- p si $lab_S(N) \in cn(P, p)$;

On dit que S est un squelette pour $\leftarrow p(x_1, \ldots, x_n)$ (ou plus simplement pour p) quand le symbole de prédicat associé à sa racine est p.

On note $sq(u)$, où $u \in cn(P)$, le squelette dont la racine est étiquetée par u et a ses fils étiquetés par des éléments de Π_p (son domaine d'arbre est $\{\varepsilon, 0, 1, \ldots, n-1\}$ si n est l'arité de $clause(u)$), voir figure 2.1. Pour tout symbole de prédicat $p \in \Pi_p$, on note $sq(p)$ le squelette enraciné par p (son domaine d'arbre est $\{\varepsilon\}$).

Soit S un squelette, on note

- $undef(S)$ l'ensemble des nœuds de S étiquetés par des éléments de Π_p, c'est l'ensemble des *nœuds indéfinis* de S ;
- $def(S)$ l'ensemble des autres nœuds (ils sont étiquetés par des éléments de $cn(P)$), c'est l'ensemble des *nœuds définis* de S.

Définition 2.2.2 Squelette complet, Squelette incomplet
Un squelette complet *est un squelette S tel que $undef(S) = \emptyset$.*
Si $undef(S) \neq \emptyset$ alors S est un squelette incomplet.

Exemple 2.2.1 Fibonacci
La figure 2.2 présente un squelette pour le programme FIB de l'exemple 1.3.2.
Son domaine est $\{\varepsilon, 0, 00, 01\}$. Il n'est pas complet car il a une feuille indéfinie : la feuille 01 étiquetée par fib.

La *greffe* du squelette S'' sur le nœud $N \in dom_S$ de S est un squelette $S' = graft(S, N, S'')$, seulement si le symbole de prédicat associé à N par S est le symbole de prédicat associé à la racine de S'' (voir figure 2.3).

S'' est le squelette *enraciné* en N dans $graft(S, N, S'')$.

Nous avons présenté l'outil central de notre étude : le *squelette*. Nous souhaitons maintenant associer un store (pur) à un squelette. Par exemple, cela va nous permettre de distinguer les squelettes "solubles" de ceux qui ne le sont pas.

Comme toujours, nous sommes confrontés au problème de renommage des clauses du programme.

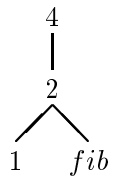
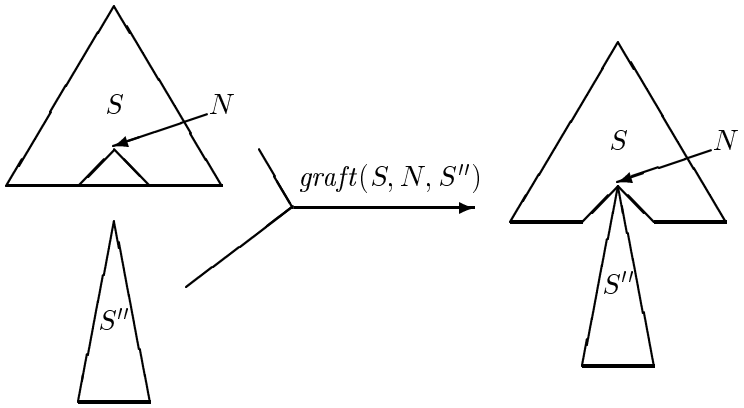


Figure 2.2 : Exemple de squelette pour le programme FIB



Si le symbole de prédicat associé à N dans S est le symbole de prédicat associé à la racine de S''.

Figure 2.3 : $graft(S, N, S'')$

Définition 2.2.3 Fonction de renommage pour un squelette
Soit S un squelette. Une fonction de renommage *pour le squelette S est une fonction $reno_S$ (de $def(S)$ vers l'ensemble des clauses renommées de P) telle que :*

1. *pour tout nœud $N \in def(S)$, il existe un renommage θ, tel que $reno_S(N) = (a \leftarrow C \,\square\, A)\theta$, où $a \leftarrow C \,\square\, A = clause(lab_S(N))$;*
2. *pour tout nœud $N \in def(S)$, soit $reno_S(N) = a \leftarrow C \,\square\, a_1 \cdots a_n$, pour tout $i = 0, \ldots, n-1$, si $(N \cdot i) \in def(S)$ alors $head(reno_S(N \cdot i)) = a_{i+1}$;*
3. *pour tout nœuds distincts N_1 et N_2 de* $\mathrm{def}(S)$*, si $reno_S(N_1) = a_1 \leftarrow C_1 \,\square\, A_1$ et $reno_S(N_2) = a_2 \leftarrow C_2 \,\square\, A_2$ alors $(var(C_1 \,\square\, A_1) \setminus var(a_1)) \cap (var(C_2 \,\square\, A_2) \setminus var(a_2)) = \emptyset$, i.e. les clauses renommées ont leurs variables existentielles qui sont différentes.*

Étant donné un squelette S de profondeur strictement positive, et une fonction de renommage $reno_S$ pour S, pour tout nœud N de S, on appelle l'*atome associé à N* :

- $head(reno_S(N))$, si $N \in def(S)$;
- a_i, si $N \in undef(S)$ est le $i^{\text{ème}}$ fils de N', i.e. $N = N' \cdot (i-1)$, et $reno_S(N') = a \leftarrow C \,\square\, a_1 \cdots a_i \cdots a_n$.

On associe à tout squelette S et toute fonction de renommage $reno_S$ pour S l'ensemble de stores : $const(S, reno_S) = \bigcup_{N \in \mathrm{def}(S)} store(reno_S(N))$. Notons que si S est fini alors la conjonction des éléments de $const(S, reno_S)$ est un store pur. Dans ce cas, on utilise la vision ensembliste d'un

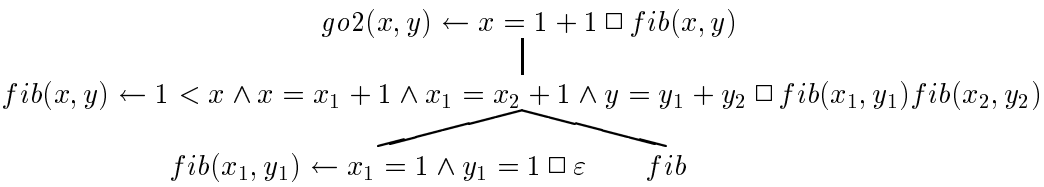


Figure 2.4 : Représentation d'un squelette avec son renommage

store pur : on peut confondre $const(S, reno_S)$ avec la conjonction de ses éléments. Si la racine de S est indéfinie alors $const(S, reno_S) = \emptyset$.

Définition 2.2.4 Fonction de renommage pour un squelette et un but
Soit S un squelette pour $\leftarrow a$, $reno_S$ est une fonction de renommage pour S et $\leftarrow a$ si la racine de S est indéfinie ou si $head(reno_S(root(S))) = a$.

> *Remarque.* Une fonction de renommage pour S et $\leftarrow a$ est en particulier une fonction de renommage pour S. ◇

Si S est un squelette fini pour $\leftarrow a$, et $reno_S$ une fonction de renommage pour S et $\leftarrow a$, on note $\mathrm{AC}(S,a)$ le store $\exists_{-a}(\bigwedge_{C \in const(S,reno_S)} C)$, ou plus simplement en utilisant la vision ensembliste des stores purs $\mathrm{AC}(S,a) = \exists_{-a} const(S, reno_S)$. $\mathrm{AC}(S,a)$ est *le store associé* à S et a. Parfois on dit qu'il est le store associé à S, il est alors défini à un renommage près.

On peut remarquer que $\mathrm{AC}(S,a)$ ne dépend que de S et a, en particulier il ne dépend pas de $reno_S$.

Exemple 2.2.2 Fibonacci
Considérons le squelette S de l'exemple 2.2.1. La figure 2.4 présente la fonction de renommage $reno_S$ pour S (l'étiquette d'un nœud N de $def(S)$ est remplacée par $reno_S(N)$ dans le dessin).
On a $const(S, reno_S) = \{x = 1+1, 1 < x, x = x_1 + 1, x_1 = x_2 + 1, y = y_1 + y_2, x_1 = 1, y_1 = 1\}$ et $\mathrm{AC}(S, go2(x,y)) = \exists x_1 \exists y_1 \exists x_2 \exists y_2 (x = 1+1 \wedge 1 < x \wedge x = x_1 + 1 \wedge x_1 = x_2 + 1 \wedge y = y_1 + y_2 \wedge x_1 = 1 \wedge y_1 = 1)$.

Parmi les squelettes il en existe qui ont un store associé insatisfiable dans l'interprétation des contraintes. Comme nous le verrons par la suite (par exemple dans le chapitre 3) ces squelettes présentent moins d'intérêt.

Par exemple, nous serons amené à considérer le store associé à un squelette fini complet pour $\leftarrow a$ comme un store réponse pour le but $\leftarrow a$ (au sens de la section 2.1). Supposons que le sens donné aux stores réponses C pour le but $\leftarrow a$ est $P \models_{\mathcal{D}} C \rightarrow a$, où P est vu comme un ensemble de formules (clauses de Horn définies) (voir chapitre 3 pour plus de précisions, notamment la définition de $P \models_{\mathcal{D}} F$). Dans ce cas, il est évident que si $\models_{\mathcal{D}} \neg C$ alors $P \models_{\mathcal{D}} C \rightarrow a$ indépendamment du programme P. On reviendra sur cette remarque à la suite du lemme 3.4.10.

D'un point de vue théorique, ces stores réponses présentent peu d'intérêt. D'un point de vue opérationnel, nous devons éviter toute exploration inutile, qui pourrait, de plus, conduire à un calcul infini. Les systèmes de PLC essaient d'éliminer ces "stores réponses triviaux" grâce à ce que nous appellerons un *critère de rejet*.

Exemple 2.2.3 Contraintes non linéaires en CLP($\mathcal{R}$)
Le système essaie d'éliminer, le plus vite possible, les réponses triviales qui ne présentent aucun intérêt pour l'utilisateur. Par exemple, en CLP($\mathcal{R}$), considérons le programme :

$$p(x,y) \leftarrow x > y \,\square\, \varepsilon$$
$$q(x,y) \leftarrow v = x \times x \wedge w = y \times y \,\square\, p(v,w)$$
$$r(x) \leftarrow x = y \,\square\, p(x,y)$$
$$r(x) \leftarrow x = y \,\square\, q(x,y)$$

Il existe deux squelettes complets pour le but $\leftarrow r(x)$ dont les stores associées, une fois simplifiés, sont : $x > x$ et $x \times x > x \times x$. Selon le domaine de contraintes sous-jacent $\mathbb{R}$, les deux sont insatisfiables. Le premier est bien éliminé par le système, tandis que le second est affiché avec le commentaire `maybe`. Le solveur de CLP($\mathcal{R}$) ne peut supprimer que des squelettes dont le store associé est linéaire.

2.3 Critère de rejet

La raison pour laquelle un store est rejeté est, du point de vue de l'utilisateur, basée sur une interprétation $\mathcal{D}$ des contraintes dont le domaine est $\mathbb{D}$. Mais certains stores insatisfiables dans l'interprétation sous-jacente des contraintes ne sont pas rejetés, parce que le solveur est incomplet (e.g. le solveur de CLP($\mathcal{R}$) quand le store n'est pas linéaire, voir l'exemple 2.2.3).

D'un point de vue théorique (plus proche du point de vue solveur de contraintes que la vision utilisateur), un store est rejeté quand sa clôture existentielle n'est pas conséquence logique de la théorie des contraintes dont un modèle est l'interprétation des contraintes sous-jacente.

Mais, d'un point de vue opérationnel ceci n'est pas toujours possible (problème d'indécidabilité, e.g. $(\mathbb{N}, +, \cdot)$) et même quand c'est possible, ce n'est parfois pas souhaitable parce que le coût de la fonction *infer* et de la relation *consistent* seraient inacceptable (e.g. $(\mathbb{R}, +, \cdot)$). De nombreux systèmes de PLC utilisent des solveurs de contraintes incomplets : CLP($\mathcal{R}$) [49], CLP(BNR) [77], PROLOG IV [81], CHIP [36], CLP($\mathcal{FD}$) [35], PROLOG III [80], etc., sur les domaines discrets (parties finies de $\mathbb{N}$) ou les domaines continus ($\mathbb{R}$). Ils sont capables de tester l'insatisfiabilité de certains stores seulement.

> *Remarque.* C'était déjà le cas pour les systèmes de programmation logique qui utilisaient l'algorithme d'unification incomplet (sans test d'occurrence). C'est d'ailleurs la recherche d'un modèle pour cet algorithme qui donna l'idée à Colmerauer d'implanter le premier système de PLC : PROLOG II [20], dont le domaine de contraintes est l'ensemble des arbres infinis (élémentairement équivalent aux arbres rationnels, le solveur devenait alors complet !). ◇

Il est intéressant de noter que la première tâche du solveur de contraintes n'est pas de résoudre des stores, mais plutôt d'essayer de prouver leur insatisfiabilité. Par exemple, Benhamou dans [9] présente un cadre général pour l'étude des solveurs de contraintes. Une contrainte atomique n-aire est vue comme un *opérateur (dit de narrowing)* monotone, contractant (indempotant et correct) de $\mathbb{D}^n \rightarrow \mathbb{D}^n$, où $\mathbb{D}$ est le domaine d'approximation des contraintes. Ces opérateurs réduisent les ensembles de solutions possibles d'un store dans le but de trouver $\emptyset$ quand il est insatisfiable. Au cours d'un calcul les ensembles d'approximations calculés à chaque étape n'ont d'autre intérêt (que de trouver $\emptyset$) puisqu'ils sont toujours plus grands que l'ensemble des solutions du store lui-même. De plus, en fin de calcul, il permettent de contribuer au "problème de présentation" qui

consiste à trouver des moyens de présenter à l'utilisateur des informations longues et complexes (on utilisera par exemple le principe d'énumération suivant certaines heuristiques pour trouver les approximations les plus fines).

Le *critère de rejet* est une relation unaire sur l'ensemble des stores, i.e. un sous-ensemble de STORE. Il exprime le fait que certains stores réponses ne sont pas fournis à l'utilisateur parce qu'on a pu déterminer qu'ils sont insatisfiables dans l'interprétation sous-jacente des contraintes.

Les propriétés naturelles supposées pour un critère de rejet RC sont : pour tout store C rejeté, i.e. $C \in \mathrm{RC}$,

1. pour tout renommage θ, $C\theta \in \mathrm{RC}$;
2. pour tout store C', $C \wedge C' \in \mathrm{RC}$;
3. $\emptyset \notin \mathrm{RC}$.

Le critère de rejet n'est, en général, pas quelconque ; il est le plus souvent défini à partir d'une relation déjà existante et doit satisfaire les trois propriétés. Voici des exemples de critère de rejet définis par :

1. un domaine $\mathcal{D}$, le critère de rejet est noté $\mathrm{RC}_{\mathcal{D}}$, un store C est rejeté si $v(C) = \mathit{false}$, pour toute valuation v dans $\mathcal{D}$, c'est-à-dire $\mathrm{RC}_{\mathcal{D}}$ est défini à partir de la relation $\models_{\mathcal{D}} \neg C$;
2. une théorie[3] $\mathcal{T}$, le critère de rejet est noté $\mathrm{RC}_{\mathcal{T}}$, un store C est rejeté si $C \in \mathrm{RC}_{\mathcal{D}}$, pour tout modèle $\mathcal{D}$ de $\mathcal{T}$, c'est-à-dire $\mathcal{T} \models \neg C$;
3. une relation *consistent* et une fonction *infer*, un store C est rejeté si $\mathit{infer}(\emptyset, C) = (C_1, C_2)$ et $C_1 \notin \mathit{consistent}$;
4. un solveur de contraintes (éventuellement incomplet) $\mathcal{A}$, le critère de rejet est noté $\mathrm{RC}_{\mathcal{A}}$, $C \in \mathrm{RC}_{\mathcal{A}}$ si $\mathcal{A}$ répond `no` pour C.

On constate que les deux premiers exemples satisfont toujours les trois propriétés. Une condition suffisante pour que le troisième exemple satisfasse ces propriétés est que la relation *consistent* et la fonction *infer* soient incrémentales [7, 96]. C'est ce qui est recherché en général. On peut constater que tous les systèmes de PLC (en particulier, tous ceux cités) ont un solveur de contraintes qui définit bien un critère de rejet[4]. La sémantique opérationnelle est donc paramétrée par ce critère et l'on montrera qu'elle généralise les sémantiques classiques.

D'un critère de rejet RC, on déduit une relation sur l'ensemble des squelettes, notée également RC de la manière suivante : $S \in \mathrm{RC}$ si il existe un sous-ensemble fini C de $\mathit{const}(S, \mathit{reno}_S)$, où reno_S est une fonction de renommage pour S, tel que $C \in \mathrm{RC}$. Cette propriété ne dépend pas de reno_S : RC (sur les stores) est fermée par renommage et si reno_S et $\mathit{reno}_S{}'$ sont deux fonctions de renommage pour S alors il existe un renommage θ tel que $\mathit{const}(S, \mathit{reno}_S)\theta = \mathit{const}(S, \mathit{reno}_S{}')$.

Notons qu'un squelette dont la racine est indéfinie n'est pas rejeté. Notons aussi qu'un squelette *fini* S est rejeté si et seulement si $\mathit{const}(S, \mathit{reno}_S)$ est rejeté, où reno_S est une fonction de renommage pour S.

Définition 2.3.1 État (état de calcul selon un critère de rejet)
On appelle état de calcul *selon* RC, *ou plus simplement* état, *un squelette non rejeté par* RC.

[3] Nous ne supposons pas que la théorie est complète pour la satisfaction des stores.
[4] Cette notion d'incrémentalité est indispensable pour donner une sémantique claire au solveur de contraintes.

Remarque. Toutes les notions définies pour les squelettes se transposent aux états. Par exemple, on parle d'état complet, de fonction de renommage pour un état, etc. ◇

Exemple 2.3.1 Fibonacci
Si le critère de rejet est celui défini par $(\mathbb{N}, +)$ avec l'interprétation usuelle des symboles de Σ et Π_c alors le squelette de la figure 2.4 est un état : le store associé n'est pas rejeté.
Une solution dans $\mathbb{N}$ est par exemple : $x = 2, x_1 = 1, x_2 = 0, y = 86, y_1 = 1, y_2 = 85$.

Lemme 2.3.2 *Si $graft(S, N, S')$, où $N \in undef(S)$, est un état alors S et S' sont des états.*

Preuve. Il suffit d'appliquer les définitions : Soit $reno_{S''}$ une fonction de renommage pour $S'' = graft(S, N, S')$. La restriction de $reno_{S''}$ à $def(S)$, notée $reno_S$ est une fonction de renommage pour S. Tout nœud N de $def(S)$ est étiqueté par le même nom de clause dans S et dans $graft(S, N, S')$. Donc $const(S, reno_S) \subseteq const(S'', reno_{S''})$ et $\mathcal{P}_f(const(S, reno_S)) \subseteq \mathcal{P}_f(const(S'', reno_{S''}))$, où $\mathcal{P}_f(E)$ est l'ensemble des parties finies de E. S'il n'existe aucune partie finie de $const(S'', reno_{S''})$ rejetée alors il n'existe aucune partie finie de $const(S, reno_S)$ rejetée. D'où, si $graft(S, N, S')$, où $N \in undef(S)$, est un état alors S est un état.
De même pour monter que S' est un état. ■

2.4 Réponses (réponses positives)

Afin d'alléger les définitions, on suppose, jusqu'à la fin de ce chapitre, qu'un critère de rejet RC est fixé.

Définition 2.4.1 Squelette réponse, Store réponse
Un squelette réponse *pour le but $\leftarrow a$ est un état fini complet S pour $\leftarrow a$.*
Dans ce cas, $\mathrm{AC}(S, a)$ *est un* store réponse *pour le but $\leftarrow a$.*

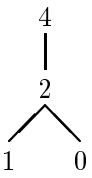
Étant donné l'importance donnée aux squelettes, nous dirons *réponse* à la place de *squelette réponse* pour alléger le texte.

On dit que S est une réponse si S est un état fini complet (i.e. il existe un atome a tel que S est une réponse pour le but $\leftarrow a$).

Lemme 2.4.2 *Soit S une réponse. Pour tout nœud N de dom_S, le squelette enraciné en N dans S est une réponse.*

Preuve. Soit S une réponse. Soit S' le squelette enraciné en un nœud $N \in dom_S$ dans S. Il est évident que S' est fini et complet. Si $reno_S$ est une fonction de renommage pour S alors $reno_{S'}$, défini par : pour tout nœud N' de $dom_{S'}$, $reno_{S'}(N') = reno_S(N \cdot N')$, est une fonction de renommage pour S'. Comme $const(S', reno_{S'}) \subseteq const(S, reno_S)$, S' n'est pas rejeté. C'est donc une réponse. ■

Exemple 2.4.1 Fibonacci
Le squelette de la figure 2.5 est un état. Comme il n'a pas de feuilles indéfinies, c'est une réponse. Donc $\exists x_1 \exists y_1 \exists x_2 \exists y_2 (x = 1 + 1 \wedge 1 < x \wedge x = x_1 + 1 \wedge x_1 = x_2 + 1 \wedge y = y_1 + y_2 \wedge x_1 = 1 \wedge y_1 = 1 \wedge x_2 = 0 \wedge y_2 = 0)$ est un store réponse au but $\leftarrow go2(x, y)$.
Conformément à l'interprétation des contraintes, il peut être simplifié en $x = 1 + 1 \wedge y = 1$.
La réponse enracinée en 01 fournit le store réponse $x = 0 \wedge y = 0$ pour le but $\leftarrow fib(x, y)$.



Cette réponse avec une fonction de renommage peut être représentée par :

$$go2(x,y) \leftarrow x = 1 + 1 \,\square\, fib(x,y)$$

$$fib(x,y) \leftarrow 1 < x \wedge x = x_1 + 1 \wedge x_1 = x_2 + 1 \wedge y = y_1 + y_2 \,\square\, fib(x_1,y_1) fib(x_2,y_2)$$

$$fib(x_1,y_1) \leftarrow x_1 = 1 \wedge y_1 = 1 \,\square\, \varepsilon \qquad\qquad fib(x_2,y_2) \leftarrow x_2 = 0 \wedge y_2 = 0 \,\square\, \varepsilon$$

Figure 2.5 : Une réponse pour le programme FIB

On comprend mieux maintenant le rôle primordial joué par les réponses aux buts "atomiques". Une réponse à un but général peut se définir comme une paire composée du store du but général et d'une forêt de réponses aux atomes présents dans le but général. La notion de paire (store, forêt de squelettes) rejetée se définit facilement en définissant la notion de fonction de renommage pour une telle paire, puis la notion de système de contraintes associé à cette paire.

Les réponses aux buts atomiques caractérisent complètement la sémantique opérationnelle du programme.

Nous retrouvons maintenant deux résultats exprimant la compositionalité ET des stores réponses dans notre cadre général.

Lemme 2.4.3 *Soit a un atome. Pour tout store réponse C au but $\leftarrow a$, il existe $u \in cn(P)$ tel que $clause(u)\theta = a \leftarrow C' \,\square\, a_1 \cdots a_n$, où θ est un renommage, et il existe n stores $C'_1, \ldots, C'_n$ tels que*

1. *pour tout $i = 1, \ldots, n$, C_i est un store réponse au but $\leftarrow a_i$;*
2. *$C = \exists_{-a}(C' \wedge \bigwedge_{i \in \{1,\ldots,n\}} C_i)$.*

Preuve. Si C est un store réponse au but $\leftarrow a$ alors il existe une réponse S telle que $C = \mathrm{AC}(S, a)$.

- si S est de profondeur 0 alors le résultat est immédiat.
- sinon, soit u la racine de S et θ un renommage tel que $clause(u)\theta = a \leftarrow C' \,\square\, a_1 \cdots a_n$. Soit S_i, $i = 1, \ldots, n$, la réponse enracinée en $i-1$ dans S. Montrons que $\mathrm{AC}(S, a) = \exists_{-a}(C' \wedge \bigwedge_{i \in \{1,\ldots,n\}} \mathrm{AC}(S_i, a_i))$.
  Pour tout $i = 1, \ldots, n$, soit $reno_{S_i}$ une fonction de renommage pour S_i telle que pour tout $j \in \{1, \ldots, n\} \setminus \{i\}$:
  - $head(reno_{S_i}(root(S_i))) = a_i$
  - $(var(const(S_i, reno_{S_i})) \setminus var(a_i)) \cap (var(a) \setminus var(a_i)) = \emptyset$
  - $var(const(S_i, reno_{S_i})) \cap var(const(S_j, reno_{S_j})) \subseteq var(a_i) \cap var(a_j)$

  La fonction $reno_S$ définie par, pour tout $N \in dom_S \setminus \{root(S)\}$, i.e. $N = (i-1) \cdot N_i$ où $i \in \{1, \ldots, n\}$, $reno_S(N) = reno_{S_i}(N_i)$ et $reno_S(root(S)) = a \leftarrow C' \,\square\, a_1 \cdots a_n$, est une fonction de renommage pour S et $\leftarrow a$: chaque $reno_{S_i}$ est une fonction de renommage pour S_i et d'après les conditions sur chaque $reno_{S_i}$

- pour tout nœud $N \in def(S)$, il existe un renommage θ tel que $reno_S(N) = clause(lab_S(N))\theta$;
- pour tout nœud $N \in def(S)$, si $reno_S(N) = a'' \leftarrow C'' \square a''_1 \cdots a''_{n''}$ alors pour tout $j \in \{1, \ldots, n''\}$, $head(reno_S(N \cdot (j-1))) = a''_j$;
- pour tous nœuds distincts N_1 et N_2 de dom_S, si $reno_S(N_1) = a'_1 \leftarrow C'_1 \square A'_1$ et $reno_S(N_2) = a'_2 \leftarrow C'_2 \square A'_2$ alors $(var(C'_1 \square A'_1) \setminus var(a'_1)) \cap (var(C'_2 \square A'_2) \setminus var(a'_2)) = \emptyset$.

Si $\{F_k\}_{k\in K}$ est une famille finie de stores et $\tilde{y}$ est un ensemble de variables tels que $\bigcap_{k\in K} var(F_k) = \tilde{x}$, $\tilde{x} \subseteq \tilde{y}$ et $(\tilde{y} \setminus \tilde{x}) \cap \bigcup_{k\in K} var(F_k) = \emptyset$ alors $\exists_{-\tilde{y}} \bigwedge_{k\in K} F_k = \bigwedge_{k\in K} \exists_{-\tilde{y}} F_k$, d'après les relations d'équivalences entre stores de la section 1.3.
Considérons l'ensemble $\{C', const(S_1, reno_{S_1}), \ldots, const(S_n, reno_{S_n})\}$ et l'ensemble de variables $\tilde{y} = var(a) \cup \bigcup_{i\in\{1,\ldots,n\}} var(a_i)$. Ils ont la propriété énoncée ci-dessus.
Or $\exists_{-a}(C' \wedge \bigwedge_{i\in\{1,\ldots,n\}} const(S_i, reno_{S_i})) = \exists_{-a}\exists_{-\tilde{y}}(C' \wedge \bigwedge_{i\in\{1,\ldots,n\}} const(S_i, reno_{S_i}))$, puisque $var(a) \subseteq \tilde{y}$.
Donc $\exists_{-a}(C' \wedge \bigwedge_{i\in\{1,\ldots,n\}} const(S_i, reno_{S_i})) = \exists_{-a}(\exists_{-\tilde{y}} C' \wedge \bigwedge_{i\in\{1,\ldots,n\}} \exists_{-\tilde{y}} const(S_i, reno_{S_i}))$.
Puisque $var(C') \cap \bigcap_{i\in\{1,\ldots,n\}} var(const(S_i, reno_{S_i})) \subseteq \tilde{y}$ alors
$\exists_{-a}(C' \wedge \bigwedge_{i\in\{1,\ldots,n\}} const(S_i, reno_{S_i})) = \exists_{-a}(C' \wedge \bigwedge_{i\in\{1,\ldots,n\}} \mathrm{AC}(S_i, a_i))$.
Or $C' \wedge \bigwedge_{i\in\{1,\ldots,n\}} const(S_i, reno_{S_i}) = const(S, reno_S)$.
Donc $\mathrm{AC}(S, a) = \exists_{-a}(C' \wedge \bigwedge_{i=1,\ldots,n} \mathrm{AC}(S_i, a_i))$.

■

Lemme 2.4.4 *Soit $u \in cn(P)$ tel que $clause(u) = a \leftarrow C \square a_1 \cdots a_n$ et n stores $C_1, \ldots, C_n$ tels que, pour tout $i = 1, \ldots, n$, C_i est un store réponse au but $\leftarrow a_i$.*
Si $(C \wedge \bigwedge_{i\in\{1,\ldots,n\}} C_i) \notin \mathrm{RC}$ alors $\exists_{-a}(C \wedge \bigwedge_{i\in\{1,\ldots,n\}} C_i)$ est un store réponse au but $\leftarrow a$.

Preuve. Similaire à la précédente. ■

Le lemme 2.4.2 peut être appelé lemme de compositionalité des réponses. Sa preuve est complètement triviale alors que les deux preuves précédentes sont plus techniques. C'est pourquoi il est bien plus simple de définir les réponses comme des squelettes plutôt que comme des stores (étant bien entendu que ce qui compte pour l'utilisateur est bien le store associé au squelette). L'intérêt du "tout squelette" est clair. Comme nous le voyons déjà, l'effort de reformulation permet d'éviter les problèmes de variables, renommages, etc. qui sont théoriquement inutiles puisque toute l'information est contenue dans le squelette. Cependant, nous pouvons retrouver l'ensemble des résultats classiques dans notre cadre, même quand ceux-ci sont techniques parce qu'il font référence aux stores (réponses) plutôt qu'aux squelettes (réponses).

2.5 Résolution

Nous décrivons maintenant un procédé qui fournit toutes les réponses à un but $\leftarrow a$.

Il est possible de générer tous les squelettes pour $\leftarrow a$, mais pour des raisons d'efficacité, nous ne souhaitons construire que des squelettes susceptibles de ne pas être rejetés, c'est-à-dire des états de calcul.

La construction descendante[5] des réponses à un but $\leftarrow p(x_1, \ldots, x_n)$ consiste à construire des états obtenus progressivement en greffant des squelettes aux feuilles indéfinies, jusqu'à obtenir des

[5] ou résolution "top-down"

états complets. On se limitera bien entendu à la construction d'états dont le symbole de prédicat associé à la racine est p.

Supposons que nous ayons déjà construit un état S pour p. Deux cas se présentent :

1. S est complet, alors S est une réponse ;
2. S est incomplet, alors S a au moins une feuille indéfinie. On progresse vers une réponse en greffant sur cette feuille indéfinie un autre squelette pour obtenir un nouvel état.

Le plus petit squelette que l'on peut greffer à une feuille indéfinie N d'un état S pour progresser vers une réponse est un $sq(u)$, où $u \in cn(P, lab_S(N))$, tel que $graft(S, N, sq(u))$ est un état. Et le plus petit squelette qui permet de démarrer cette construction est $sq(p)$.

Définition 2.5.1 Relation de transition entre états de calcul
Soit $\hookrightarrow$ la relation binaire sur l'ensemble des états, appelée relation de transition *entre états de calcul, définie par : $S \hookrightarrow S'$ si il existe une feuille $N \in undef(S)$ et un nom de clause $u \in cn(P, lab_S(N))$ tel que $S' = graft(S, N, sq(u))$.*

$\hookrightarrow$ définit un système de transition entre états de calcul. On dit que $S' = graft(S, N, sq(u))$ *dérive* de S (par la feuille N) si $S \hookrightarrow S'$.

2.5.1 dérivation SLD (calcul positif)

La *résolution SLD*[6] est une résolution descendante qui, étant donné un symbole de prédicat p, construit une suite d'états de la manière suivante :

1. le premier état est $sq(p)$;
2. à partir d'un état S,
   - si S est complet alors S est appelé une *réponse calculée* ;
   - sinon, l'état suivant est un état S' qui dérive de S, i.e. $S \hookrightarrow S'$.

Définition 2.5.2 État initial, État final, État succès, État échec
Soit S un état.

- *S est un* état initial *s'il existe $p \in \Pi_p$ tel que $S = sq(p)$.*
- *S est un* état final *s'il n'existe aucun état qui en dérive, i.e. pour tout état S' : $S \not\hookrightarrow S'$. Alors S est un* état succès *si S est complet, sinon c'est un* état échec.

On appelle *dérivation SLD* pour le but $\leftarrow p(x_1, \ldots, x_n)$ (ou plus simplement pour p) toute suite (finie ou infinie) d'états $S_1 \cdots S_i \cdots$ telle que $S_1 = sq(p)$ pour un $p \in \Pi_p$, chaque état S_i (sauf le premier) dérive du précédent (i.e. $S_{i-1} \hookrightarrow S_i$) et la suite est infinie ou le dernier état est un état final.

Une dérivation SLD succès (respectivement échec) est une dérivation SLD finie dont l'état final est un état succès (respectivement échec).

L'état final d'une dérivation SLD succès est une *réponse calculée*.

[6] La résolution SLD fut décrite par Kowalski dans [56] pour les programmes logiques purs. Le terme résolution SLD, dû à Apt et Van Emden [3] signifie résolution Linéaire avec fonction de Sélection pour clauses Définies.

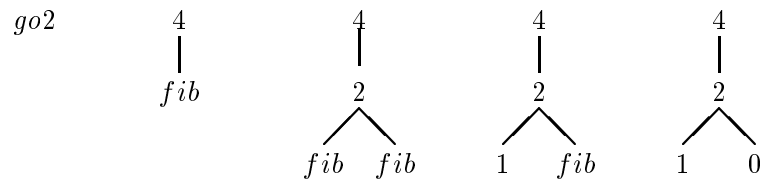


Figure 2.6 : Une dérivation SLD succès pour le but $\leftarrow go2(x, y)$

Exemple 2.5.1 Fibonacci

La dérivation SLD de la figure 2.6 est une dérivation SLD succès. Elle calcule la réponse de la figure 2.5.
Si l'on remplace l'étiquette de la feuille 00 (i.e. la feuille la plus à gauche) par 0 dans l'avant dernier état de la dérivation, alors cet état devient une feuille échec.

Lemme 2.5.3 *La résolution SLD a les deux propriétés suivantes :*

1. *toute réponse calculée est une réponse ;*
2. *elle calcule toutes les réponses.*

Preuve.

1. La première propriété est triviale : la résolution SLD ne construit que des états finis, donc s'ils sont complets, ils sont des réponses.
2. Pour prouver la seconde, considérons une réponse S pour p. Le parcours en largeur de S fournit une suite de nœud $N_0, \ldots, N_n$. On définit la suite d'états $S_0, S_1, \ldots, S_n$, telle que :
   - $S_0 = sq(p)$, où p est le symbole de prédicat associé à la racine de S ;
   - pour tout $i = 1, \ldots, n$, $def(S_i) = \{N_0, N_1, \ldots, N_i\}$, et, pour tout $N_j \in def(S_i)$, $lab_{S_i}(N_j) = lab_S(N_j)$.

   Il est facile de vérifier que $S_0, S_1, \ldots, S_m$ est une dérivation SLD et $S_m = S$. Par conséquent, S est une réponse calculée.

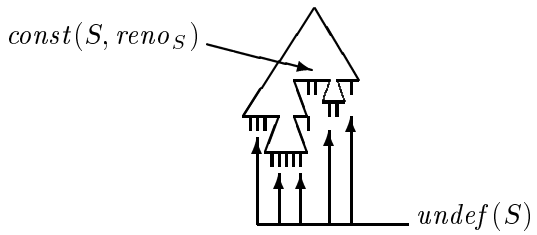
■

On peut distinguer deux niveaux d'indéterminisme dans la résolution SLD :

- le choix de la feuille indéfinie sur laquelle est greffé le squelette ;
- le choix du squelette qui sera greffé (en fait, le choix du nom de clause qui enracine le squelette greffé).

Le premier niveau d'indéterminisme correspond à la manière de construire une réponse alors que le second correspond au choix de la réponse qui sera construite.

Pour se rapprocher d'une résolution effective, nous allons réduire l'indéterminisme. Pour cela on fixe le choix de la feuille indéfinie ; il ne restera plus qu'à explorer toutes les solutions possibles engendrées par les noms de clauses possibles enracinant le squelette greffé sur la feuille choisie.



Au squelette S, ci-dessus, et à la fonction de renommage $reno_S$ pour S, on peut associer le but général :

$$\leftarrow const(S, reno_S) \square a_1 \cdots a_n$$

où chaque a_i est l'atome associé à un nœud indéfini de S par $reno_S$.

Figure 2.7 : But général associé à un squelette incomplet

Définition 2.5.4 Règle de calcul
On appelle règle de calcul *une fonction r qui, pour tout état incomplet S, fournit une feuille de $undef(S)$.*

> *Remarque.* Une règle de calcul est parfois définie comme une fonction qui à tout but (général) associe un atome du but. Cette définition est nettement moins générale que la nôtre, puisque le contexte ayant conduit au but courant n'est pas pris en compte. Pour deux squelettes différents représentant un même but (voir figure 2.7), la règle de calcul peut choisir deux atomes différents. ◇

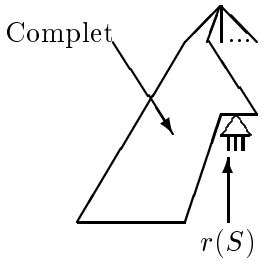
Étant donné une règle de calcul r, on définit la relation $\hookrightarrow_r$ incluse dans $\hookrightarrow$ comme suit : $S \hookrightarrow_r S'$ si S' dérive de S par la feuille $r(S)$, i.e. il existe un nom de clause $u \in cn(P, lab_S(r(S)))$ $(lab_S(r(s)) \in \Pi_p$ puisque $r(S) \in undef(S))$ tel que $S' = graft(S, r(S), sq(u))$. Une dérivation SLD selon la règle de calcul r se définit comme une dérivation SLD en remplaçant partout $\hookrightarrow$ par $\hookrightarrow_r$:

La *résolution SLD selon la règle de calcul* r est une résolution SLD qui, étant donné un but $\leftarrow p(x_1, \ldots, x_n)$, construit une suite d'états de la manière suivante :

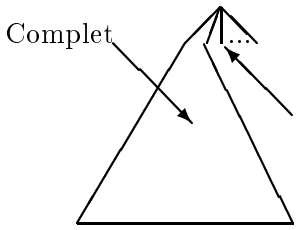
1. le premier état est $sq(p)$;
2. à partir d'un état S,
   - si S est complet alors S est appelé une *réponse calculée* selon r ;
   - sinon, l'état suivant est un état S' tel que $S \hookrightarrow_r S'$.

Lemme 2.5.5 *Les réponses calculées sont indépendantes de la règle de calcul.*

> *Preuve.* Soit S une réponse et r une règle de calcul. Considérons la dérivation SLD selon r définie par :
>
> - le premier état est $S_0 = sq(p)$, où p est le symbole de prédicat associé à la racine de S ;



Feuille indéfinie la plus à gauche
elle est aussi la plus profonde



Feuille indéfinie la plus à gauche
elle est aussi la plus profonde

Figure 2.8 : Règle de calcul standard

- si S_i est un état de la dérivation SLD alors l'état suivant est

$$S_{i+1} = \mathit{graft}(S_i, r(S_i), \mathit{sq}(\mathit{lab}_S(r(S_i))))$$

.

Cette dérivation SLD est finie et son état final est S (chaque S_i est un état d'après le lemme 2.3.2). ■

Une dérivation SLD $S_1 S_2 \cdots S_i \cdots$ selon r est *équitable* si elle est finie ou si, pour tout état S_i de la dérivation, pour toute feuille $N \in \mathit{undef}(S_i)$, il existe un état S_j de la dérivation $(i \leq j)$ où $N = r(S_j)$ est la feuille choisie. Une dérivation SLD est donc équitable si elle est finie ou toute feuille indéfinie d'un de ses états, est définie dans un autre état.

Une règle de calcul r est *équitable* si toute dérivation SLD selon r est équitable.

Une dérivation SLD $S_1 S_2 \cdots S_i \cdots$ selon r est *sans co-routinage* si, pour tout état incomplet S_i de la dérivation, $r(S_i)$ est une feuille indéfinie de l'ensemble des feuilles indéfinies les plus profondes. Une dérivation SLD est donc sans co-routinage si, pour tout état S de la dérivation, si S' est un état de la dérivation tel que $r(S') \in \mathit{undef}(S) \setminus \{r(S)\}$ alors alors l'état enraciné en $r(S)$ dans S' est complet. On en déduit que, pour tout état S_i $(i \neq 1)$ de la dérivation, si $r(S_{i-1})$ a un fils dans S_i alors $r(S_i)$ est un fils de $r(S_{i-1})$.

Une règle de calcul r est *sans co-routinage* si toute dérivation SLD selon r est sans co-routinage.

La *règle de calcul standard* est la règle r qui pour tout état S choisit la feuille indéfinie "la plus à gauche" de S : $r(S) = N$ si pour tout $N' \in \mathit{undef}(S) \setminus \{N\}$, il existe un préfixe $M \cdot n$ de N et un préfixe $M \cdot n'$ de N' tel que $n < n'$.

La règle de calcul standard est sans co-routinage (Voir Fig. 2.8).

Exemple 2.5.2 Fibonacci

La dérivation succès de la figure 2.6 est sans co-routinage. Elle est aussi équitable, puisqu'elle est finie.
On peut remarquer qu'elle utilise la règle de calcul standard.

Il est évident que si le critère de rejet est défini à partir d'une fonction *infer* et d'une relation *consistent* toutes deux incrémentales et si le système présenté en section 2.1 est quick-checking alors

> (C,C') est un store réponse au but $\leftarrow a$ selon la sémantique de la section 2.1 si et seulement si il existe une réponse S au but $\leftarrow a$ et $infer(\emptyset, \mathrm{AC}(S, a)) = (C, C')$.

Rappelons que tous les systèmes cités sont quick-checking et que le comportement du solveur de contraintes de chacun d'entre eux peut être modélisé par une fonction *infer* et une relation *consistent* toutes deux incrémentales.

La comparaison de notre sémantique avec d'autres sémantiques basées sur une interprétation ou une théorie des contraintes sera faite dans le chapitre 3.

2.5.2 Arbre de recherche SLD (calcul négatif)

Définition 2.5.6 Relation de transition selon une règle de calcul pour un prédicat de programme *Soit r une règle de calcul et p un prédicat de programme. Soit $\hookrightarrow_r^p$ la relation incluse dans $\hookrightarrow_r$ définie par $S \hookrightarrow_r^p S'$ si et seulement si $S \hookrightarrow_r S'$, où S et S' sont des squelettes pour $p \in \Pi_p$.*

Lemme 2.5.7 *Soit r une règle de calcul. Soit dom_r^p l'ensemble d'états $dom_r^p = \{S \mid sq(p) \hookrightarrow_r^{p*} S\}$. Alors $(dom_r^p, \hookrightarrow_r^p)$ est un arbre. Il est appelé l'arbre SLD pour p (ou pour le but $\leftarrow p(x_1, \ldots, x_n)$) selon la règle de calcul r.*

> *Preuve.* Montrons que la relation $\hookrightarrow_r^p$ a les propriétés d'une relation de parenté sur dom_r^p :
>
> - $sq(p)$ est la racine de l'arbre, d'après la définition de dom_r^p ;
> - $(sq(p), sq(p)) \notin \hookrightarrow_r^p$, d'après la définition de $\hookrightarrow_r$;
> - pour tout $S \in dom_r^p$, différent de $sq(p)$, il existe exactement un $S' \in dom_r^p$, tel que $S' \hookrightarrow_r^p S$:
>   - S' existe, d'après la définition de dom_r^p ;
>   - il est unique : considérons deux dérivation SLD selon r pour p ayant un état $S \neq sq(p)$ en commun ; elle ont au moins un autre état en commun S'' (par exemple $sq(p)$), la feuille choisie en S'' est unique et dans les deux dérivations les squelettes greffés sont identiques, sinon le nœud $r(S'')$ aurait des étiquettes différentes dans les états qui dérivent de S'' dans chacune des deux dérivations et il serait impossible d'aboutir plus loin au même état S ; en itérant le processus, on constate que les états qui précèdent S dans chacune des deux dérivations sont identiques.
>
> ■

> *Remarque.* Notre définition des arbres SLD révèle la nature même de ces arbres. Elle montre qu'ils ne peuvent se définir qu'étant donné une règle de calcul.
> Les définitions classiques ne mettent pas en avant ce que sont ces arbres, elles ne précisent que la nature de leurs étiquettes.

Notons que les arbres SLD, dans notre reformulation, n'ont pas besoin d'être définis comme des arbres étiquetés. Cependant, afin de retrouver une définition plus classique, nous pourrions étiqueté un arbre SLD par des buts généraux : à tout nœud on peut associer un but général (voir figure 2.7). Mais à nouveau on serait confronté aux difficultés liées au renommage des variables. ◇

Exemple 2.5.3 Fibonacci

La figure 2.9 présente un arbre SLD pour le but $\leftarrow go1(x, y)$.
On peut remarquer qu'il y a deux réponses, et si l'on poursuit le dessin (selon une règle équitable) alors on trouve une autre réponse (le store réponse simplifié est $x = 1{+}1{+}1{+}1{+}1 \wedge y = 1{+}1{+}1{+}1{+}1$, on trouve des feuilles échecs et une branche infinie au centre). (voir aussi figure 2.10)

En résumé, les caractéristiques essentielles d'un arbre SLD T pour le but $\leftarrow p(x_1, \ldots, x_n)$ selon la règle de calcul r sont :

- ses nœuds sont des états pour p ;
- sa racine est $sq(p)$;
- un nœud complet n'a pas de fils, c'est un *nœud succès* ;
- pour tout nœud S incomplet de T, soit $N = r(S)$, les fils de N correspondent exactement aux états qui dérivent de S par la feuille N. Si S n'a pas de fils alors S est un *nœud échec.*

Remarque. Notre définition de la règle de calcul permet de décrire plus d'arbres SLD que la définition de règle de calcul s'appliquant à des buts (un but peut avoir plusieurs occurrences dans l'arbre SLD alors que les squelettes, qui en sont les nœuds, n'en ont qu'une seule). ◇

Une branche de l'arbre SLD $(dom_r^p, \hookrightarrow_r^p)$ *est exactement* une dérivation SLD (selon r) pour p.

Preuve. Voir les définitions de dérivation SLD (section 2.5.1) et de branche d'un arbre (section 1.1). ■

Une branche succès (respectivement échec) est une branche finie qui se termine par une feuille succès (respectivement échec). L'arbre SLD contient exactement trois types de branches :

1. les branches succès qui sont les dérivations SLD succès selon r ;
2. les branches échecs qui sont les dérivations SLD échecs selon r ;
3. les branches infinies qui sont les dérivations SLD infinies selon r.

Un arbre SLD $(dom_r^p, \hookrightarrow_r^p)$ est *équitable* (respectivement *sans co-routinage*, *standard*) si ses branches sont équitables (respectivement sans co-routinage, standard), i.e. la restriction de r à l'ensemble dom_r^p est équitable (respectivement sans co-routinage, standard). En particulier, tout arbre SLD fini est équitable.

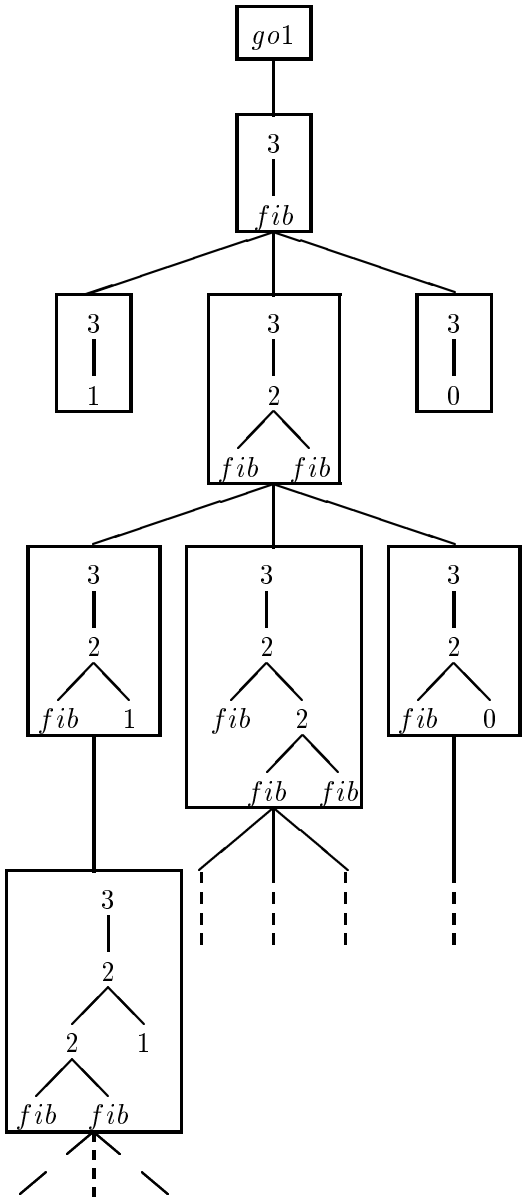


Figure 2.9 : Un arbre SLD pour le but $\leftarrow go1(x, y)$

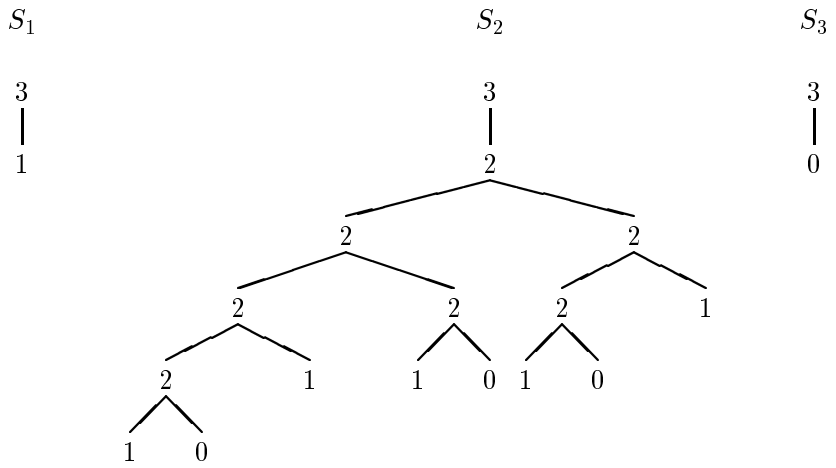


Figure 2.10 : Réponses de l'ensemble $success(go1(x, y))$

Lemme 2.5.8 *L'ensemble des feuilles succès d'un arbre SLD pour p selon la règle de calcul r est l'ensemble des réponses pour p.*

Preuve. Trivial, d'après les définitions. ■

Lemme 2.5.9 *Soit S un squelette et r une règle de calcul.*
Les propositions suivantes sont équivalentes :

1. *S est une réponse pour le but $\leftarrow a$.*
2. *S est une réponse calculée pour le but $\leftarrow a$.*
3. *S est une réponse calculée pour le but $\leftarrow a$ selon la règle de calcul r.*
4. *S est une feuille succès de l'arbre SLD pour $\leftarrow a$ selon la règle de calcul r.*

Preuve. L'équivalence entre *1*, *2* et *3* a déjà été montrée dans la section 2.5.1.
4 est équivalent à *3* car les branches de l'arbre SLD pour le but $\leftarrow a$ selon la règle de calcul r sont les dérivations SLD pour le but $\leftarrow a$ selon la règle de calcul r (voir la preuve du lemme 2.5.7). ■

Corollaire 2.5.10 *Les feuilles succès d'un arbre SLD sont indépendantes de la règle de calcul.*

On note $success(a)$ l'ensemble des réponses pour le but $\leftarrow a$.

Exemple 2.5.4 Fibonacci
L'ensemble $success(go1(x, y)) = \{S_1, S_2, S_3\}$ est constitué des trois réponses de la figure 2.10.
Les trois stores réponses pour le but $\leftarrow go1(x, y)$ sont respectivement (une fois simplifiés) : $x = 1 \wedge y = 1$, $x = 1+1+1+1+1 \wedge y = 1+1+1+1+1$ et $x = 0 \wedge y = 0$.
On peut remarquer que l'ensemble $success(fib(x, y))$ est infini.

Remarque. Il est possible d'orienter un arbre SLD en fixant un ordre sur les noms de clauses, i.e. un ordre total sur les $cn(P,p)$. Les fils d'un nœud S sont alors naturellement ordonnés par l'ordre sur les noms des clauses qui enracinent les squelettes greffés en $r(S)$. ◇

Définition 2.5.11 Arbre SLD d'échec fini
*Un arbre SLD est d'*échec fini *s'il est fini et s'il n'a aucune feuille succès.*

Remarque. La notion la plus intéressante ici est celle d'arbre SLD fini. La bonne vision d'arbre SLD d'échec fini n'est que comme un cas particulier d'arbre SLD fini. Il n'y a à peine plus à dire si le nombre de feuilles succès est nul que s'il est 1 ou 7 (même dans le cadre d'une sémantique logique). C'est pourquoi la sémantique déclarative sera défini par tous les arbres SLD finis. ◇

Nous avons vu qu'étant donné un but $\leftarrow a$, il existe une bijection entre les branches succès de deux arbres SLD pour $\leftarrow a$ (une branche succès de l'un correspond à une branche succès de l'autre si elles ont le même état final).

Il existe un autre type de branches qui sont en bijections. Il s'agit de branches infinies. Cette bijection s'exprime facilement quand les arbres SLD sont équitables.

Lemme 2.5.12 *Soit r une règle de calcul équitable. Les branches infinies de $(dom_r^p, \hookrightarrow_r^p)$ sont en bijection avec les états complets infinis pour p.*

Preuve. Soit $(dom_r^p, \hookrightarrow_r^p)$ un arbre SLD équitable.

- Considérons une branche infinie de $(dom_r^p, \hookrightarrow_r^p)$, c'est-à-dire une dérivation SLD infinie équitable $U = (S_1 S_2 \cdots S_i \cdots)$ pour p.

  Remarquons, d'une part, que si $N \in def(S_i) \cap def(S_j)$ alors $lab_{S_i}(N) = lab_{S_j}(N)$ et, d'autre part, que la dérivation étant équitable, toute feuille indéfinie d'un de ses états devient définie dans un autre.

  Soit S l'arbre sur le domaine d'arbre standard $dom_S = \bigcup_{S_i \in U} def(S_i)$ dont la fonction d'étiquetage lab_S est, pour tout $N \in dom_S$, $lab_S(N) = lab_{S_i}(N)$, où $N \in def(S_i)$.

  Pour tout nœud N de S, soit $a \leftarrow C \square a_1 \cdots a_n$ la clause de nom $lab_S(N)$, N a bien n fils et le symbole de prédicat associé à chacun est bien le symbole de prédicat de l'atome a_i correspondant. S est donc un squelette pour p et de plus il est complet (lab_S fait correspondre à tout nœud de dom_S un nom de clause). Il est aussi infini car U est infinie et chaque étape de dérivation défini un nœud qui était indéfini.

  Soit $reno_S$ une fonction de renommage pour S, comme nous l'avons vu dans la preuve du lemme 2.3.2, $reno_S$ est une fonction de renommage pour chacun des $S_i \in U$. Pour toute partie finie C de $const(S, reno_S)$, il existe un $S_i \in U$ tel que $C \subseteq const(S_i, reno_S)$, donc S est un état.

  S est un état complet infini pour p.

- Considérons un état S infini et complet pour p, et montrons qu'il est possible de construire un SLD-dérivation U, selon r, pour p, infinie et qui "calcule" S, c'est-à-dire une branche infinie de $(dom_r^p, \hookrightarrow_r^p)$.

  – Le premier squelette est $S_1 = sq(p)$.

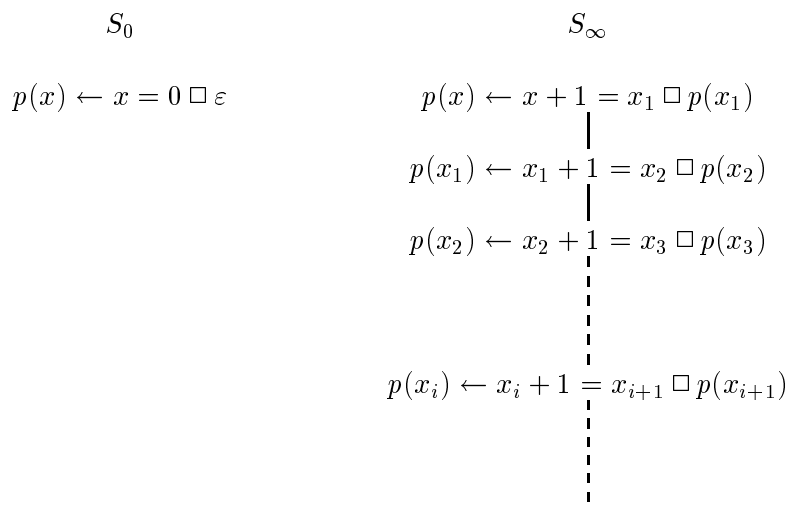


Figure 2.11 : État infini

– Supposons qu'on ait déjà construit la suite d'état $S_1 \cdots S_i$, préfixe de la branche U. $S_{i+1} = graft(S_i, r(S_i), sq(lab_S(r(S_i))))$ est un état incomplet fils de S_i dans $(dom_r^p, \hookrightarrow_r^p)$. C'est l'état qui suit S_i dans U.

On définit ainsi une branche infinie U de $(dom_r^p, \hookrightarrow_r^p)$ et comme elle est équitable elle calcule bien l'état complet S.

■

Remarque. La preuve précédente montre qu'à tout état complet infini pour p correspond une branche infinie de $(dom_r^p, \hookrightarrow_r^p)$, pour toute règle de calcul r (même non équitable). En fait, pour toute règle de calcul r, pour tout symbole de prédicat p

- à tout état fini complet pour p correspond une branche succès de $(dom_r^p, \hookrightarrow_r^p)$;
- à tout état infini complet pour p correspond une branche infinie de $(dom_r^p, \hookrightarrow_r^p)$;
- les autres branches de $(dom_r^p, \hookrightarrow_r^p)$, sont soit des branches échecs, soit des branches infinies qui ne calculent pas un état complet (des branches non équitables).

◇

Exemple 2.5.5 État infini

Soit P le programme en CLP($\mathcal{N}$) :

$$0 : p(x) \leftarrow x = 0 \,\square\, \varepsilon$$
$$1 : p(x) \leftarrow x + 1 = y \,\square\, p(y)$$

Tout arbre SLD pour le but $\leftarrow p(x)$ contient une branche succès qui correspond à la réponse S_0 et une branche infinie qui calcule l'état infini S_∞ (voir la figure 2.11).

Corollaire 2.5.13 *Soit r_1 et r_2 deux règles de calcul équitables. Les branches non échecs (i.e. branches succès et branches infinies) de $(dom_{r_1}^p, \hookrightarrow_{r_1}^p)$ et $(dom_{r_2}^p, \hookrightarrow_{r_2}^p)$ sont en bijection.*

Preuve. Elles sont en bijections avec les états complets pour p. ∎

Corollaire 2.5.14 *S'il existe un arbre SLD fini pour le but $\leftarrow a$ alors tout arbre SLD équitable pour le but $\leftarrow a$ est fini.*

Corollaire 2.5.15 *S'il existe un arbre SLD d'échec fini pour le but $\leftarrow a$ alors tout arbre SLD équitable pour le but $\leftarrow a$ est d'échec fini.*

Preuve. Soit $\leftarrow a$ un but ayant un arbre SLD d'échec fini.
Le corollaire 2.5.14 assure que tout arbre SLD équitable pour $\leftarrow a$ est fini : le cas où l'arbre SLD est d'échec fini est un cas particulier.
Le lemme 2.5.8 assure que tout arbre SLD pour $\leftarrow a$ n'a pas de feuille succès. ∎

Certains auteurs considèrent que la règle de calcul sélectionne à chaque étape soit un atome soit une contrainte basique du but courant (c'est le cas pour la sémantique classique présentée dans la section 2.1). Nous ne sélectionnons que des atomes dans le but courant. Le store de la clause choisie pour l'atome sélectionné est ajouté au store courant dès que cette clause est choisie. De ce point de vue, notre description de la sémantique opérationnelle n'est pas complètement fidèle au comportement des systèmes existants (excepté des langages comme Prolog III [80] qui se comportent exactement comme nous l'avons décrit). Mais on sait qu'au cours de la résolution si l'on commence par sélectionner toutes les contraintes basiques du but courant avant de sélectionner un atome alors on obtient toutes les réponses (les solveurs de contraintes ne bouclent pas). Cependant, pour être plus efficace, il est parfois utile de sélectionner un atome, qui va ajouter au store courant des contraintes fortes sur les variables, avant une contrainte basique, qui pourrait demander un travail coûteux au solveur de contraintes (voir l'exemple 2.5.6).

Malgré tout, ce qui est perdu en fidélité est sans importance pour ce que l'on veut modéliser. Il est évident que l'on peut modifier les définitions pour rendre compte de cette caractéristique de la plupart des systèmes, mais cela compliquerait le formalisme de façon tout à fait inutile ici.

Exemple 2.5.6 SENDMORY en CLP($\mathcal{FD}$)

Considérons le fameux puzzle de crypto-arithmétique : "SEND + MORE = MONEY". L'exercice consiste à assigner un chiffre différent à chaque lettre, pour que l'addition suivante soit vérifiée :

$$\begin{array}{cccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

On impose de plus que M et S soient tous deux différents de 0.

Le programme `send.pl` écrit en `clp(FD)` [35, 19] qui résout l'exercice est :

```
send(L) :- L = [S,E,N,D,M,O,R,Y],
           domaine(L),
           S#\=0,
           M#\=0,
                        1000*S + 100*E + 10*N + D
           +            1000*M + 100*O + 10*R + E
           #= 10000*M + 1000*O + 100*N + 10*E + Y,
           solution(L).
```

```
domaine(L) :- alldifferent(L), domain(L,0,9).

solution(L) :- labeling(L).
```

Voici les solutions fournies par `clp(FD)`

```
clp_fd                                    INRIA Rocquencourt - ChLoE Project
Version 2.21                                                Daniel Diaz - 1994

| ?- [send].

(0 ms) yes
| ?- send(L).

L = [9,5,6,7,1,0,8,2] ? ;

(10 ms) no
| ?-
```

On peut changer l'ordre des éléments dans le corps de la première clause et constater l'impact sur les performances.

Le même exercice est possible avec l'exemple de [21] qui consiste à assigner les dix chiffres 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 aux dix lettres D, G, R, O, E, N, B, A, L, T pour que l'addition suivante soit vérifiée :

$$\begin{array}{cccccccc} & D & O & N & A & L & D \\ + & G & E & R & A & L & D \\ \hline = & R & O & B & E & R & T \end{array}$$

2.6 $\vee$-réponses (réponses négatives)

Définition 2.6.1 $\vee$-réponse, $\vee$-store réponse
La $\vee$-réponse *pour le but* $\leftarrow a$ *est success*(a) *s'il existe une règle de calcul* r *telle que l'arbre SLD pour le but* $\leftarrow a$ *selon la règle de calcul* r *est fini. (La* $\vee$*-réponse pour le but* $\leftarrow a$ *si elle existe ne dépend pas de* r*).*
Dans ce cas, le $\vee$-store réponse *pour le but* $\leftarrow a$ *est le multi-ensemble* $\{\mathrm{AC}(S, a) \mid S \in success(a)\}$.

D'un point de vue opérationnel, seuls les calculs finis présentent un intérêt. C'est la raison pour laquelle nous avons défini les $\vee$-stores réponses sous la condition d'existence d'un arbre SLD fini. Dans la définition de store réponse nous n'avions aucune hypothèse sur l'existence d'un arbre SLD fini. Malgré tout nous considérions tout de même des calculs finis, mais il s'agissait d'un autre niveau de calcul : les SLD-dérivations.

Nous avons donc défini deux niveaux de réponses :

- les états finis complets (réponses positives) ;
- l'ensemble des feuilles succès des arbres SLD finis (réponses négatives).

Ces deux niveaux de réponses correspondent à deux niveaux de calculs :

- les dérivations SLD (calculs positifs) ;
- les arbres SLD (calculs négatifs).

En général, ce sont toujours les premiers qui sont considérés. On peut tout de même noter que pour la sémantique opérationnelle de Prolog IV [81] et celle de Escher [63], la définition de réponse est identique à notre définition de $\vee$-store réponse.

Exemple 2.6.1 Fibonacci
Il n'existe pas de $\vee$-réponse au but $\leftarrow go1(x, y)$: l'ensemble $success(go1(x, y))$ est fini (voir exemple 2.5.4), mais il n'existe pas d'arbre SLD fini pour ce but (voir l'exemple 2.5.3).
La $\vee$-réponse pour le but $\leftarrow go2(x, y)$ ne contient que la réponse de la figure 2.5.
Enfin, il n'existe pas de $\vee$-réponse pour le but $\leftarrow fib(x, y)$ puisque l'ensemble $success(fib(x, y))$ est infini.

2.7 Ensembles succès

Nous définissons maintenant deux ensembles succès qui caractérisent les deux niveaux de sémantique opérationnelle présentés. L'*ensemble succès positif* $\mathrm{SS}(P)$, appelé simplement *l'ensemble succès*, qui regroupe les stores réponses aux buts. L'*ensemble succès négatif* $\mathrm{SS}_\vee(P)$, appelé simplement *l'ensemble $\vee$-succès*, qui regroupe les $\vee$-stores réponses aux buts. Il est important de noter qu'aucun de ces deux ensembles ne se déduit de l'autre. Il est seulement possible de déduire un sous-ensemble de $\mathrm{SS}(P)$ à partir de $\mathrm{SS}_\vee(P)$.

Définition 2.7.1 Ensembles succès

- $\mathrm{SS}(P, a) = \{\mathrm{AC}(S, a) \mid S \in success(a)\}$
- $\mathrm{SS}(P) = \bigcup_{a \in \mathrm{ATOM}} \{a \leftarrow C \mid C \in \mathrm{SS}(P, a)\}$
- $\mathrm{SS}_\vee(P) = \{a \rightarrow \mathrm{SS}(P, a) \mid a \in \mathrm{ATOM}$, *il existe une $\vee$-réponse au but* $\leftarrow a\}$

  Remarque. $a \rightarrow R \in \mathrm{SS}_\vee(P)$ si et seulement si R est l'ensemble des stores du $\vee$-stores réponse pour $\leftarrow a$.
  On remarque aussi que R est un ensemble fini de stores qui peut être vu comme la disjonction de ses éléments.
  $\mathrm{SS}_\vee(P)$ est un ensemble d'atomes couverts ; $\mathrm{SS}(P)$ est un ensemble d'atomes contraints.
  $\diamond$

Soit P' le programme qui contient une clause pour chaque réponse du programme P. Si S est une réponse du programme P alors la clause qui correspond à S dans P' est $head(reno_S(root(S))) \leftarrow const(S, reno_S) \,\square\, \varepsilon$, où $reno_S$ est une fonction de renommage pour S. Le nom de cette clause de P' peut être par exemple S.

On constate que $\mathrm{SS}(P) = \mathrm{SS}(P')$. De plus, la multiplicité d'un store réponse à un but $\leftarrow a$ dans P et P' est identique. En revanche, les $\vee$-stores réponses ne sont pas les mêmes (c'est la différence entre l'existence d'un arbre SLD fini et l'existence d'un nombre fini de réponses). P' rend compte de la sémantique positive de P, mais pas de sa sémantique négative. On pourrait définir un programme P'' à partir des $\vee$-réponses du programme P. Alors P'' rendrait compte de la sémantique négative de P mais pas de sa sémantique positive.

Remarque. En fait P' n'est pas exactement un programme : il peut contenir une infinité de clauses pour la définition d'un symbole de prédicat. Par exemple, son complété (section 3.1) n'est pas défini (P'' serait quant à lui un programme). $\diamond$

Si l'on autorise les stores dans les corps de clauses (à la place des stores purs) alors $\mathrm{SS}(\{a \leftarrow C \,\square\, \varepsilon \mid a \leftarrow C \in \mathrm{SS}(P)\}) = \mathrm{SS}(P)$.

Exemple 2.7.1 Fibonacci

L'ensemble SS(FIB) contient, pour tout $(x, y) \in V^2$:

$$
\begin{array}{l}
go1(x,y) \leftarrow x = 0 \wedge y = 0 \\
go1(x,y) \leftarrow x = 1 \wedge y = 1 \\
go1(x,y) \leftarrow x = 1+1+1+1+1 \wedge y = 1+1+1+1+1 \\
go2(x,y) \leftarrow x = 1+1 \wedge y = 1 \\
fib(x,y) \leftarrow x = 0 \wedge y = 0 \\
fib(x,y) \leftarrow x = 1 \wedge y = 1 \\
fib(x,y) \leftarrow x = 1+1 \wedge y = 1 \\
fib(x,y) \leftarrow x = 1+1+1 \wedge y = 1+1 \\
fib(x,y) \leftarrow x = 1+1+1+1 \wedge y = 1+1+1 \\
fib(x,y) \leftarrow x = 1+1+1+1+1 \wedge y = 1+1+1+1+1 \\
\ldots
\end{array}
$$

L'ensemble $\mathrm{SS}_\vee(\mathrm{FIB}) = \{go2(x,y) \rightarrow \{x = 1+1 \wedge y = 1\} \mid (x,y) \in V^2\}$.

Chapitre 3

Sémantique déclarative

Du point de vue de l'utilisateur, un programme ne se résume pas seulement à son aspect syntaxique : les contraintes ont une signification sous-entendue. Si le chapitre 2 décrit bien la sémantique d'un programme par rapport à ce qui est calculé par le système (c'est la sémantique opérationnelle), ce chapitre présente la sémantique du programme telle qu'elle est vue par l'utilisateur (c'est la sémantique déclarative). Nous développons différentes sémantiques déclaratives basées sur différents sens donnés aux contraintes. Nous envisageons le cas où le critère de rejet est correct par rapport à une pré-interprétation (interprétation du langage des contraintes) ou une théorie pour les contraintes. Nous terminons en généralisant les notions de *couverture* des deux cas précédents, par une relation abstraite appelée *relation de couverture*. La relation de couverture qui abstrait les notions de conséquences est le pendant du critère de rejet qui abstrait les notions de satisfiabilité.

Dans ce chapitre, une clause est vue comme une formule du langage du premier ordre construit sur $(V, \Sigma, \Pi_p \cup \Pi_c)$: pour la clause $a \leftarrow C \,\square\, a_1 \cdots a_n$, on considère la formule $a \leftarrow C \wedge a_1 \wedge \cdots \wedge a_n$. Pour ne pas alourdir les notations, on confond la formule et la clause.

On suppose qu'un programme P est fixé.

3.1 Complété du programme

Nous nous intéressons dans cette section à des ensembles de formules qui jouent un rôle particulièrement important pour la sémantique déclarative des programmes ; notamment quand le critère de rejet est correct pour une pré-interprétation $\mathcal{D}$ ou une théorie $\mathcal{T}$.

Pour tout prédicat de programme $p \in \Pi_p$, on définit

$$\mathrm{IF}(P,p) = a \leftarrow \bigvee_{u \in cn(P,p)} \exists_{-\tilde{x}^u} (C^u \wedge a_1^u \wedge \cdots \wedge a_{n_u}^u)$$

où pour chaque $u \in cn(P,p)$: $a \leftarrow C^u \,\square\, a_1^u \cdots a_{n_u}^u$ est un renommage de la clause de nom u et $\tilde{x}^u = var(C^u \,\square\, a_1^u \cdots a_{n_u}^u) \setminus var(a)$.

Si la définition de p est vide alors la disjonction est indexée par $\emptyset$ et se résume donc à l'élément neutre de $\vee$ qui est *false*.

> *Remarque.* La formule $\mathrm{IF}(P,p)$ est définie à un renommage près ; pour lever cette ambiguïté on peut imposer des restrictions quant à la forme des clauses d'un programme : toutes les têtes des clauses d'une même définition sont identiques, dans ce cas $\mathrm{IF}(P,p)$ se définit directement à partir des clauses du programme sans renommage. ◇

Pour tout prédicat $p \in \Pi_p$, on définit $\mathrm{FI}(P,p)$ la formule obtenue en inversant le sens de l'implication dans $\mathrm{IF}(P,p)$ et on définit $\mathrm{IFF}(P,p) = \mathrm{IF}(P,p) \wedge \mathrm{FI}(P,p)$. $\mathrm{IFF}(P,p)$ est obtenu aussi en remplaçant l'implication de $\mathrm{IF}(P)$ par une équivalence.

On définit, enfin, les trois ensembles de formules :

- $\mathrm{IF}(P) = \{\mathrm{IF}(P,p) \mid p \in \Pi_p\}$;
- $\mathrm{FI}(P) = \{\mathrm{FI}(P,p) \mid p \in \Pi_p\}$;
- $\mathrm{IFF}(P) = \{\mathrm{IFF}(P,p) \mid p \in \Pi_p\}$.

$\mathrm{IFF}(P)$ est appelé le *complété du programme* P, il est souvent noté P^*. Le complété du programme a été décrit à l'origine dans [17] pour les programmes logiques purs, dans ce cadre, on l'appelle parfois *complétion de Clark* du programme. Notons que nous n'ajoutons pas à $\mathrm{IFF}(P)$ les axiomes de l'égalité de Clark CET (Clark Equality Theory). En effet, nous n'utilisons pas de prédicat $=$ supplémentaire pour définir le complété. Le rôle joué par CET en programmation logique est remplacé ici par une théorie des contraintes.

Exemple 3.1.1 Fibonacci

Le complété du programme FIB est :

$$\begin{array}{ll} fib(x,y) \leftrightarrow & (x = 0 \wedge y = 0) \vee (x = 1 \wedge y = 1) \vee \\ & \exists x_1 \exists y_1 \exists x_2 \exists y_2 (1 < x \wedge x = x_1 + 1 \wedge x_1 = x_2 + 1 \wedge y = y_1 + y_2 \wedge \\ & \qquad fib(x_1, y_1) \wedge fib(x_2, y_2)) \\ go1(x,y) \leftrightarrow & x = y \wedge fib(x,y) \\ go2(x,y) \leftrightarrow & x = 1 + 1 \wedge fib(x,y) \end{array}$$

Remarque. $\mathrm{IF}(P)$, $\mathrm{FI}(P)$ et $\mathrm{IFF}(P)$ ne sont pas des programmes, ce sont des ensembles de formules construites sur $(V, \Sigma, \Pi_c \cup \Pi_p)$. ◇

Les modèles de $\mathrm{IF}(P)$ sont mis en relation avec les ensembles clos de l'opérateur associé au programme, les modèles de $\mathrm{FI}(P)$ sont mis en relation avec ses ensembles supportés et les modèles de $\mathrm{IFF}(P)$ sont mis en relation avec ses points fixes.

Nous envisageons plusieurs définitions d'opérateur et de sémantique déclarative associés au programme.

3.2 Selon une pré-interprétation $\mathcal{D}$

On s'intéresse à la sémantique déclarative des programmes quand la signification des contraintes est basée sur un domaine (par exemple : les termes, les sous-ensembles finis de $\mathbb{N}$, les réels, les ensembles, les arbres infinis, les lambda-termes, les booléens, etc.). La sémantique des contraintes est déterminée par une pré-interprétation $\mathcal{D}$, c'est-à-dire une interprétation du langage construit sur (V, Σ, Π_c). La satisfiabilité d'un store est déterminée par $\mathcal{D}$.

On étudie les modèles de $\mathrm{IF}(P)$, $\mathrm{FI}(P)$ et $\mathrm{IFF}(P)$ qui "prolongent" $\mathcal{D}$.

Dans la section 3.2.1 on examine le lien entre les "$\mathcal{D}$-conséquences" du programme et ce qui est calculé par le système quand le critère de rejet est correct pour $\mathcal{D}$.

Définition 3.2.1 Pré-interprétation

Une pré-interprétation $\mathcal{D}$ *est constituée :*

- *d'un ensemble non vide* $\mathbb{D}$, *l'ensemble des* éléments du domaine ;
- *pour tout symbole de fonction* f *d'arité* n *de* Σ, *d'une application* $f_{\mathcal{D}}$ *de* $\mathbb{D}^n$ *dans* $\mathbb{D}$; *en particulier, à chaque constante* f *est associé un élément* $f_{\mathcal{D}}$ *de* $\mathbb{D}$;
- *pour tout symbole de prédicat de contrainte* p *d'arité* n *de* Π_c, *d'une relation* $p_{\mathcal{D}}$ *sur* $\mathbb{D}^n$; *en particulier, si* $n = 0$ *alors* $\mathcal{D}$ *associe à* p *une valeur de vérité de l'ensemble* $\{true, false\}$.

On remarque que $\mathcal{D}$ ne précise pas l'interprétation des symboles de prédicat de programme, c'est pourquoi on l'appelle pré-interprétation ou *interprétation des contraintes*. Le domaine $\mathbb{D}$ d'une pré-interprétation $\mathcal{D}$ est appelé *domaine de contraintes*.

Cependant, pour l'interprétation des prédicats de programme, on considère des "pseudo-atomes" appelés $\mathcal{D}$-*atomes* (ou atomes valués) qui sont des $n+1$-uplets de la forme $\langle p, d_1, \ldots, d_n \rangle$, où p est un symbole de prédicat de programme de Π_p d'arité n et les d_i sont des éléments du domaine $\mathbb{D}$. Pour des raisons évidentes, un $\mathcal{D}$-atome est noté $p(d_1, \ldots, d_n)$.

On appelle $\mathcal{D}$-base l'ensemble de tous les $\mathcal{D}$-atomes, c'est-à-dire, l'ensemble $\bigcup_{n \in \mathbb{N}} \Pi_{p,n} \times \mathbb{D}^n$, où $\Pi_{p,n}$ est le sous-ensemble de Π_p des symboles de prédicat d'arité n.

Exemple 3.2.1 Fibonacci

Une pré-interprétation pour le programme FIB est $\mathcal{N}$, dont le domaine est $\mathbb{N}$, où $0_{\mathcal{N}}$ et $1_{\mathcal{N}}$ sont les entiers 0 et 1, $+_{\mathcal{N}}$ est l'addition usuelle, $<_{\mathcal{N}}$ et $=_{\mathcal{N}}$ sont respectivement la relation d'ordre usuelle et l'égalité dans $\mathbb{N}$.
$fib(3,7)$ est un $\mathcal{N}$-atome et la $\mathcal{N}$-base est l'ensemble $\{fib(n_1, n_2) \mid (n_1, n_2) \in \mathbb{N}^2\} \cup \{go1(n_1, n_2) \mid (n_1, n_2) \in \mathbb{N}^2\} \cup \{go2(n_1, n_2) \mid (n_1, n_2) \in \mathbb{N}^2\}$.

Définition 3.2.2 $\mathcal{D}$-clause

Une $\mathcal{D}$-clause *est un triplet* $a_{\mathcal{D}} \leftarrow z \,\square\, A_{\mathcal{D}}$, *où* $a_{\mathcal{D}}$ *est un* $\mathcal{D}$-*atome,* $z \in \{true, false\}$ *est une valeur de vérité et* $A_{\mathcal{D}}$ *est une suite finie de* $\mathcal{D}$-*atomes.*

Exemple 3.2.2 Fibonacci

Par exemple, $fib(2,1) \leftarrow true \,\square\, fib(1,1)fib(0,0)$, $fib(2,10) \leftarrow false \,\square\, fib(1,2)go1(2,2)go2(4,1)$ et $fib(2,10) \leftarrow true \,\square\, fib(1,1)fib(0,0)$ sont des $\mathcal{N}$-clauses.

Définition 3.2.3 Valuation

Une valuation v *est une application de* V *dans* $\mathbb{D}$ *qui s'étend naturellement en une application* $v_{\mathcal{D}}$:

- *des termes vers* $\mathbb{D}$:
  - $v_{\mathcal{D}}(t) = v(t)$ *si* t *est une variable,*
  - $v_{\mathcal{D}}(t) = f_{\mathcal{D}}(v_{\mathcal{D}}(t_1), \ldots, v_{\mathcal{D}}(t_n))$ *si* t *est le terme* $f(t_1, \ldots, t_n)$;
- *des contraintes atomiques vers* $\{true, false\}$: $v_{\mathcal{D}}(p(t_1, \ldots, t_n)) = p_{\mathcal{D}}(v_{\mathcal{D}}(t_1), \ldots, v_{\mathcal{D}}(t_n))$, *où* p *d'arité* n *est un élément de* Π_c *et les* t_i *sont des termes ; qui s'étend elle-même en une application des stores vers* $\{true, false\}$;
- *des atomes vers la* $\mathcal{D}$-*base :* $v_{\mathcal{D}}(p(t_1, \ldots, t_n)) = p(v_{\mathcal{D}}(t_1), \ldots, v_{\mathcal{D}}(t_n))$, *où* $p \in \Pi_p$ *est d'arité* n *et les* t_i *sont des termes ;*
- *des clauses vers les* $\mathcal{D}$-*clauses :* $v_{\mathcal{D}}(a \leftarrow C \,\square\, a_1 \cdots a_n) = v_{\mathcal{D}}(a) \leftarrow v_{\mathcal{D}}(C) \,\square\, v_{\mathcal{D}}(a_1) \cdots v_{\mathcal{D}}(a_n)$.

Un store C est *satisfiable* s'il existe une valuation v telle que $v_{\mathcal{D}}(C) = \mathit{true}$. Dans ce cas, on dit que v est une *solution* de C.

Définition 3.2.4 Interprétation dans $\mathbb{D}$
Une interprétation I *dans* $\mathbb{D}$ *qui prolonge* $\mathcal{D}$ *associe à tout symbole de prédicat de programme* p *de* Π_p*, d'arité* n*, une relation* p_I *sur* $\mathbb{D}^n$.

Définition 3.2.5 $\mathcal{D}$-interprétation
Une $\mathcal{D}$-interprétation I *est une partie de la* $\mathcal{D}$*-base.*

On peut identifier toute $\mathcal{D}$-interprétation I à une interprétation dans $\mathbb{D}$ qui prolonge $\mathcal{D}$ car elle définit pour tout $p \in \Pi_p$, d'arité n, la relation p_I d'arité n : $\{(d_1, \ldots, d_n) \mid p(d_1, \ldots, d_n) \in I\}$. Inversement, de toute interprétation I dans $\mathbb{D}$, on déduit la $\mathcal{D}$-interprétation $\{p(d_1, \ldots, d_n) \mid p \in \Pi_p, (d_1, \ldots, d_n) \in p_I\}$.

Étant donné une $\mathcal{D}$-interprétation I, toute valuation v s'étend en une application v_I :

- des termes vers $\mathbb{D}$: $v_I(t) = v_{\mathcal{D}}(t)$;
- des stores vers $\{\mathit{true}, \mathit{false}\}$: $v_I(C) = v_{\mathcal{D}}(C)$;
- des atomes vers $\{\mathit{true}, \mathit{false}\}$: $v_I(a) = \mathit{true}$ si et seulement si $v_{\mathcal{D}}(a) \in I$;
- des clauses vers $\{\mathit{true}, \mathit{false}\}$: $v_I(a \leftarrow C \,\square\, a_1 \cdots a_n) = \mathit{true}$ si et seulement si l'implication $v_I(C) \wedge v_I(a_1) \wedge \cdots \wedge v_I(a_n) \rightarrow v_I(a)$ est vraie.

Une $\mathcal{D}$-interprétation I est *modèle* de la clause $a \leftarrow C \,\square\, A$ si pour toute valuation v dans $\mathbb{D}$: $v_I(a \leftarrow C \,\square\, A) = \mathit{true}$.

Pour toute expression E du langage, E est *satisfiable* dans l'interprétation I, s'il existe une valuation v telle que $v_I(E) = \mathit{true}$. Dans ce cas, on dit que v est solution de E (dans l'interprétation I).

Définition 3.2.6 $\mathcal{D}$-modèle d'un programme
Un $\mathcal{D}$-modèle *du programme* P *est une* $\mathcal{D}$*-interprétation* I *modèle de chacune des clauses de* P.

Exemple 3.2.3 Fibonacci
La $\mathcal{N}$-interprétation

$$\begin{aligned} I_1 = \quad & \{go1(0,0), go1(1,1), go1(5,5), go2(2,1)\} \cup \\ & \{fib(0,0), fib(1,1), fib(2,1), fib(3,2), fib(4,3), fib(5,5), \ldots\} \end{aligned}$$

(i.e. l'ensemble des $fib(n_1, n_2)$ tel que n_2 est l'image de n_1 par la fonction de Fibonacci) est un $\mathcal{N}$-modèle du programme.
$I_2 = \{go1(n_1, n_2) \mid fib(n_1, n_2) \in I_1\} \cup I_1$ est aussi un $\mathcal{N}$-modèle du programme.
Toute $\mathcal{N}$-interprétation qui ne contient pas I_1 n'est pas $\mathcal{N}$-modèle du programme.

Remarque. Pour tout programme P, pour toute pré-interprétation $\mathcal{D}$, la $\mathcal{D}$-base entière est un $\mathcal{D}$-modèle de P. ◇

On remarque qu'une pré-interprétation $\mathcal{D}$ partitionne en deux l'ensemble des $\mathcal{D}$-clauses : celles pour lesquelles le store a pris la valeur *true* et celles pour lesquelles il a pris la valeur *false*. Soit $v_{\mathcal{D}}(a \leftarrow C \,\square\, A)$ une $\mathcal{D}$-clause et I une $\mathcal{D}$-interprétation

- si $v_{\mathcal{D}}(C) = \mathit{false}$ alors $v_I(a \leftarrow C \,\square\, A) = \mathit{true}$;
- si $v_{\mathcal{D}}(C) = \mathit{true}$ alors $v_I(a \leftarrow C \,\square\, A) = \mathit{true}$ si et seulement si $\{v_{\mathcal{D}}(a_i) \mid a_i \in A\} \subseteq I$ implique $v_{\mathcal{D}}(a) \in I$.

On remarque que si $v_{\mathcal{D}}(C) = \mathit{false}$, pour toute valuation v, alors toute $\mathcal{D}$-interprétation I est un $\mathcal{D}$-modèle de $a \leftarrow C \,\square\, A$.

On appelle *$\mathcal{D}$-règle* une paire, notée $a \leftarrow A$, où a est un $\mathcal{D}$-atome et A est une suite finie de $\mathcal{D}$-atomes. Les $\mathcal{D}$-règles sont des règles sur la $\mathcal{D}$-base au sens de la section 1.2.1.

À tout programme on va associer un ensemble de $\mathcal{D}$-règles. Cet ensemble détermine la sémantique déclarative point fixe selon $\mathcal{D}$ du programme. Nous étudions le lien entre cette sémantique et la sémantique logique selon $\mathcal{D}$ donnée par les $\mathcal{D}$-modèles de $\mathrm{IF}(P)$ (ou $\mathrm{FI}(P)$ ou $\mathrm{IFF}(P)$).

Définition 3.2.7 Système de $\mathcal{D}$-règles associé à une clause
Le système de $\mathcal{D}$-règles *associé à une clause* $a \leftarrow C \,\square\, a_1 \cdots a_n$ *est* $\Phi(\mathcal{D}, a \leftarrow C \,\square\, a_1 \cdots a_n) = \{v_{\mathcal{D}}(a) \leftarrow v_{\mathcal{D}}(a_1) \cdots v_{\mathcal{D}}(a_n) \mid v_{\mathcal{D}}(C) = \mathit{true}\}$.

> *Remarque.* Une $\mathcal{D}$-interprétation I est modèle de la clause R si et seulement si I est close par $T_{\Phi(\mathcal{D},R)}$. ◇

Définition 3.2.8 Système de $\mathcal{D}$-règles associé à un programme
Le système de $\mathcal{D}$-règles *associé à un programme* P *est* $\Phi(\mathcal{D}, P) = \bigcup_{R \in P} \Phi(\mathcal{D}, R)$.
Son opérateur associé $T_{\Phi(\mathcal{D},P)}$ *sera noté* $T_P^{\mathcal{D}}$.

Lemme 3.2.9 *Une $\mathcal{D}$-interprétation I est modèle de P si et seulement si I est close par $T_P^{\mathcal{D}}$.*

> *Preuve.* Conséquence de la remarque précédente et de $T_P^{\mathcal{D}}(I) = \bigcup_{R \in P} T_{\Phi(\mathcal{D},R)}(I)$. ■

Lemme 3.2.10 *Le programme P a un plus petit $\mathcal{D}$-modèle qui est $pppf(T_P^{\mathcal{D}})$.*

> *Preuve.* Les $\mathcal{D}$-modèles de P sont les ensembles clos par $T_P^{\mathcal{D}}$. ■

On a utilisé ici directement les résultats de la section 1.2 sur les définitions inductives.

Exemple 3.2.4 Fibonacci

Le système de $\mathcal{N}$-règles associé au programme FIB est

$$\begin{array}{l}\{fib(0,0) \leftarrow \varepsilon, fib(1,1) \leftarrow \varepsilon\} \cup \\ \{fib(n_1,n_2) \leftarrow fib(n_3,n_4)fib(n_5,n_6) \mid (n_1,n_2,n_3,n_4,n_5,n_6) \in \mathbb{N}^6, \\ \ n_1 > 1, n_1 = n_3 + 1, n_1 = n_5 + 2, n_2 = n_4 + n_6\} \cup \\ \{go1(n,n) \leftarrow fib(n,n) \mid n \in \mathbb{N}\} \cup \\ \{go2(2,n) \leftarrow fib(2,n) \mid n \in \mathbb{N}\}\end{array}$$

Par exemple, il contient la $\mathcal{N}$-règle $fib(2,10) \leftarrow fib(1,4)fib(0,6)$.
Le plus petit $\mathcal{N}$-modèle de $\Phi(\mathcal{N}, \mathrm{FIB})$ est l'interprétation I_1 de l'exemple 3.2.3.

Lemme 3.2.11 *La $\mathcal{D}$-interprétation I est modèle de* $\mathrm{IF}(P)$ *si et seulement si I est modèle de P.*

Preuve. Ici, P est vu comme un ensemble de formules. Les formules de P peuvent se regrouper par paquets de clauses (l'ensemble des clauses de la définitions d'un prédicat de programme p). Chaque paquet peut s'identifier à une conjonction de la forme $\bigwedge_{u\in cn(P,p)}(a \leftarrow C_u \,\square\, A_u)$ (en renommant correctement les clauses du paquet). Chaque conjonction $\bigwedge_{u\in cn(P,p)}(a \leftarrow C_u \,\square\, A_u)$ est logiquement équivalente à $a \leftarrow \bigvee_{u\in cn(P,p)}(C_u \,\square\, A_u)$ qui est logiquement équivalente à $a \leftarrow \bigvee_{u\in cn(P,p)} \exists_{-var(a)}(C_u \,\square\, A_u)$, i.e. à $\mathrm{IF}(P,p)$. ∎

Corollaire 3.2.12 *Les propositions suivantes sont équivalentes :*

1. *I est $\mathcal{D}$-modèle de P ;*
2. *I est $\mathcal{D}$-modèle de $\mathrm{IF}(P)$;*
3. *I est close par $T_P^{\mathcal{D}}$.*

Une formule F est une $\mathcal{D}$-*conséquence* du programme P, noté $P \models_{\mathcal{D}} F$, si F est vraie dans tous les $\mathcal{D}$-modèles de P. C'est-à-dire, pour tout $\mathcal{D}$-modèle I, pour toute valuation v, $v_I(F) = true$.

On note, plus généralement, $E \models_{\mathcal{D}} F$ si F est vraie dans tout $\mathcal{D}$-modèle de E.

Corollaire 3.2.13 *Le plus petit $\mathcal{D}$-modèle de $\mathrm{IF}(P)$ est $pppf(T_P^{\mathcal{D}})$ qui est aussi le plus petit I tel que $T_P^{\mathcal{D}}(I) \subseteq I$.*
Pour tout atome a, $P \models_{\mathcal{D}} a$ si et seulement si, pour toute valuation v, $v_{\mathcal{D}}(a) \in pppf(T_P^{\mathcal{D}})$.

Le plus grand $\mathcal{D}$-modèle de $\mathrm{IF}(P)$ est la $\mathcal{D}$-base. En revanche, le plus grand $\mathcal{D}$-modèle de $\mathrm{FI}(P)$ (nous montrons qu'il est toujours défini) est très intéressant. Étudions maintenant les $\mathcal{D}$-modèles de $\mathrm{FI}(P)$.

Lemme 3.2.14 *I est un $\mathcal{D}$-modèle de $\mathrm{FI}(P)$ si et seulement si I est supporté par $T_P^{\mathcal{D}}$.*

Preuve. On reprend les notations de la section 3.1 pour $\mathrm{FI}(P)$.

- Soit I un $\mathcal{D}$-modèle de $\mathrm{FI}(P)$. Montrons que pour tout $\mathcal{D}$-atome $p(d_1,\ldots,d_n)$: si $p(d_1,\ldots,d_n) \in I$ alors $p(d_1,\ldots,d_n) \in T_P^{\mathcal{D}}(I)$.
  I est $\mathcal{D}$-modèle de $\mathrm{FI}(P,p) = p(x_1,\ldots,x_n) \rightarrow \bigvee_{u\in cn(P,p)}(\exists_{-\tilde{x}_u}(C^u \wedge a_1^u \wedge \cdots \wedge a_{n_u}^u)$, donc, pour toute valuation v tel que $v_{\mathcal{D}}(p(x_1,\ldots,x_n)) = p(d_1,\ldots,d_n) \in I$, il existe $u \in cn(P,p)$ tel que $v_I(\exists_{-\tilde{x}_u}(C^u \wedge a_1^u \wedge \cdots \wedge a_{n_u}^u)) = true$, donc il existe une valuation v' tel que $v'(x_i) = v(x_i)$ et $v'_I(C^u \wedge a_1^u \wedge \cdots \wedge a_{n_u}^u) = true$. Par conséquent : d'une part, tous les $v'_{\mathcal{D}}(a_i^u)$ sont dans I, d'autre part, la $\mathcal{D}$-règle $v'_{\mathcal{D}}(p(x_1,\ldots,x_n)) \leftarrow v'_{\mathcal{D}}(a_1^u)\cdots v'_{\mathcal{D}}(a_{n_u}^u)$ appartient à $\Phi(\mathcal{D},P)$ ($v'_{\mathcal{D}}(C_u) = true$). On en déduit que $v'_{\mathcal{D}}(p(x_1,\ldots,x_n)) = p(d_1,\ldots,d_n)$ est dans $T_P^{\mathcal{D}}(I)$.
- Soit I supporté par $T_P^{\mathcal{D}}$. Montrons que I est $\mathcal{D}$-modèle de $\mathrm{FI}(P)$, i.e., pour tout $p \in \Pi_p$, I est $\mathcal{D}$-modèle de $\mathrm{FI}(P,p) = p(x_1,\ldots,x_n) \rightarrow \bigvee_{u\in cn(P,p)}(\exists_{-\tilde{x}_u}(C^u \wedge a_1^u \wedge \cdots \wedge a_{n_u}^u)$.
  Pour toute valuation v :
  - soit $v_{\mathcal{D}}(p(x_1,\ldots,x_n)) \notin I$. Alors $\mathrm{FI}(P,p)$ est trivialement satisfaite par v ;
  - soit $v_{\mathcal{D}}(p(x_1,\ldots,x_n)) \in I$. Alors $v_{\mathcal{D}}(p(x_1,\ldots,x_n)) \in T_P^{\mathcal{D}}(I)$, donc, il existe une $\mathcal{D}$-règle issue de $clause(u)$, $u \in cn((,P),p)$: $v'_{\mathcal{D}}(p(x_1,\ldots,x_n)) \leftarrow v'_{\mathcal{D}}(a_1^u)\cdots v'_{\mathcal{D}}(a_{n_u}^u) \in \Phi(\mathcal{D},P)$ telle que $v'_{\mathcal{D}}(p(x_1,\ldots,x_n)) = v_{\mathcal{D}}(p(x_1,\ldots,x_n))$ et $\{v'_{\mathcal{D}}(a_1^u),\ldots,v'_{\mathcal{D}}(a_{n_u}^u)\} \subseteq I$. On peut de plus choisir v' telle que $v'_{\mathcal{D}}(C_u) = true$. Donc $\mathrm{FI}(P,p)$ est satisfaite par v'.

Par conséquent si I est supporté par $T_P^{\mathcal{D}}$ alors I est $\mathcal{D}$-modèle de chaque $\mathrm{FI}(P,p)$ donc de $\mathrm{FI}(P)$.

■

Corollaire 3.2.15 $\mathrm{FI}(P)$ *a un plus grand* $\mathcal{D}$*-modèle qui est* $pgpf(T_P^{\mathcal{D}})$*, c'est aussi le plus grand* I *tel que* $T_P^{\mathcal{D}}(I) \supseteq I$.

Remarque. Le plus petit $\mathcal{D}$-modèle de $\mathrm{FI}(P)$ est $\emptyset$. ◇

Nous avons caractérisé les $\mathcal{D}$-modèles de $\mathrm{IF}(P)$ et $\mathrm{FI}(P)$ en termes d'ensemble clos ou supporté par $T_P^{\mathcal{D}}$. Comme, pour tout $p \in \Pi_P$, $\mathrm{IFF}(P,p) = \mathrm{IF}(P,p) \wedge \mathrm{FI}(P,p)$, on peut caractériser les $\mathcal{D}$-modèles de $\mathrm{IFF}(P)$ par l'intersection de l'ensemble des $\mathcal{D}$-modèles de $\mathrm{IF}(P)$ avec l'ensemble des $\mathcal{D}$-modèles de $\mathrm{FI}(P)$.

Lemme 3.2.16 I *est un* $\mathcal{D}$*-modèle de* $\mathrm{IFF}(P)$ *si et seulement si* I *est un point fixe de* $T_P^{\mathcal{D}}$.

Preuve. $\mathrm{IFF}(P)$ est équivalent à $\mathrm{IF}(P) \cup \mathrm{FI}(P)$. Donc I est un $\mathcal{D}$-modèle de $\mathrm{IFF}(P)$ s'il est à la fois $\mathcal{D}$-modèle de $\mathrm{IF}(P)$ et de $\mathrm{FI}(P)$, i.e. si I est clos par $T_P^{\mathcal{D}}$ et supporté par $T_P^{\mathcal{D}}$, i.e. si $T_P^{\mathcal{D}}(I) = I$, en d'autres termes si I est un point fixe de $T_P^{\mathcal{D}}$.
Le raisonnement inverse montre que si I est un point fixe de $T_P^{\mathcal{D}}$ alors I est un $\mathcal{D}$-modèle de $\mathrm{IFF}(P)$. ■

Corollaire 3.2.17 *Pour toute pré-interprétation* $\mathcal{D}$*:*

1. *tout* $\mathcal{D}$*-modèle de* $\mathrm{IFF}(P)$ *(le complété de* P*) est:*
   (a) *un* $\mathcal{D}$*-modèle de* P*,*
   (b) *un* $\mathcal{D}$*-modèle de* $\mathrm{IF}(P)$*,*
   (c) *un* $\mathcal{D}$*-modèle de* $\mathrm{FI}(P)$*;*
2. *le plus petit* $\mathcal{D}$*-modèle de* $\mathrm{IFF}(P)$ *est le plus petit* $\mathcal{D}$*-modèle de* $\mathrm{IF}(P)$ *(ou de* P*);*
3. *le plus grand* $\mathcal{D}$*-modèle de* $\mathrm{IFF}(P)$ *est le plus grand* $\mathcal{D}$*-modèle de* $\mathrm{FI}(P)$.

Lemme 3.2.18 *Pour tout atome* a*,* $P \models_{\mathcal{D}} \tilde{\exists} a$ *si et seulement si* $\mathrm{IFF}(P) \models_{\mathcal{D}} \tilde{\exists} a$.

Preuve. P et $\mathrm{IFF}(P)$ ont le même plus petit $\mathcal{D}$-modèle. ■

3.2.1 Lien avec la sémantique opérationnelle

Voyons le lien entre la sémantique déclarative selon $\mathcal{D}$ d'un programme P et la sémantique opérationnelle de P.

Définition 3.2.19 Critère de rejet correct/complet pour une pré-interprétation
Un critère de rejet RC *est* correct *pour la pré-interprétation* $\mathcal{D}$ *si, pour tout store* C*:* $C \in \mathrm{RC}$ *implique que* C *est insatisfiable dans* $\mathcal{D}$.
Il est complet *si l'implication inverse est vraie.*

Remarque. Le critère de rejet $\mathrm{RC}_{\mathcal{D}}$ est correct pour $\mathcal{D}$. Il est même complet.
Le critère de rejet $\emptyset$ est correct pour toute pré-interprétation $\mathcal{D}$. ◇

On suppose, jusqu'à la fin de cette section, que le critère de rejet RC est correct pour la pré-interprétation $\mathcal{D}$.

Remarque. Le cas $\mathrm{RC} = \mathrm{RC}_{\mathcal{D}}$ n'est qu'un cas particulier, c'est celui qui correspond aux définitions plus classiques de [54, 65, 47, 55, 39] sur lesquelles nous reviendrons. ◇

Un *$\mathcal{D}$-arbre de preuve* est un arbre de preuve (orienté et étiqueté par la $\mathcal{D}$-base) pour $\Phi(\mathcal{D}, P)$. Un *$\infty\mathcal{D}$-arbre de preuve* est un ∞-arbre de preuve pour $\Phi(\mathcal{D}, P)$. Si A est un $(\infty)\mathcal{D}$-arbre de preuve, on note dom_A son domaine d'arbre et lab_A sa fonction d'étiquetage.

Tout $(\infty)\mathcal{D}$-arbre de preuve utilise, pour "prouver" sa racine, des $\mathcal{D}$-règles de $\Phi(\mathcal{D}, P)$. Ces $\mathcal{D}$-règles sont issues des clauses du programme P.

Nous allons maintenant définir une correspondance entre les $\mathcal{D}$-arbres de preuve et les squelettes réponses. Cette correspondance explique la terminologie *squelette*. Les réponses sont les squelettes des preuves dans le système $\Phi(\mathcal{D}, P)$.

On peut voir un $\mathcal{D}$-arbre de preuve comme une "réponse valuée".

En effet, soit S une réponse, $reno_S$ une fonction de renommage pour S et v une valuation telles que $v_{\mathcal{D}}(const(S, reno_S)) = true$. On note $pt(S, reno_S, v)$ l'arbre orienté sur le domaine d'arbre dom_S, étiqueté par la $\mathcal{D}$-base, défini par :

- $pt(S, reno_S, v)$ et S ne diffèrent que par leur fonction d'étiquetage ;
- pour tout nœud N_S de dom_S : si $reno_S(N_S) = a \leftarrow C \,\square\, a_1 \cdots a_n$ alors $lab_{pt(S,reno_S,v)}(N_S) = v_{\mathcal{D}}(a)$.

Lemme 3.2.20 *Si S est une réponse et $reno_S$ une fonction de renommage pour S alors, pour toute valuation v telle que $v_{\mathcal{D}}(const(S, reno_S)) = true$, $pt(S, reno_S, v)$ est un $\mathcal{D}$-arbre de preuve.*

Preuve. Par induction sur les réponses. ■

Inversement, on peut voir une réponse comme le "squelette d'un $\mathcal{D}$-arbre de preuve".

En effet, soit A un $\mathcal{D}$-arbre de preuve. On note $sk(A)$ l'arbre orienté sur le domaine d'arbre dom_A, étiqueté par des noms de clauses, défini par :

- A et $sk(A)$ ne diffèrent que par leur fonction d'étiquetage ;
- pour tout nœud N_A de dom_A : si u est le nom de la clause d'où est issue la $\mathcal{D}$-règle qui relie $lab_A(N_A)$ aux étiquettes de ses fils alors $lab_{sk(A)}(N_A) = u$.

Lemme 3.2.21 *Si A est un $\mathcal{D}$-arbre de preuve alors $sk(A)$ est une réponse.*
De plus, pour toute fonction de renommage $reno_{sk(A)}$, il existe une valuation v telle que $v_{\mathcal{D}}(const(S, reno_{sk(A)})) = true$ (i.e. $sk(A)$ est une réponse selon $\mathrm{RC}_{\mathcal{D}}$).

Preuve. Par induction sur les arbres de preuves. ■

Exemple 3.2.5 Fibonacci

La figure 3.1 présente un $\mathcal{N}$-arbre de preuve qui correspond à la réponse S_2 de la figure 2.10.
La figure 3.2 présente une $\infty\mathcal{N}$-arbre de preuve qui correspond au squelette S_∞ de la figure 2.11.

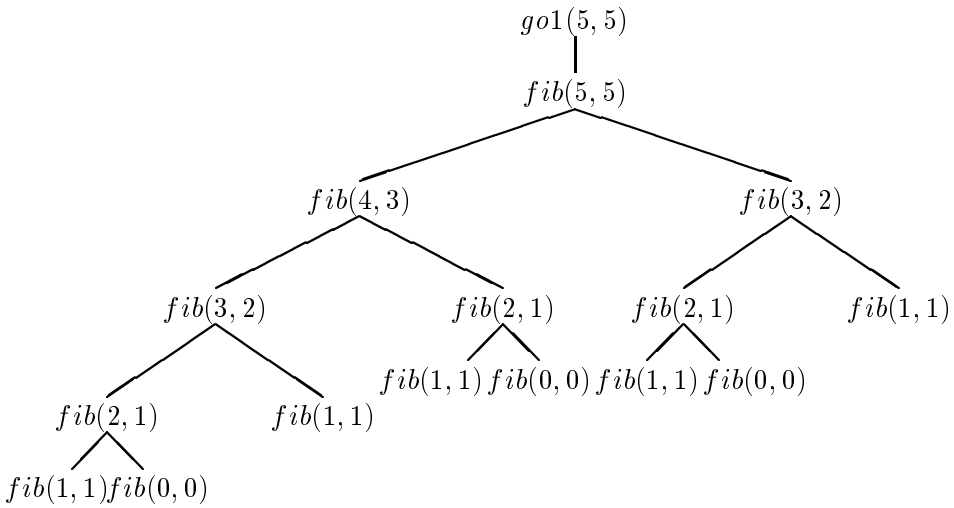


Figure 3.1 : $\mathcal{N}$-arbre de preuve pour $\Phi(\mathcal{N}, \text{FIB})$

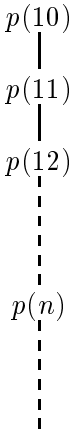


Figure 3.2 : $\infty\mathcal{N}$-arbre de preuve

La correspondance entre les réponses et les $\mathcal{D}$-arbres de preuve permet de montrer le lien entre $pppf(T_P^{\mathcal{D}})$ et $\text{SS}(P)$.

Soit X un ensemble d'atomes contraints. On note $[X]_{\mathcal{D}} = \{v_{\mathcal{D}}(a) \mid \text{il existe } a \leftarrow C \in X, v_{\mathcal{D}}(C) = \mathit{true}\}$.

Lemme 3.2.22 $pppf(T_P^{\mathcal{D}}) = [\text{SS}(P)]_{\mathcal{D}}$.

Preuve.

$\subseteq$ Soit $p(d_1, \ldots, d_n) \in pppf(T_P^{\mathcal{D}})$. Donc $p(d_1, \ldots, d_n)$ est racine d'un $\mathcal{D}$-arbre de preuve A, $sk(A)$ est une réponse et il existe v solution de $\text{AC}(sk(A), p(x_1, \ldots, x_n))$ telle que $v_{\mathcal{D}}(p(x_1, \ldots, x_n)) = p(d_1, \ldots, d_n)$. Donc $p(d_1, \ldots, d_n) \in [\text{SS}(P)]_{\mathcal{D}}$.

$\supseteq$ Soit $p(d_1, \ldots, d_n) \in [\text{SS}(P)]_{\mathcal{D}}$. Donc il existe $p(x_1, \ldots, x_n) \leftarrow C \in \text{SS}(P)$ et il existe v tels que $v_{\mathcal{D}}(C) = \mathit{true}$ et $v_{\mathcal{D}}(p(x_1, \ldots, x_n)) = p(d_1, \ldots, d_n)$. C est un store réponse

pour $\leftarrow p(x_1, \ldots, x_n)$ donc il existe une réponse S pour p et une fonction de renommage $reno_S$ pour S et $\leftarrow p(x_1, \ldots, x_n)$ telles que $\exists_{\{x_1,\ldots,x_n\}} const(S, reno_S) = C$. D'après le lemme 3.2.20, $pt(S, reno_S, v)$ est un $\mathcal{D}$-arbre de preuve, donc l'étiquette de sa racine $p(d_1, \ldots, d_n) \in pppf(T_P^{\mathcal{D}})$.

■

Lemme 3.2.23 *Si C est un store réponse pour le but $\leftarrow a$ alors $P \models_{\mathcal{D}} C \rightarrow a$.*

Preuve. Par induction sur les réponses. ■

Remarque. La réciproque du lemme 3.2.23 est en général fausse (voir l'exemple 3.2.6). ◇

Exemple 3.2.6 Couverture des réponses en CLP($\mathcal{R}$)
Soit le programme P, en CLP($\mathcal{R}$) :

$$p(x) \leftarrow x < 0 \,\square\, \varepsilon$$
$$p(x) \leftarrow x = 0 \,\square\, \varepsilon$$
$$p(x) \leftarrow x > 0 \,\square\, \varepsilon$$

La pré-interprétation $\mathcal{R}$ a pour domaine $\mathbb{R}$ et interprète de façon usuelle les symboles de prédicats de contrainte $<$, $=$ et $>$. Supposons que le langage contienne la contrainte atomique $\top$ interprétée par la valeur de vérité *true*. Le but $\leftarrow p(x)$ admet trois réponses auxquelles correspondent les trois stores réponses : $x < 0$, $x = 0$ et $x > 0$.
$P \models_{\mathcal{R}} \top \rightarrow p(x)$ car, pour toute valuation v, $v_{\mathcal{D}}(p(x)) \in pppf(T_P^{\mathcal{D}})$ (l'ensemble de $\mathcal{D}$-règles est : $\{p(d) \leftarrow \varepsilon \mid d \in \mathbb{R}\}$). Pourtant $\top$ n'est pas une contrainte réponse pour le but $\leftarrow p(x)$.
Cependant : $\models_{\mathcal{R}} \top \rightarrow x < 0 \vee x = 0 \vee x > 0$.

Il existe néanmoins une réciproque plus faible du lemme 3.2.23. Pour énoncer cette réciproque nous introduisons la notion de couverture.

Définition 3.2.24 Couverture dans une pré-interprétation
Soit C un store et $\{C_i\}_{i\in I}$ une famille éventuellement infinie de stores. On note $\models_{\mathcal{D}} C \rightarrow \{C_i\}_{i\in I}$, qui se lit C est couvert *par $\{C_i\}_{i\in I}$ dans $\mathcal{D}$, la relation : pour toute valuation v solution de C il existe $i \in I$ tel que v soit solution de C_i.*
Remarquons que si I est fini alors $\models_{\mathcal{D}} C \rightarrow \{C_i\}_{i\in I}$ se ramène à $\models_{\mathcal{D}} C \rightarrow \bigvee_{i\in I} C_i$.

Lemme 3.2.25 *Pour tout atome a et tout store C, si $P \models_{\mathcal{D}} C \rightarrow a$ alors $\models_{\mathcal{D}} C \rightarrow \mathrm{SS}(P, a)$.*

Preuve. Si $P \models_{\mathcal{D}} C \rightarrow a$ alors pour toute valuation v solution de C, $v_{\mathcal{D}}(a) \in pppf(T_P^{\mathcal{D}})$ donc il existe un $\mathcal{D}$-arbre de preuve A de racine $v_{\mathcal{D}}(a)$, donc $sk(A)$ est une réponse pour $\leftarrow a$ et v est solution de $\mathrm{AC}(S, a)$. ■

Remarque. La réciproque du lemme 3.2.25 est cette fois trivialement vraie en utilisant le lemme 3.2.23. ◇

Remarque. Il n'existe pas toujours de couverture finie si l'ensemble des réponses pour le but est infini, contrairement au lemme 3.3.14 où $\mathcal{D}$ est remplacée par une théorie $\mathcal{T}$. ◇

Exemple 3.2.7 Couverture infinie en CLP($\mathcal{N}$)
Soit le programme P, en CLP($\mathcal{N}$) :

$$p(x) \leftarrow x = 0 \,\square\, \varepsilon$$
$$p(x) \leftarrow x = y + 1 \,\square\, p(y)$$

Le domaine est $\mathbb{N}$ avec l'interprétation habituelle pour les symboles de fonction et les prédicats de contraintes.
Le but $\leftarrow p(x)$ admet l'infinité de contraintes réponses :

$$\begin{array}{ll} C_0 : & x = 0 \\ C_1 : & \exists x_1 (x = x_1 + 1 \wedge x_1 = 0) \\ \cdots & \cdots \\ C_i : & \exists x_1 \cdots \exists x_i (x = x_1 + 1 \wedge \cdots \wedge x_{i-1} = x_i + 1 \wedge x_i = 0) \\ \cdots & \cdots \end{array}$$

Une valuation v est solution de C_i si et seulement si $v(x) = i$.
Il est facile de voir que $P \models_{\mathcal{N}} \top \rightarrow p(x)$, où $\top$ est interprété par *true*, puisque $pppf\,(T_P^{\mathcal{N}}) = \{p(i) \mid i \in \mathbb{N}\}$. Néanmoins, $\top$ n'est pas couverte dans $\mathcal{N}$ par une partie finie de $\{C_i\}_{i \in \mathbb{N}}$: pour toute partie finie il existe une valuation v solution d'aucun store de la partie finie.

Jusqu'ici nous nous sommes intéressés aux $\mathcal{D}$-conséquences positives (c'est-à-dire aux "atomes contraints calculés") de IF(P) et de IFF(P). Intéressons-nous aux $\mathcal{D}$-conséquences négatives (c'est-à-dire aux "atomes couverts calculés"). Nous pouvons déjà remarquer que IF(P) n'a pas de $\mathcal{D}$-conséquences négatives car la $\mathcal{D}$-base est toujours un $\mathcal{D}$-modèle de IF(P). Du point de vue dual FI(P) permet d'étudier les $\mathcal{D}$-conséquences négatives. Remarquons que FI(P) n'a pas de $\mathcal{D}$-conséquences positives : $\emptyset$ est toujours un $\mathcal{D}$-modèle de FI(P).

On utilise la terminologie "$\mathcal{D}$-conséquences négatives" car la contraposée d'un atome couvert fait intervenir une négation. Mais, on préfère les exprimer sous la forme d'atomes couverts afin de rester dans le langage des contraintes (on préfère considérer un ensemble de stores plutôt qu'une conjonction de négation de stores).

On exprime le lien entre FI(P) et la sémantique opérationnelle au sens de ce qui est calculé.

Lemme 3.2.26 *S'il existe un arbre SLD fini pour* $\leftarrow a$ *alors* $\mathrm{FI}(P) \models_{\mathcal{D}} a \rightarrow \mathrm{SS}(P, a)$.

Preuve. Soit $\leftarrow a$ un but pour lequel il existe un arbre SLD fini.
Supposons que non($\mathrm{FI}(P) \models_{\mathcal{D}} a \rightarrow \mathrm{SS}(P, a)$). Donc il existe un $\mathcal{D}$-modèle I de FI(P) et une valuation v tels que $v_I(a) = \mathit{true}$ et, pour tout $C \in \mathrm{SS}(P, a)$, $v_I(C) = \mathit{false}$. Comme $v_{\mathcal{D}}(a) \in I$ alors $v_{\mathcal{D}}(a) \in pgpf\,(T_P^{\mathcal{D}})$ qui est le plus grand $\mathcal{D}$-modèle de FI(P). Donc il existe un $\infty\mathcal{D}$-arbre de preuve enraciné par $v_{\mathcal{D}}(a)$. Il lui correspond, comme dans le lemme 3.2.21, un état complet S tel que pour toute fonction de renommage $reno_S$ pour S, il existe une valuation v', $v'_{\mathcal{D}}(a) = v_{\mathcal{D}}(a)$, v' est solution de toute partie finie de $const(S, reno_S)$. Si l'état complet S est infini alors dans tout arbre SLD il lui correspond une branche infinie (lemme 2.5.12 et contraposée du lemme 2.5.14). D'où la contradiction par hypothèse, il existe un arbre SLD fini pour $\leftarrow a$. Donc S est fini, v est solution de $\mathrm{AC}(S, a)$ et $\mathrm{AC}(S, a) \in \mathrm{SS}(P, a)$. ∎

Notons que la réciproque du lemme 3.2.26 est en général fausse (voir l'exemple 3.2.8).

Remarque. Dans la preuve précédente, nous avons fait correspondre à tout $\infty\mathcal{D}$-arbre de preuve un état complet comme dans le lemme 3.2.21. La correspondance est évidente et ne nécessite pas de preuve. Mais, la correspondance inverse, comme dans le lemme 3.2.20 est plus subtile. En effet, à tout état complet et à toute valuation solution de son ensemble de contraintes associé on peut faire correspondre un $\infty\mathcal{D}$-arbre de preuve parce que la définition fait intervenir une valuation. En revanche, il se peut qu'aucune valuation ne soit solution de l'ensemble de contrainte associé à un état infini. ◇

Corollaire 3.2.27 *S'il existe un arbre SLD fini pour* $\leftarrow a$ *alors* $\mathrm{FI}(P) \models_{\mathcal{D}} a \rightarrow \bigvee_{C \in \mathrm{SS}(P,a)} C$.

Preuve. S'il existe un arbre SLD fini pour $\leftarrow a$ alors $\mathrm{SS}(P, a)$ est fini. ■

Corollaire 3.2.28 *S'il existe un arbre SLD d'échec fini pour* $\leftarrow a$ *alors* $\mathrm{FI}(P) \models_{\mathcal{D}} \neg a$.

Preuve. C'est le cas particulier $\mathrm{SS}(P, a) = \emptyset$. ■

En programmation logique, le corollaire précédent exprime la correction de la négation par l'échec.

Dans le cadre de la programmation logique, le lemme 3.2.26, qui s'exprime facilement, est bien plus intéressant.

La réciproque du lemme 3.2.26 est en général fausse, comme le montre le contre exemple suivant de la réciproque du corollaire 3.2.1 :

Exemple 3.2.8 $\mathcal{D}$-conséquences négatives du complété

Considérons le programme de l'exemple 3.2.7 auquel on a retiré la première clause. Soit P le programme en $\mathrm{CLP}(\mathcal{N})$:

$$p(x) \leftarrow x = y + 1 \,\square\, p(y)$$

Son complété est : $\mathrm{IFF}(P) = p(x) \leftrightarrow \exists y(x = y + 1 \wedge p(y))$. Il est tel que $\mathrm{IFF}(P) \models_{\mathcal{N}} \neg p(x)$.

Preuve. Supposons que $\exists x\, p(x)$ soit une $\mathcal{N}$-conséquence de $\mathrm{IFF}(P)$, donc il existe un $\mathcal{N}$-modèle I de $\mathrm{IFF}(P)$ et une valuation v tels que $v_{\mathcal{N}}(p(x)) \in I$. Or, I est une $\mathcal{N}$-modèle de la clause $p(x) \leftarrow x = y + 1 \,\square\, p(y)$, donc $v_I(\exists y(x = y + 1 \wedge p(y))) = \mathit{true}$; donc $p_{\mathcal{N}}(v_{\mathcal{N}}(y)) \in I$. En itérant le processus, on construit une suite infinie décroissante d'entiers de $\mathbb{N}$, ce qui est bien entendu impossible. Donc, $\neg p(x)$ est une $\mathcal{N}$-conséquence de $\mathrm{IFF}(P)$. ■

Pourtant le but $\leftarrow p(x)$ n'admet pas d'arbre SLD fini. L'arbre SLD pour ce but est indépendant de la règle de calcul, puisque l'unique clause de P est d'arité 1, et cet arbre a une seule branche qui est infinie.

On exprime enfin le corollaire qui établit le lien entre la sémantique déclarative du programme et sa sémantique opérationnelle pour la partie positive et la partie négative dans un même énoncé.

Corollaire 3.2.29 *S'il existe un arbre SLD fini pour le but* $\leftarrow a$ *alors* $\mathrm{IFF}(P) \models_{\mathcal{D}} a \leftrightarrow \mathrm{SS}(P, a)$.

Preuve. Soit $\leftarrow a$ un but pour lequel il existe un arbre SLD fini.
Le lemme 3.2.26 assure que $\mathrm{FI}(P) \models_{\mathcal{D}} a \rightarrow \mathrm{SS}(P, a)$. Le lemme 3.2.23 assure que $\mathrm{IF}(P) \models_{\mathcal{D}} a \leftarrow C$, pour tout $C \in \mathrm{SS}(P, a)$. $\mathrm{SS}(P, a)$ est fini donc $\mathrm{IF}(P) \models_{\mathcal{D}} a \leftarrow \mathrm{SS}(P, a)$. Du corollaire 3.2.17 on déduit $\mathrm{IFF}(P) \models_{\mathcal{D}} a \leftrightarrow \mathrm{SS}(P, a)$. ■

Nous donnons maintenant une définition complémentaire en relation avec la sémantique opérationnelle. C'est la définition de l'ensemble d'échec fini. Cette définition n'a pas été placée dans le chapitre 2 car elle n'est utile que pour le lemme 3.2.33.

Comme on ne s'intéresse qu'aux buts atomiques, on ne peut définir l'ensemble d'échecs finis comme l'ensemble des atomes pour lesquels il existe un arbre SLD d'échec fini. Cette notion est bien trop pauvre. Nous le définissons comme un ensemble d'atomes contraints.

Définition 3.2.30 Ensemble d'échec fini
$\mathrm{FF}(P) = \{a \leftarrow C \mid$ *il existe* $n \in \mathbb{N}$, *tout état* S *pour* $\leftarrow a$ *ayant ses feuilles indéfinies à la profondeur* n *est tel que* $C \wedge \mathrm{AC}(S, a) \in \mathrm{RC}\}$

Comme nous l'avons déjà précisé, l'ensemble d'échec fini ne joue pas un rôle très important. Il n'est utile que pour l'énoncé du lemme 3.2.33.

Remarque. On peut remarquer que si $a \leftarrow C \in \mathrm{FF}(P)$ alors, si on ajoute la clause $p \leftarrow C \,\square\, a$ au programme P (où p est un prédicat de programme qui n'apparaît pas dans l'ensemble Π_p d'origine), il existe un arbre SLD d'échec fini pour le but $\leftarrow p$.
On remarque aussi que pour tout atome a et tout store C il existe un arbre SLD d'échec fini pour le but $\leftarrow p$ dans le programme P augmenté de la clause $p \leftarrow C \,\square\, a$ si et seulement si $a \leftarrow C \in \mathrm{FF}(P)$.
Enfin, nous aurions pu noter un élément $a \leftarrow C$ de $\mathrm{FF}(P)$ sous la forme $\neg(a \,\square\, C)$, mais nous tenons à conserver un ensemble de même nature que ceux manipulés par $T_{\mathcal{D}}^P$. $\diamond$

Exemple 3.2.9 Fibonacci
L'ensemble FF(FIB) contient par exemple $fib(x, y) \leftarrow x = 1 + 1 + 1 \wedge y = 1 + 1 + 1$. Mais il ne contient pas $go1(x, y) \leftarrow x > 1 + 1 + 1 + 1 + 1 + 1$.

Intéressons nous maintenant au lien entre le plus grand point fixe de l'opérateur $T_P^{\mathcal{D}}$ et l'ensemble des échecs finis.

Soit C un store et $\{C_i\}_{i \in I}$ une famille de stores. On écrit $\models_{\mathcal{D}} C \leftrightarrow \{C_i\}_{i \in I}$ pour exprimer que :

- pour toute valuation v solution de C, il existe $i \in I$ tel que v soit solution de C_i ;
- pour tout $i \in I$, pour toute valuation v solution de C_i, v est solution de C.

C'est-à-dire :

- $\models_{\mathcal{D}} C \rightarrow \{C_i\}_{i \in I}$;
- pour tout $i \in I$, $\models_{\mathcal{D}} C_i \rightarrow \{C\}$.

Remarque. Si I est fini alors $\models_{\mathcal{D}} C \leftrightarrow \{C_i\}_{i \in I}$ est équivalent à $\models_{\mathcal{D}} C \leftrightarrow \bigvee_{i \in I} C_i$. $\diamond$

Définition 3.2.31 Langage des contraintes compact pour les solutions
Le langage des contraintes CONST *est* compact pour les solutions *selon la pré-interprétation* $\mathcal{D}$ *si pour tout store* C *il existe un ensemble de stores* $\{C_i\}_{i \in I}$ *tel que* $\models_{\mathcal{D}} \neg C \leftrightarrow \{C_i\}_{i \in I}$.

La notion de compacité des solutions a été définie dans [54]. Conformément à [64, 55] nous n'utilisons que la partie SC_2.

Nous avons traduit le terme anglais "solution compact" par compacité des solutions. Cependant, la notion de compacité intervenant dans cette définition n'est pas très claire.

On peut remarquer qu'il n'y a pas de langage de contraintes raisonnable et utilisé qui ne soit pas compact pour les solutions dans la pré-interprétation sous-jacente.

Définition 3.2.32 $\mathcal{D}$-arbre de preuve partiel à la profondeur n
Un $\mathcal{D}$-arbre de preuve partiel à la profondeur $n \in \mathbb{N}$ *est un arbre de profondeur* n *défini comme un* $\mathcal{D}$*-arbre de preuve excepté que ses feuilles de profondeur* n *ne sont pas nécessairement étiquetées par des conclusions de règles de* $\Phi(\mathcal{D}, P)$ *dont l'ensemble des prémisses est* $\emptyset$*.*

Comme pour le lemme 3.2.20 et le lemme 3.2.21 il existe une correspondance évidente entre les $\mathcal{D}$-arbres de preuve partiels à la profondeur n et les états de profondeur n qui ont toutes leurs feuilles indéfinies à la profondeur n.

Lemme 3.2.33 *Soient* $\mathcal{D}$ *une pré-interprétation.*
Si le langage des contraintes est compact pour les solutions selon la pré-interprétation $\mathcal{D}$ *alors*

$$[\mathrm{FF}(P)]_{\mathcal{D}} = \overline{T_{\mathcal{D}}^P \downarrow \omega}$$

Preuve. Seule la partie $\supseteq$ utilise la compacité des solutions.

$\subseteq$ Soit $a \leftarrow C \in \mathrm{FF}(P)$. Donc pour store réponse C' au but $\leftarrow a$, $C \wedge C' \in \mathrm{RC}$. Soit v une solution de C. Supposons que $v_{\mathcal{D}}(a) \in T_{\mathcal{D}}^P \downarrow \omega$. Donc pour tout $n \in \mathbb{N}$, il existe un $\mathcal{D}$-arbre de preuve A_n partiel à la profondeur n enraciné par $v_{\mathcal{D}}(a)$. Au $\mathcal{D}$-arbre de preuve partiel A_n, on peut faire correspondre (en adaptant la définition qui précède le lemme 3.2.21) un état S_n dont les feuilles indéfinies sont à la profondeur n. De plus, il existe une solution v' de $\mathrm{AC}(S_n, a)$ telle que $v'_{\mathcal{D}}(a) = v_{\mathcal{D}}(a)$. Comme v' ne concerne que les variables de a, on peut prendre v. Comme $a \leftarrow C \in \mathrm{FF}(P)$ alors il existe $n_0 \in \mathbb{N}$ tel que tout état S' qui a ses feuilles indéfinies à une profondeur n_0 est soit une réponse, soit tel que $C \wedge \mathrm{AC}(S', a) \in \mathrm{RC}$. Comme RC est correct pour $\mathcal{D}$, cela contredit l'hypothèse.

$\supseteq$ Soit $p(d_1, \ldots, d_n) \in \overline{T_{\mathcal{D}}^P \downarrow \omega}$. Il existe $n_0 \in \mathbb{N}$ tel que $p(d_1, \ldots, d_n) \notin T_{\mathcal{D}}^P \downarrow n_0$. Donc $p(d_1, \ldots, d_n)$ n'enracine aucun $\mathcal{D}$-arbre de preuve partiel à la profondeur n_0. Soit $\{S_i\}_{i \in \{1,\ldots,m\}}$ l'ensemble des états pour p ayant leurs feuilles indéfinies à la profondeur n_0. Pour tout $i = 1, \ldots, m$, si v est solution de $\mathrm{AC}(S_i, p(x_1, \ldots, x_n))$ alors $v_{\mathcal{D}}(p(x_1, \ldots, x_n)) \neq p(d_1, \ldots, d_n)$, sinon on déduirait de S_i (en adaptant la définition qui précède le lemme 3.2.20) un $\mathcal{D}$-arbre de preuve partiel à la profondeur n_0 enraciné par $p(d_1, \ldots, d_n)$.

- Si $m = 0$ alors, d'après les définitions, $p(d_1, \ldots, d_n) \in [\mathrm{FF}(P)]_{\mathcal{D}}$.
- Sinon, pour tout $i = 1, \ldots, m$, il existe une famille de stores $\{C_{i,j}\}_{j \in J_i}$ telle que $\models_{\mathcal{D}} \neg\mathrm{AC}(S_i, p(x_1, \ldots, x_n)) \leftrightarrow \{C_{i,j}\}_{j \in J_i}$. Soit v une valuation telle que $v_{\mathcal{D}}(p(x_1, \ldots, x_n)) = p(d_1, \ldots, d_n)$. Donc il existe $j \in J_i$ tel que $v_{\mathcal{D}}(C_{i,j}) = \mathit{true}$. Pour chaque $i = 1, \ldots, m$, on choisit un tel store $C_{i,j}$, notée C'_i. $\mathrm{AC}(S_i, p(x_1, \ldots, x_n)) \wedge C'_i$ n'a pas de solution puisque $\models_{\mathcal{D}} C'_i \rightarrow \neg\mathrm{AC}(S_i, p(x_1, \ldots, x_n))$. Soit $\{\theta_i\}_{i \in \{1,\ldots,m\}}$ une famille de renommage tel que pour tout $j \neq j'$, $var(C'_j\theta_j) \cap var(C'_{j'}\theta_{j'}) \subseteq \{x_1, \ldots, x_n\}$, et pour tout $k = 1, \ldots, n$, $x_k\theta_j = x_k\theta_{j'} = x_k$. Donc $C = \bigwedge_{i=1,\ldots,m} C'_i\theta_i$ a une solution v' telle que $v'_{\mathcal{D}}(p(x_1, \ldots, x_n)) = p(d_1, \ldots, d_n)$. Donc, on constate que $p(x_1, \ldots, x_n) \leftarrow C \in \mathrm{FF}(P)$.

Alors $p(d_1, \ldots, d_n) \in [\mathrm{FF}(P)]_{\mathcal{D}}$.

■

Exemple 3.2.10 Langage non compact pour les solutions
Considérons un langage de contraintes qui contient uniquement les contraintes basiques de l'ensemble : $\{x = a \mid x \in V\} \cup \{x = b \mid x \in V\}$.
Soit $\mathcal{D}$ une pré-interprétation telle que

- $\mathbb{D} = \{\alpha, \beta, \gamma\}$;
- $a_{\mathcal{D}} = \alpha$ et $b_{\mathcal{D}} = \beta$
- $=_{\mathcal{D}}$ est la relation binaire sur $\mathbb{D}$: $\{(\alpha,\alpha),(\beta,\beta),(\gamma,\gamma)\}$.

Le langage des contraintes n'est pas compact pour les solutions selon $\mathcal{D}$: il n'existe pas de famille $\{C_i\}_{i\in I}$ telle que $\models_{\mathcal{D}} \neg(x = a) \leftrightarrow \{C_i\}_{i\in I}$.
Soit P le programme :

$$p(x) \leftarrow x = a$$

Le critère de rejet est $\mathrm{RC}_{\mathcal{D}}$.
D'une part, on a $\mathrm{FF}(P) = \{p(x) \leftarrow x = b \mid x \in V\}$, c'est-à-dire $[\mathrm{FF}(P)]_{\mathcal{D}} = \{p(\beta)\}$.
D'autre part, on a $T^P_{\mathcal{D}} \downarrow 1 = T^P_{\mathcal{D}} \downarrow \omega = \{p(\alpha)\}$, c'est-à-dire $\overline{T^P_{\mathcal{D}} \downarrow \omega} = \{p(\beta), p(\gamma)\}$.
On constate alors que $[\mathrm{FF}(P)]_{\mathcal{D}} \neq \overline{T^P_{\mathcal{D}} \downarrow \omega}$.
On montre ici, sur un exemple très simple (le programme est simple, l'interprétation des contraintes est simple), la raison pour laquelle le lemme 3.2.33 est faux si le langage des contraintes n'est pas compact pour les solutions dans $\mathcal{D}$. En fait, il ne s'agit que d'un problème relatif à l'expressivité du langage des contraintes (si on ajoute les contraintes basiques $x = c$, pour tout $x \in V$, au langage alors il devient compact pour les solutions). Dans la réalité, on peut remarquer que tous les langages semblent compacts pour les solutions dans leur domaine sous-jacent.

Les résultats qui précèdent ont été montré sans supposer que le critère de rejet soit complet pour $\mathcal{D}$. Seule sa correction était utile.

On suppose maintenant que le critère de rejet RC est *correct et complet* pour $\mathcal{D}$, i.e. $\mathrm{RC} = \mathrm{RC}_{\mathcal{D}}$. Cela nous permet de comparer notre travail avec les travaux classiques de la littérature.

Corollaire 3.2.34 *Les propositions suivantes sont équivalentes :*

1. *il existe une réponse pour le but* $\leftarrow a$ *(selon* $\mathrm{RC}_{\mathcal{D}}$*) ;*
2. $P \models_{\mathcal{D}} \tilde{\exists} a$ *;*
3. *il existe une valuation* v *telle que* $v_{\mathcal{D}}(a) \in pppf(T^{\mathcal{D}}_P)$.

On retrouve donc les ensembles succès de la littérature.
À titre de comparaison, on remarque que si le critère de rejet est $\mathrm{RC}_{\mathcal{D}}$ (i.e. le critère de rejet correct et complet pour $\mathcal{D}$) alors :

- $\mathrm{SS}(P) = pppf(S^{\mathcal{D}}_P)$, où $S^{\mathcal{D}}_P$ est l'opérateur défini sur les ensembles d'atomes contraints dans [65], repris dans [55].
- $\mathrm{SS}(P) = SS_3(P, \mathcal{D})$, où $SS_3(P, \mathcal{D})$ est l'ensemble succès défini dans [47].
- $[\mathrm{SS}(P)]_{\mathcal{D}} = pppf(T^{\mathcal{D}}_P)$, où $T^{\mathcal{D}}_P$ est l'opérateur sur les ensembles de $\mathcal{D}$-atomes de [65, 55].
- $[\mathrm{SS}(P)]_{\mathcal{D}} = SS_1(P, \mathcal{D})$, où $SS_1(P, \mathcal{D})$ est l'ensemble de $\mathcal{D}$-atomes succès de [47].
- $[\mathrm{SS}(P)]_{\mathcal{D}} = [SS_2(P, \mathcal{D})]_{\mathcal{D}}$, où $SS_2(P, \mathcal{D})$ est l'ensemble succès défini dans [47].

On peut remarquer que dans [47] le théorème 5.9 annonce à tort que $P \models_{\mathcal{D}} C \rightarrow p(x_1, \ldots, x_n)$ si et seulement si $p(x_1, \ldots, x_n) \leftarrow C \in SS_2(P, \mathcal{D})$ alors que le programme P de l'exemple 3.2.7 est tel que $P \models_{\mathcal{N}} true \rightarrow p(x)$ et $p(x) \leftarrow true \notin SS_2(P, \mathcal{N})$.

3.3 Selon une théorie $\mathcal{T}$

Soit $\mathcal{L}$ un langage du premier ordre. Une formule F est *conséquence logique* d'un sous-ensemble Γ de formules closes (i.e. sans variables libres) du langage $\mathcal{L}$, noté $\Gamma \models F$, si F est vraie dans toute interprétation de $\mathcal{L}$ qui est modèle de Γ. Une *théorie* sur le langage du premier ordre $\mathcal{L}$ est un sous-ensemble des formules closes du langage $\mathcal{L}$ fermé par conséquence logique. La *théorie engendrée* par l'ensemble de formule closes Γ est $\{F \mid \Gamma \models F, F \text{ formule close}\}$. Une théorie $\mathcal{T}$ est *complète* si pour toute formule close F, soit $F \in \mathcal{T}$, soit $\neg F \in \mathcal{T}$. Un ensemble de formules Γ est *satisfiable* dans la théorie $\mathcal{T}$ s'il existe un modèle $\mathcal{D}$ de $\mathcal{T}$ et une valuation v dans $\mathcal{D}$ tel que, pour toute formule $F \in \Gamma$, $v_{\mathcal{D}}(F) = \mathit{true}$.

Le langage du premier ordre auquel on s'intéresse est celui construit sur (V, Σ, Π_c). Dans notre cadre, on s'intéresse à une notion plus faible de théorie complète : une théorie $\mathcal{T}$ est *complète pour la satisfaction* des stores, si pour tout store C, soit $\mathcal{T} \models \tilde{\exists} C$, soit $\mathcal{T} \models \neg C$. Ce type de théorie joue un rôle important dans la mesure où la satisfiabilité d'un store dans la théorie et sa satisfiabilité dans un modèle de la théorie (une pré-interprétation) sont deux notions équivalentes.

Lemme 3.3.1 *Si $\mathcal{T}$ est une théorie complète pour la satisfaction des stores et $\mathcal{D}$ est un modèle de $\mathcal{T}$ alors* $\mathrm{RC}_{\mathcal{T}} = \mathrm{RC}_{\mathcal{D}}$.

Preuve. Évidente. ∎

Lemme 3.3.2 *$P, \mathcal{T} \models \tilde{\exists} a$ si et seulement si* $\mathrm{IFF}(P), \mathcal{T} \models \tilde{\exists} a$.
$P, \mathcal{T} \models C \to a$ si et seulement si $\mathrm{IFF}(P), \mathcal{T} \models C \to a$.

Preuve.

1. $P, \mathcal{T} \models \tilde{\exists} a$ si et seulement si $\mathrm{IFF}(P), \mathcal{T} \models \tilde{\exists} a$.
   - $P, \mathcal{T} \models \tilde{\exists} a \Rightarrow \mathrm{IFF}(P), \mathcal{T} \models \tilde{\exists} a$: car tout modèle de $\mathrm{IFF}(P)$ et $\mathcal{T}$ est un modèle de P et $\mathcal{T}$.
   - $P, \mathcal{T} \models \tilde{\exists} a \Leftarrow \mathrm{IFF}(P), \mathcal{T} \models \tilde{\exists} a$: si $\mathrm{IFF}(P), \mathcal{T} \models \tilde{\exists} a$ alors $\mathrm{IFF}(P) \models_{\mathcal{D}} \tilde{\exists} a$, donc $P \models_{\mathcal{D}} \tilde{\exists} a$, pour tout modèle $\mathcal{D}$ de $\mathcal{T}$. Donc $P, \mathcal{T} \models \tilde{\exists} a$.
2. $P, \mathcal{T} \models C \to a$ si et seulement si $\mathrm{IFF}(P), \mathcal{T} \models C \to a$.
   - $P, \mathcal{T} \models C \to a \Rightarrow \mathrm{IFF}(P), \mathcal{T} \models C \to a$: car tout modèle de $\mathrm{IFF}(P)$ est un modèle de P.
   - $P, \mathcal{T} \models C \to a \Leftarrow \mathrm{IFF}(P), \mathcal{T} \models C \to a$: si $\mathrm{IFF}(P), \mathcal{T} \models C \to a$ alors, pour tout modèle $\mathcal{D}$ de $\mathcal{T}$, pour toute valuation v dans $\mathcal{D}$, pour tout $\mathcal{D}$-modèle I de $\mathrm{IFF}(P)$, $v_{\mathcal{D}}(a) \in I$. Donc $v_{\mathcal{D}}(a)$ est dans le plus petit $\mathcal{D}$ modèle de $\mathrm{IFF}(P)$ qui est aussi le plus petit $\mathcal{D}$-modèle de P. Donc $P, \mathcal{T} \models C \to a$.

∎

On peut envisager différentes théories, définies à partir

- d'une pré-interprétation $\mathcal{D}$;
- d'un critère de rejet RC ;
- d'un solveur de contraintes $\mathcal{A}$.

Soit d'abord $\mathcal{D}$ une pré-interprétation. La théorie associée à $\mathcal{D}$ est $th(\mathcal{D})$ engendrée par $\{\tilde{\exists} c \mid c \in \mathrm{CONST}, \models_{\mathcal{D}} \tilde{\exists} c\} \cup \{\tilde{\forall} \neg c \mid c \in \mathrm{CONST}, \models_{\mathcal{D}} \neg c\}$.

Lemme 3.3.3 *Soit $\mathcal{D}$ une pré-interprétation.*
$th(\mathcal{D})$ est complète pour la satisfaction des contraintes basiques.

Preuve. D'après la définition de $th(\mathcal{D})$. ■

Remarque. On constate que $\models_{\mathcal{D}}$ et $th(\mathcal{D})$ se comportent de la même manière sur les stores, mais leur effet peut être différent sur les autres formules construite sur (V, Σ, Π_c) ainsi que sur les formules du langage du programme. ◇

Remarque. (**importante**)
De plus, si l'on considère la théorie $\mathcal{T}$ constituée de l'ensemble des formules vraies dans $\mathcal{D}$ alors $\mathcal{T}$ est une théorie complète. Ses modèles sont élémentairement équivalents.
Cependant, les propriétés de la sémantique selon que l'on se place du point de vue de $\mathcal{D}$ ou de $\mathcal{T}$ sont différentes. Les conséquences logiques de $\mathcal{T}$ et les $\mathcal{D}$-conséquences sont les mêmes tant que l'on parle de formules. Mais quand on considère les pseudo-formules infinies (du type $C \rightarrow \{C_i\}_{i \in I}$) alors les conséquences de $\mathcal{T}$ et les $\mathcal{D}$-conséquences ne sont plus les mêmes.
C'est parce que le théorème de compacité de la logique du premier ordre (voir corollaire 3.3.13) n'a pas de pendant pour une interprétation fixée. ◇

Soit maintenant RC un critère de rejet. La théorie associée à RC est $th(\mathrm{RC})$ engendrée par $\{\tilde{\exists} C \mid C \notin \mathrm{RC}\} \cup \{\tilde{\forall} \neg C \mid C \in \mathrm{RC}\}$.

Lemme 3.3.4 *Soit $\mathcal{D}$ une pré-interprétation.*
Si RC *est correct et complet pour $\mathcal{D}$ alors $th(\mathrm{RC})$ est complète pour la satisfaction des stores.*

Preuve. Si RC est correct et complet pour $\mathcal{D}$ ($\mathrm{RC} = \mathrm{RC}_{\mathcal{D}}$) alors pour tout store C :

- si $\models_{\mathcal{D}} \tilde{\exists} C$ alors $C \notin \mathrm{RC}$ donc $\tilde{\exists} C \in th(\mathrm{RC})$ donc $th(\mathrm{RC}) \models \tilde{\exists} C$;
- si $\models_{\mathcal{D}} \neg \tilde{\exists} C$ alors $C \in \mathrm{RC}$ donc $\tilde{\forall} \neg C \in th(\mathrm{RC})$ donc $th(\mathrm{RC}) \models \neg C$.

■

Remarque. $th(\mathrm{RC}_{\mathcal{D}})$ n'est pas obligatoirement complète ni même consistante (à cause des formules qui ne sont pas des stores). ◇

Soit enfin $\mathcal{A}$ un solveur de contraintes. Si le solveur de contraintes $\mathcal{A}$ ne répond que `oui` ou `non` alors cela revient à considérer la théorie associée à un critère de rejet.

En revanche, si $\mathcal{A}$ a trois types de réponse (par exemple CLP($\mathcal{R}$)) qui sont `yes`, `no` et `maybe` alors la théorie associée à $\mathcal{A}$ est $th(\mathcal{A})$ engendrée par : $\{\tilde{\exists} C \mid \mathcal{A}$ répond `yes` pour $C\} \cup \{\tilde{\forall} \neg C \mid \mathcal{A}$ répond `no` pour $C\}$.

Si $\mathcal{A}$ répond `yes` pour C seulement si C admet une solution dans la pré-interprétation $\mathcal{D}$ et `no` seulement s'il n'en admet pas, alors il existe un modèle $\mathcal{D}'$ de $th(\mathcal{A})$, i.e $th(\mathcal{A})$ est consistante.

Nous ne donnons pas de système de règles associé à un programme P et une théorie $\mathcal{T}$. On pourrait le définir comme un système de règles sur l'ensemble des atomes contraints comme dans la section 3.4. Alors, il suffirait de suivre tout le travail de la section 3.2 pour retrouver de nombreux résultats exprimant les liens entre les ensembles clos, supportés, points fixes de ce système et les conséquences logiques de $\mathrm{IF}(P)$, $\mathrm{FI}(P)$, $\mathrm{IFF}(P)$. Nous nous intéressons tout de suite aux liens entre la sémantique déclarative selon une théorie et la sémantique opérationnelle (ce qui est calculé).

3.3.1 Lien avec la sémantique opérationnelle

Nombre des résultats de la section 3.2 s'adaptent en remplaçant $\mathcal{D}$ par $\mathcal{T}$, il suffit alors de vérifier que, selon le cas, le résultat est vrai pour tout modèle $\mathcal{D}$ de $\mathcal{T}$, ou il existe un modèle $\mathcal{D}$ de $\mathcal{T}$ pour lequel le résultat est vrai.

Définition 3.3.5 Critère de rejet correct pour une théorie
Un critère de rejet RC *est* correct *pour la théorie* $\mathcal{T}$ *si, pour tout store* C *:* $C \in$ RC *implique* $\mathcal{T} \models \neg\mathcal{C}$.
Il est complet *si l'implication inverse est vraie.*

Remarque. Le critère de rejet $\mathrm{RC}_{\mathcal{T}}$ est correct et complet pour $\mathcal{T}$.
Le critère de rejet $\emptyset$ est correct pour toute théorie $\mathcal{T}$. ◇

Lemme 3.3.6 *Si* RC *est correct pour* $\mathcal{T}$ *alors* RC *est correct pour tout modèle* $\mathcal{D}$ *de* $\mathcal{T}$.

Preuve. Évidente. ■

On suppose, jusqu'à la fin de cette section, qu'un critère de rejet RC correct pour la théorie $\mathcal{T}$ est fixé.

Remarque. Le cas $\mathrm{RC} = \mathrm{RC}_{\mathcal{T}}$ n'est qu'un cas particulier qui correspond, si de plus $\mathcal{T}$ est complète pour la satisfaction des stores, aux définitions de [54]. ◇

On commence par quelques résultats d'existence d'une réponse et de correction des réponses.

Lemme 3.3.7 *Si* $\mathrm{RC} = \mathrm{RC}_{\mathcal{T}}$ *et s'il existe une réponse au but* $\leftarrow a$ *alors non*$(P, \mathcal{T} \models \neg a)$.

Preuve. S'il existe une réponse S au but $\leftarrow a$ alors $\mathrm{non}(\mathcal{T} \models \neg\mathrm{AC}(S, a))$. Donc il existe un modèle $\mathcal{D}$ de $\mathcal{T}$ et une valuation v tels que $v_{\mathcal{D}}(\mathrm{AC}(S, a)) = \mathit{true}$. Donc $P \models_{\mathcal{D}} \tilde{\exists} a$, donc $\mathrm{non}(P, \mathcal{T} \models \neg a)$. ■

Lemme 3.3.8 *Si* $P, \mathcal{T} \models \tilde{\exists} a$ *alors il existe une réponse au but* $\leftarrow a$.

Preuve. Si $P, \mathcal{T} \models \tilde{\exists} a$ alors $P \models_{\mathcal{D}} \tilde{\exists} a$ pour tout modèle $\mathcal{D}$ de $\mathcal{T}$. Soit $\mathcal{D}$ un modèle de $\mathcal{T}$. Puisque $P \models_{\mathcal{D}} \tilde{\exists} a$ alors il existe un squelette fini complet S pour $\leftarrow a$ tel que $\mathrm{AC}(S, a)$ a une solution dans $\mathcal{D}$. Donc $\mathrm{non}(\mathcal{T} \models \neg\mathrm{AC}(S, a))$. Comme RC est correct pour $\mathcal{T}$ alors S est une réponse pour le but $\leftarrow a$. ■

Lemme 3.3.9 *Supposons que* $\mathcal{T}$ *soit complète pour la satisfaction. Il existe une réponse* S *au but* $\leftarrow a$ *si et seulement si* $P, \mathcal{T} \models \tilde{\exists} a$.

Preuve. Triviale. ■

Lemme 3.3.10 *Si* C *est un store réponse pour le but* $\leftarrow a$ *alors* $P, \mathcal{T} \models C \rightarrow a$.

Preuve. Cas particulier du lemme 3.4.9. ■

Intéressons nous maintenant au problème de la complétude des réponses calculées.

Définition 3.3.11 Couverture dans une théorie
Soit $\{C_i\}_{i\in I}$ une famille de stores. On note $\mathcal{T} \models C \rightarrow \{C_i\}_{i\in I}$ qui se lit C est couverte par $\{C_i\}_{i\in I}$ dans $\mathcal{T}$ et qui signifie : $\models_{\mathcal{D}} C \rightarrow \{C_i\}_{i\in I}$, pour tout modèle $\mathcal{D}$ de $\mathcal{T}$.

Lemme 3.3.12 *Si $P, \mathcal{T} \models C \rightarrow a$ alors $\mathcal{T} \models C \rightarrow \mathrm{SS}(P, a)$.*

Preuve. Si $P, \mathcal{T} \models C \rightarrow a$ alors pour tout modèle $\mathcal{D}$ de $\mathcal{T}$, $P \models_{\mathcal{D}} C \rightarrow a$. Donc $\models_{\mathcal{D}} C \rightarrow \{\mathrm{AC}(S, a)\}_{S\in success(a)}$ pour tout modèle $\mathcal{D}$ de $\mathcal{T}$. Donc $\mathcal{T} \models C \rightarrow \mathrm{SS}(P, a)$. ■

Corollaire 3.3.13 *du théorème de compacité de la logique du premier ordre.*
$\mathcal{T} \models C \rightarrow \{C_i\}_{i\in I}$ si et seulement si il existe une partie finie $I_0 \subseteq I$ telle que $\mathcal{T} \models C \rightarrow \bigvee_{i_0\in I_0} C_{i_0}$.

Preuve.

⇒ Pour tout modèle $\mathcal{D}$ de $\mathcal{T}$, l'ensemble de formules $\mathcal{T} \cup \{C\} \cup \{\neg C_i\}_{i\in I}$ n'a pas de solution dans $\mathcal{D}$. D'après le théorème de compacité de la logique, il existe une partie finie de $\mathcal{T} \cup \{C\} \cup \{\neg C_i\}_{i\in I}$ insatisfiable. L'intersection de cette partie finie avec $\{C_i\}_{i\in I}$ est une partie finie de $\{C_i\}_{i\in I}$. Donc il existe une partie finie $I_0 \subseteq I$ telle que $\mathcal{T} \models C \rightarrow \bigvee_{i_0\in I_0} C_{i_0}$.

⇐ Évident.

■

Lemme 3.3.14 *Si $P, \mathcal{T} \models C \rightarrow a$ alors $T \models C \rightarrow \{\mathrm{AC}(S, a)\}_{S\in success_f(a)}$, où $success_f(a)$ est une partie finie de $success(a)$.*

Preuve. Conséquence du lemme 3.3.12 et du corollaire 3.3.13. ■

C'est principalement pour ce lemme et le suivant qu'on note une différence avec les résultats de la section 3.2.1.

Définition 3.3.15 Indépendance des contraintes négatives
*Le langage des contraintes a la propriété d'*indépendance des contraintes négatives *(INC) si $\mathcal{T} \models C \rightarrow \bigvee_{i\in\{1,\ldots,n\}} C_i$ implique qu'il existe $j \in \{1, \ldots, n\}$ tel que $\mathcal{T} \models C \rightarrow C_j$.*

La propriété INC a été énoncé par Maher [65] de manière équivalente par $\mathcal{T} \models \neg\tilde{\exists}(C \wedge \bigwedge_{i\in\{1,\ldots,n\}} \neg C_i)$ implique qu'il existe $j \in \{1, \ldots, n\}$ tel que $\mathcal{T} \models \neg\tilde{\exists}(C \wedge \neg C_j)$.

Cette propriété est vérifiée dans le cas de la programmation logique, où les substitutions sont vues comme des contraintes. C'est la raison pour laquelle le résultat suivant est bien connu pour les programmes logiques.

Lemme 3.3.16 *Supposons que le langage des contraintes vérifie la propriété d'indépendance des contraintes négatives.*
Si $P, \mathcal{T} \models C \rightarrow a$ alors il existe une réponse S telle que $\mathcal{T} \models C \rightarrow \mathrm{AC}(S, a)$.

Preuve. Conséquence immédiate du lemme 3.3.14 et de la propriété d'indépendance des contraintes négatives. ■

Lemme 3.3.17 *Si $\leftarrow a$ a un arbre SLD d'échec fini alors $\mathrm{IFF}(P), \mathcal{T} \models \neg a$.*

Preuve. Similaire à la preuve du lemme 3.2.26. ■

3.4 Selon un critère de rejet RC

Cette section ne définit pas une sémantique déclarative pour les programmes. Elle aurait sa place à la fin du chapitre 2.

Cependant, nous l'avons placée ici comme une introduction à la section 3.5 qui généralise les sections 3.2 et 3.3 par l'intermédiaire d'une relation abstraite de couverture.

Définition 3.4.1 RC-clause
Une RC*-clause est une paire $b \leftarrow B$, où b est un atome contraint et B est une suite finie d'atomes contraints.*

Définition 3.4.2 Système de RC-règles associé à une clause
Le système de RC*-règles associé à la clause $a \leftarrow C \,\square\, a_1 \cdots a_n$, noté $\Phi(\mathrm{RC}, a \leftarrow C \,\square\, a_1 \cdots a_n)$, est l'ensemble des $(a\theta \leftarrow C_0) \leftarrow (a_1\theta \leftarrow C_1) \cdots (a_n\theta \leftarrow C_n)$ où θ est un renommage, $C_1, \ldots, C_n$ sont n stores, $C_0 = \exists_{-a\theta}(C\theta \wedge C_1 \wedge \cdots \wedge C_n)$ et $C_0 \notin \mathrm{RC}$.*

Définition 3.4.3 Système de RC-règles associé au programme
Le système de RC*-règle associé à P est*

$$\Phi(\mathrm{RC}, P) = \bigcup_{R \in P} \Phi(\mathrm{RC}, R)$$

Son opérateur associé $T_{\Phi(\mathrm{RC},P)}$ est noté T_P^{RC}.

Lemme 3.4.4 *T_P^{RC} a un plus petit point fixe et $pppf(T_P^{\mathrm{RC}}) = T_P^{\mathrm{RC}} \uparrow \omega$.*

Preuve. Voir section 1.2. ∎

3.4.1 Lien avec la sémantique opérationnelle

Un (∞)RC-arbre de preuve est un (∞-)arbre de preuve pour le système de règle $\Phi(\mathrm{RC}, P)$. Les étiquettes de ces arbres sont des atomes contraints.

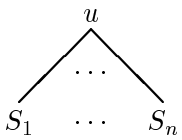
On peut voir un RC-arbre de preuve comme une "réponse renommée".

En effet, soit S une réponse et $reno_S$ une fonction de renommage pour S. Soit $pt(S, reno_S)$ l'arbre orienté sur le domaine d'arbre standard dom_S, étiqueté par des atomes contraints, défini par :

- $pt(S, reno_S)$ et S ne diffèrent que par leur fonction d'étiquetage ;
- pour tout nœud N de dom_S : si $head(reno_S(N)) = a$, et S_N est le squelette enraciné en N dans S, alors $lab_{pt(S,reno_S)}(N) = a \leftarrow \mathrm{AC}(S_N, a)$.

Lemme 3.4.5 *Si S est une réponse et $reno_S$ est une fonction de renommage pour S alors l'arbre $pt(S, reno_S)$ est un* RC*-arbre de preuve.*

Preuve. Soit S une réponse et $reno_S$ une fonction de renommage pour S. On procède par induction sur la profondeur de S.



La réponse S, ci-dessus, est enracinée par u, nom de la clause $a \leftarrow C \,\square\, a_1 \cdots a_n$.
Chaque S_i, $i = 1, \ldots, n$, est la réponse enracinée en $i-1$ dans S.

Figure 3.3 : Réponses enracinées en chaque fils d'une réponse

- Si S est une réponse de profondeur 0. S est enracinée par le nom de fait u (i.e. $S = sq(u)$). Soit $clause(u) = a \leftarrow C \,\square\, \varepsilon$ et θ tel que $reno_S(root(S)) = a\theta \leftarrow C\theta \,\square\, \varepsilon$. On a $\mathrm{AC}(S, a\theta) = \exists_{-a\theta} C\theta$.
  L'arbre A est de profondeur 0 et sa racine est étiquetée par $a\theta \leftarrow \exists_{-a\theta} C\theta$. La règle $(a\theta \leftarrow \exists_{-a\theta} C\theta) \leftarrow \varepsilon$ appartient à $\Phi(\mathrm{RC}, a \leftarrow C \,\square\, \varepsilon)$.
  Donc A est un RC-arbre de preuve.
- Supposons que toute réponse de profondeur inférieur à n vérifie la propriété.
  Si S est une réponse de profondeur $n+1$. S est enraciné par u. Soit $clause(u) = a \leftarrow C \,\square\, a_1 \cdots a_n$. Soit θ un renommage tel que $reno_S(root(S)) = (a \leftarrow C \,\square\, a_1 \cdots a_n)\theta$. Soit S_i la réponse enracinée en i dans S (voir figure 3.3) et $reno_{S_i}$ la fonction de renommage pour S_i déduite de $reno_S$ (voir lemme 2.4.2).
  Pour tout $i = 1, \ldots, n$, $pt(S_i, reno_{S_i})$ est un RC-arbre de preuve dont la racine est étiquetée par : $a_i\theta \leftarrow \mathrm{AC}(S_i, a_i\theta)$
  Les $pt(S_i, reno_{S_i})$ sont les arbres enracinés en chaque fils de la racine de A.
  La racine de A est étiquetée par : $a\theta \leftarrow \mathrm{AC}(S, a\theta)$.
  $\mathrm{AC}(S, a\theta) = \exists_{-a\theta}(C\theta \wedge \mathrm{AC}(S_1, a_1\theta) \wedge \cdots \wedge \mathrm{AC}(S_n, a_n\theta))$ (voir preuve du lemme 2.4.3.
  D'où la règle $(a\theta \leftarrow \exists_{-a\theta}(C\theta \wedge \mathrm{AC}(S_1, a_1\theta) \wedge \cdots \wedge \mathrm{AC}(S_n, a_n\theta))) \leftarrow (a_1\theta \leftarrow \mathrm{AC}(S_1, a_1\theta)) \cdots (a_n\theta \leftarrow \mathrm{AC}(S_n, a_n\theta))$ appartient à $\Phi(\mathrm{RC}, a \leftarrow C \,\square\, a_1 \cdots a_n)$ et $pt(S, reno_S)$ est un RC-arbre de preuve.

■

Inversement, on peut voir une réponse comme le "squelette d'un RC-arbre de preuve".
En effet, soit A un RC-arbre de preuve. Soit $sk(A)$ l'arbre sur le domaine d'arbre standard dom_A, étiqueté par des noms de clauses, défini par :

- A et $sk(A)$ ne diffèrent que par leur fonction d'étiquetage ;
- pour tout nœud N de dom_A : si u est la clause d'où est issue la RC-règle qui relie $lab_A(N)$ aux étiquettes des fils de N alors $lab_{sk(A)}(N) = u$.

Lemme 3.4.6 *Si A est un* RC*-arbre de preuve alors $sk(A)$ est une réponse.*

Preuve. La preuve se fait par induction sur les RC-arbres de preuve comme pour la preuve du lemme 3.4.4 en montrant qu'à partir de fonctions de renommage pour les réponses enracinées aux fils de la racine de $sk(A)$ on déduit une fonction de renommage pour $sk(A)$ comme dans la preuve du lemme 2.4.3 et que la contrainte associée à $sk(A)$ est non rejetée. ■

Lemme 3.4.7 *$pppf(T_P^{\mathrm{RC}}) = \mathrm{SS}(P)$.*

Preuve. $pppf(T_P^{\mathrm{RC}})$ est l'ensemble des étiquettes de racines de RC-arbres de preuve. D'après les deux lemmes précédents et la définition de $\mathrm{SS}(P)$, il est évident que les deux ensembles sont égaux. ■

On retrouve à nouveau les ensembles succès de la littérature :

Lemme 3.4.8 *Si le critère de rejet est $\mathrm{RC}_{\mathcal{T}}$ (i.e. le critère de rejet correct et complet pour la théorie $\mathcal{T}$) alors $\mathrm{SS}(P) = SS(P, \mathcal{T})$, où $SS(P, \mathcal{T})$ est l'ensemble succès défini dans [54].*

Nous nous intéressons maintenant au critère de rejet vide ($\mathrm{RC} = \emptyset$, c'est-à-dire que tout squelette est un état), afin de montrer les résultats généraux de correction et complétude.

Remarque. Dans la suite, le terme réponse peut être remplacé simplement par squelette fini complet. ◇

Lemme 3.4.9 *Si S est une réponse pour le but $\leftarrow a$ alors $P \models \mathrm{AC}(S, a) \rightarrow a$.*

Preuve. Soit S une réponse pour le but $\leftarrow a$ (selon le critère de rejet $\emptyset$).
Le critère de rejet $\emptyset$ est correct pour toute pré-interprétation $\mathcal{D}$. Donc si S est une réponse pour le but $\leftarrow a$ alors $P \models_{\mathcal{D}} \mathrm{AC}(S, a) \rightarrow a$ (lemme 3.2.23) et cela pour tout $\mathcal{D}$.
Or $P \models \mathrm{AC}(S, a) \rightarrow a$ signifie pour toute pré-interprétation $\mathcal{D}$: $P \models_{\mathcal{D}} \mathrm{AC}(S, a) \rightarrow a$. ■

Lemme 3.4.10 *Si $P \models C \rightarrow a$ alors $\models C \rightarrow \bigvee_{S \in success_f(a)} \mathrm{AC}(S, a)$, où $success_f(a)$ est une partie finie de $success(a)$.*

Preuve. Ce lemme est un cas particulier du lemme 3.3.14 dans le cas où $\mathcal{T} = \emptyset$. En effet l'ensemble des réponses selon le critère de rejet $\mathrm{RC}_{\emptyset}$ est inclus dans l'ensemble des réponses selon le critère de rejet $\emptyset$. ■

On peut constater que les réponses selon le critère de rejet $\emptyset$ définissent en quelque sorte les réponses générales d'un programme. C'est-à-dire que si S est un squelette fini complet pour $\leftarrow a$ alors $P \models \mathrm{AC}(S, a) \rightarrow a$. On en déduit que si $reno_S$ est une fonction de renommage pour S et $head(reno_S(root(S))) = a$ alors $P \models const(S, reno_S) \rightarrow a$. Cette propriété reste vraie même si nous supprimons la troisième condition, sur les variables existentielles, dans la définition de fonction de renommage. Mais, comme tout système de PLC, nous voulons définir les stores réponses les "plus généraux" pour des raisons d'efficacité évidentes. Cette condition sur les variables existentielles des clauses renommées n'est qu'une optimisation opérationnelle.

3.5 Selon une relation de couverture $\vdash$

On constate que la sémantique déclarative décrite selon un pré-interprétation $\mathcal{D}$ ou une théorie $\mathcal{T}$ fait intervenir une notion de couverture : C est un "store réponse au sens déclaratif" pour le but $\leftarrow a$ si C est couverte par une partie des stores réponses pour le but $\leftarrow a$.

En revanche, dans la section précédente, si RC est quelconque il manque ce résultat de complétude des réponses. Nous ne retrouvons pas de propriété similaire. Cela vient simplement du fait que nous ne disposons pas d'une notion de couverture dans ce cadre général.

L'objet de cette section est de définir une sémantique déclarative, basée sur une *relation de couverture* qui étend, à un critère de rejet quelconque, les notions de couverture dans les cas particuliers où le critère de rejet est correct par rapport à une pré-interprétation ou une théorie des contraintes.

Nous introduisons une relation abstraite, appelée *relation de couverture* entre un store et un ensemble de stores.

Définition 3.5.1 Relation de couverture
Une relation de couverture $\vdash$ *est une relation sur* $\text{STORE} \times 2^{\text{STORE}}$ *vérifiant les propriétés suivantes :*
pour toute variable x, pour tout renommage θ, pour tout store C, pour tout ensemble de stores R tel que $C \vdash R$, pour tout ensemble de stores R' tel que $R \subseteq R'$, pour toute famille finie de stores $\{C_i\}_{i\in I}$, pour toute famille finie d'ensemble de stores $\{R_i\}_{i\in I}$ telle que, pour tout $i \in I$, $C_i \vdash R_i$, pour toute famille d'ensemble de stores $\{R_{C'}\}_{C'\in R}$ telle que, pour tout $C' \in R$, $C' \vdash R_{C'}$,

RENA	$C\theta \vdash \{C'\theta \mid C' \in R\}$
TRUE	$C \vdash \{\emptyset\}$
REFL	$C \vdash \{C\}$
TRAN	$C \vdash \bigcup_{C'\in R} R_{C'}$
EXIS$_g$	$\exists x C \vdash R$, si $x \notin \bigcup_{C'\in R} var(C')$
EXIS$_d$	$C \vdash \{\exists x C' \mid C' \in R\}$
CONJ	$\bigwedge_{i\in I} C_i \vdash \{\bigwedge_{i\in I} C'_i \mid C'_i \in R_i, i \in I\}$
MONO	$C \vdash R'$

Des propriétés de la règle de couverture $\vdash$ et des équivalences entre stores, on déduit d'autres propriétés de $\vdash$: pour toute variable x, pour tout store C et C', pour tout ensemble de stores R_1 et R_2 tels que $C \vdash R_1$ et $C \vdash R_2$,

EXIS	$\exists x C \vdash \{\exists x C'_1 \mid C'_1 \in R_1\}$ (**EXIS**$_d$ et **EXIS**$_g$)
CONJ$_g$	$C \wedge C' \vdash R_1$ (**TRUE** et **CONJ**)
CONJ$_d$	$C \vdash \{C'_1 \wedge C'_2 \mid C'_1 \in R_1, C'_2 \in R_2\}$ (**CONJ** et $C \wedge C = C$ section 1.3)

Remarque. Notons qu'il est important de pouvoir considérer des ensembles infinis de stores en partie droite de la relation de couverture, si l'on veut que celle ci étende la

couverture dans une pré-interprétation qui, comme nous l'avons vu dans l'exemple 3.2.7, ne peut se ramener en général à une couverture par un ensemble fini de stores.
Remarquons que $\{\emptyset\}$ (dans la règle **TRUE**) est identifié logiquement à *true*. En effet, le store vide est identifié logiquement à *true* (élément neutre de la conjonction), donc l'ensemble de stores qui contient le store vide est identifié à *true*. Il ne doit pas être confondu avec l'ensemble vide de stores qui est identifié à *false* (élément neutre de la disjonction). ◇

(CONST, $\vdash$) définit un système de contraintes en un sens similaire à [84, 48, 18].

En effet, la sémantique des langages de programmation Concurrente avec Contraintes (CC) [85] est décrite via un système de contraintes [84, 86]. Le système de contraintes est spécifié comme un système d'information partielle, dans le style des systèmes d'information de Scott [87], enrichi des notions de variables, de quantification existentielle et de substitutions. Par définition, ces systèmes doivent vérifier des propriétés [84, 18]. Saraswat dans [85] montre l'intérêt d'un cadre basé sur la relation d'inférence sous-jacente plutôt que sur une pré-interprétation ou une théorie des contraintes.

Dans [84], la relation de conséquence est une relation sur $\mathcal{P}_f(\text{STORE}) \times \text{STORE}$ (où $\mathcal{P}_f(E)$ est l'ensemble des parties finies de E). L'ensemble de stores en partie gauche est vu comme la conjonction de ses éléments. Comme il suppose STORE fermé par conjonction (STORE est le plus petit ensemble qui contient CONST, fermé par conjonction et quantification existentielle), c'est donc une relation sur STORE $\times$ STORE. Il indique qu'il serait possible d'étendre cette relation à des ensembles finis de stores en partie droite. Mais contrairement à l'habitude en calcul des séquents, il l'étendrait de la manière suivante : $C \vdash R$ si et seulement si $C \vdash \{C'\}$ pour tout $C' \in R$. En fait, il considère l'ensemble de stores R comme la conjonction de ses éléments (il souhaite n'avoir que des séquents intuitionistes). C'est évidement différent de notre manière d'envisager l'extension puisque notre objectif est d'abstraire la notion de couverture dans une pré-interprétation ou une théorie des contraintes. Notons que nous pourrions également considérer des ensembles finis de stores en partie gauche qui seraient vu également comme la conjonction de leurs éléments. En revanche, il serait absurde de considérer des ensembles infinis en partie gauche.

Remarque. Il existe dans notre cadre une asymétrie entre les conjonctions (de contraintes basiques) et les disjonctions (de stores). On considère des ensembles de stores éventuellement infinis alors qu'on a toujours considéré des parties finies de conjonction de contraintes basiques.
Cela vient d'une différence fondamentale entre la conjonction et la disjonction : pour vérifier qu'une disjonction infinie de stores est calculée, il suffit de vérifier que chacun des stores est calculée, alors qu'une conjonction infinie ne peut s'exprimer comme une somme d'information finie calculée (on retrouve une argumentation similaire à celle de [89] section Observable properties). ◇

On retrouve la relation de conséquence des CC, à partir de la relation de couverture, quand l'ensemble en partie droite est un singleton. On constate que l'on retrouve les propriétés vérifiées par les relations de conséquence dans ce cas particulier.

Notons qu'à la différence des CC, il n'est pas question ici d'implanter un algorithme qui calcule des éléments de la relation de couverture. Cette relation sert à décrire la sémantique déclarative, non la sémantique opérationnelle décrite, quant à elle, par le critère de rejet dont l'implantation est le solveur de contraintes du système.

La définition des systèmes de contraintes de [48] utilise les algèbres cylindriques [50]. La structure d'interprétation des contraintes est un semi-anneau cylindrique clos paramétré par un système

de termes contenant un opérateur binaire de substitution $s_{\tilde{x}}$ pour tout ensemble de variables $\tilde{x}$. Intuitivement, la quantification existentielle est modélisée par les opérateurs de cylindrification et les éléments diagonaux modélisent l'unification. Il est intéressant de noter que [48] définit la sémantique par un ensemble de même nature que $\mathrm{SS}_\vee(P)$. Dans la version de 1992, les auteurs retrouvent dans leur cadre la relation de conséquence des CC et suggèrent, sans le faire, de l'étendre à des ensembles en partie droite.

En général, la relation de couverture ne sera pas quelconque. Elle ne sert qu'à décrire une sémantique déclarative en exprimant la notion de couverture d'un store par un ensemble de stores. Elle sera le plus souvent :

- déduite d'une pré-interprétation $\mathcal{D}$, elle est notée $\vdash_\mathcal{D}$ et définie par : $C \vdash_\mathcal{D} R$ si pour toute valuation dans $\mathcal{D}$ qui satisfait C il existe un store de R satisfait par cette valuation (i.e. $\models_\mathcal{D} C \to R$) ;
- déduite d'une théorie $\mathcal{T}$, elle est notée $\vdash_\mathcal{T}$ et définie par : $C \vdash_\mathcal{T} R$ si pour tout modèle $\mathcal{D}$ de $\mathcal{T}$: $C \vdash_\mathcal{D} R$ (i.e. $\mathcal{T} \models C \to R$) ;
- associée à un critère de rejet RC, c'est une relation vérifiant les propriétés et telle que C est couverte par $\emptyset$ si et seulement si $C \in \mathrm{RC}$.

On remarque que la relation de couverture généralise le critère de rejet. Le critère de rejet peut être vu comme le cas particulier $C \vdash \emptyset$.

Il est évident que $\vdash_\mathcal{D}$ et $\vdash_\mathcal{T}$ vérifient les propriétés de la définition 3.5.1.

Comme pour un critère de rejet, on peut définir la notion de relation de couverture correcte ou complète :

Définition 3.5.2 Relation de couverture correcte/complète pour $\mathcal{D}/\mathcal{T}$
Une relation de couverture $\vdash$ est correcte *pour la pré-interprétation $\mathcal{D}$ si, pour tout store C et tout ensemble de stores R, $C \vdash R$ implique $\models_\mathcal{D} C \to R$.*
Elle est complète *pour $\mathcal{D}$ si l'implication inverse est vraie.*
Une relation de couverture $\vdash$ est correcte *pour la théorie $\mathcal{T}$ si, pour tout store C et tout ensemble de stores R, $C \vdash R$ implique $\mathcal{T} \models C \to R$.*
Elle est complète *pour $\mathcal{T}$ si l'implication inverse est vraie.*

On vérifie facilement que $\vdash_\mathcal{D}$ est correcte et complète pour la pré-interprétation $\mathcal{D}$ et que $\vdash_\mathcal{T}$ est correcte et complète pour la théorie $\mathcal{T}$.

Nous définissons maintenant le système de règles sur l'ensemble des atomes contraints associé à un programme et une relation de couverture, qui décrit la sémantique déclarative du programme.

Définition 3.5.3 Système de règle associé au programme
Le système de règle associé au programme P, noté $\Phi(\vdash, P)$ se compose de deux types de règles :

les règles de programme *pour toute clause renommée $a \leftarrow C \,\square\, a_1 \cdots a_n$, et tous stores $C_1, \ldots, C_n$, on a la* règle de programme *:*

$$(a \leftarrow \exists_{-a}(C \wedge C_1 \wedge \cdots \wedge C_n)) \leftarrow (a_1 \leftarrow C_1) \cdots (a_n \leftarrow C_n)$$

les règles de couverture *pour tout atome a, tout store C et toute famille de stores $(C_i)_{i\in I}$, tels que $C \vdash \{C_i\}_{i\in I}$, on a la* règle de couverture *:*

$$(a \leftarrow C) \leftarrow (a \leftarrow C_i)_{i\in I}$$

Remarquons les deux natures de flèches imposées par les définitions (atomes contraints et règles).

Remarque. L'ensemble "d'axiomes" du système $\Phi(\vdash, P)$ est composé de deux types d'axiomes :

les axiomes de programme quand la clause est un fait (i.e. $n = 0$), on déduit l'axiome :

$$(a \leftarrow \exists_{-a} C) \leftarrow \varepsilon$$

les axiomes de couverture quand $I = \emptyset$, on déduit l'axiome :

$$(a \leftarrow C) \leftarrow \varepsilon$$

$\diamond$

L'ensemble défini inductivement par $\Phi(\vdash, P)$ représente la sémantique déclarative du programme. On le note $\mathrm{SS}_{\vdash}(P)$. Un élément de $\mathrm{SS}_{\vdash}(P)$ est appelé une *réponse déclarative.*

On définit maintenant quatre opérateurs monotones dont les points fixes seront reliés à la sémantique opérationnelle ou déclarative du programme (pour trois d'entre eux).

Définition 3.5.4 Opérateurs de conséquence immédiate
Soit T_P, $T_{\vdash}$, $T^{\cup}_{\vdash,P}$ et $T^{\circ}_{\vdash,P}$ les quatre opérateurs des ensembles d'atomes contraints dans les ensembles d'atomes contraints définis par :

- $T_P(I) = \{a \leftarrow C \mid$ *il existe une règle de programme* $(a \leftarrow C) \leftarrow (a_1 \leftarrow C_1) \ldots (a_n \leftarrow C_n)$ $\{a_i \leftarrow C_i\}_{i \in \{1,\ldots,n\}} \subseteq I\}$
- $T_{\vdash}(I) = \{a \leftarrow C \mid$ *il existe une règle de couverture* $(a \leftarrow C) \leftarrow (a \leftarrow C')_{C' \in R}$ $\{a \leftarrow C'\}_{C' \in R} \subseteq I\}$
- $T^{\cup}_{\vdash,P}(I) = T_P(I) \cup T_{\vdash}(I)$
- $T^{\circ}_{\vdash,P}(I) = T_{\vdash}(T_P(I))$

Lemme 3.5.5 $pppf(T_P) = T_P \uparrow \omega$

Preuve. T_P est compact car les règles de programmes sont finitaires. ■

Un élément de $pppf(T_P)$ est appelé une *réponse opérationnelle.* Nous verrons dans la section 3.5.1 le lien entre $pppf(T_P)$ et $\mathrm{SS}(P)$.

Lemme 3.5.6 $pppf(T^{\cup}_{\vdash,P}) = \mathrm{SS}_{\vdash}(P)$

Preuve. Voir section 1.2. ■

L'opérateur T_P est l'opérateur associé aux règles de programme, l'opérateur $T^{\cup}_{\vdash,P}$ est l'opérateur associé aux deux types de règles. Le plus petit point fixe de T_P correspond aux racines d'arbres de preuve n'utilisant que les règles de programme, le plus petit point fixe de $T^{\cup}_{\vdash,P}$ correspond aux racines d'arbres de preuve. Le plus petit point fixe de $T^{\circ}_{\vdash,P}$ correspond aux arbres de preuves qui utilisent aux profondeurs paires des règles de programme et aux profondeurs impaires des règles de couverture. Le plus petit point fixe de $T_{\vdash}$ est sans intérêt ici.

Nous énonçons maintenant le lemme exprimant la complétude des réponses calculées.

Lemme 3.5.7 $\mathrm{SS}_\vdash(P) = \{a \leftarrow C \mid$ *il existe* $R, C \vdash R$, *pour tout* $C' \in R, a \leftarrow C' \in \mathit{pppf}(T_P)\}$

Preuve. On prouve le lemme par induction sur les règles de $\Phi(\vdash, P)$ en distinguant les deux cas correspondant aux deux types de règles.

1. Soit $(a \leftarrow \exists_{-a}(C \wedge C_1 \wedge \cdots \wedge C_n)) \leftarrow (a_1 \leftarrow C_1) \cdots (a_n \leftarrow C_n)$ une règle de programme.
   Supposons que pour tout $i = 1, \ldots, n$, $a \leftarrow C_i \in \mathrm{SS}_\vdash(P)$ et il existe un ensemble de stores R_i tel que $C_i \vdash R_i$ et $a_i \leftarrow C'_i \in \mathit{pppf}(T_P)$, pour tout $C'_i \in R_i$.
   Montrons qu'il existe R tel que $\exists_{-a}(C \wedge C_1 \wedge \cdots \wedge C_n) \vdash R$ et $a \leftarrow C' \in \mathit{pppf}(T_P)$, pour tout $C' \in R$.
   La règle de programme est issue de la clause renommée $a \leftarrow C \,\square\, a_1 \cdots a_n$.
   La propriété **REFL** de la relation de couverture assure que
   $$C \vdash \{C\}$$
   La propriété **CONJ** de la relation de couverture assure que
   $$C \wedge C_1 \wedge \cdots \wedge C_n \vdash \{C \wedge C'_1 \wedge \cdots \wedge C'_n \mid C'_i \in R_i, i = 1, \ldots, n\}$$
   La propriété **EXIS** de la relation de couverture assure que
   $$\exists_{-a}(C \wedge C_1 \wedge \cdots \wedge C_n) \vdash \{\exists_{-a}(C \wedge C'_1 \wedge \cdots \wedge C'_n) \mid C'_i \in R_i, i = 1, \ldots, n\}$$
   Donc, il suffit de prendre $R = \{\exists_{-a}(C \wedge C'_1 \wedge \cdots \wedge C'_n) \mid C'_i \in R_i, i = 1, \ldots, n\}$.
2. Soit $(a \leftarrow C) \leftarrow (a \leftarrow C')_{C' \in R'}$ une règle de couverture.
   Supposons que, pour tout $C' \in R'$, $a \leftarrow C' \in \mathrm{SS}_\vdash(P)$ et il existe un ensemble de store $R_{C'}$ tel que $C' \vdash R_{C'}$ et $a \leftarrow C'' \in \mathit{pppf}(T_P)$, pour tout $C'' \in R_{C'}$.
   Montrons qu'il existe un ensemble de stores R tel que $C \vdash R$ et $a \leftarrow C'' \in \mathit{pppf}(T_P)$, pour tout $C'' \in R$.
   La règle de couverture est issue de $C \vdash R'$.
   La propriété **TRAN** de la relation de couverture assure que
   $$C \vdash \bigcup_{C' \in R'} R_{C'}$$
   Donc, il suffit de prendre $R = \bigcup_{C' \in R'} R_{C'}$.

■

Corollaire 3.5.8 $\mathit{pppf}(T^{\cup}_{\vdash,P}) = T_\vdash(\mathit{pppf}(T_P))$ *et* $\mathit{pppf}(T^{\cup}_{\vdash,P}) = T^{\cup}_{\vdash,P} \uparrow \omega + 1$.

C'est-à-dire, pour tout arbre de preuve enraciné par $a \leftarrow C$, il existe un arbre de preuve enraciné aussi par $a \leftarrow C$ qui utilise une unique règle de couverture, et cette règle de couverture est utilisée à la racine de l'arbre de preuve.

On retrouve les résultats bien connus de complétude des réponses opérationnelles dans le cas où la relation de couverture est $\vdash_{\mathcal{D}}$ ou $\vdash_{\mathcal{T}}$.

Remarque. Notons qu'un élément de $\mathit{pppf}(T^{\cup}_{\vdash_{\mathcal{T}},P})$ est une réponse selon [97].
On remarque que dans le résultat de [97], appelé *strong completeness*, il n'est pas utile d'imposer $\mathcal{T} \models \tilde{\exists} C$ (voir le lemme 3.3.14). ◇

Intéressons nous maintenant à l'opérateur $T^{\circ}_{\vdash,P}$. Il est facile de vérifier que cet opérateur est monotone. Par conséquent il a un plus petit point fixe.

On ne définira pas le système de règle associé à cet opérateur, il peut se déduire facilement à partir des compositions des règles de programme et des règles de couverture de $\Phi(\vdash, P)$.

Lemme 3.5.9 $pppf\,(T^{\circ}_{\vdash,P}) = pppf\,(T^{\cup}_{\vdash,P}) = T_{\vdash}(pppf\,(T_P))$

Preuve. La seconde égalité est déjà prouvée. Pour la première :

- $pppf\,(T^{\circ}_{\vdash,P}) \supseteq T_{\vdash}(pppf\,(T_P))$ car la propriété **REFL** des règles de couverture assure que pour tout store C, $C \vdash \{C\}$.
- $pppf\,(T^{\circ}_{\vdash,P}) \subseteq pppf\,(T^{\cup}_{\vdash,P})$ car pour tout ordinal α, il existe un ordinal β tel que $T^{\circ}_{\vdash,P} \uparrow \alpha \subseteq T^{\cup}_{\vdash,P} \uparrow \beta$: pour tout $\alpha \in \mathbb{N}$, il suffit de prendre $\beta = 2\alpha$, donc $T^{\circ}_{\vdash,P} \uparrow \omega \subseteq T^{\cup}_{\vdash,P} \uparrow \omega$, de la même manière on a $T^{\circ}_{\vdash,P} \uparrow \omega + 1 \subseteq T^{\cup}_{\vdash,P} \uparrow \omega + 2$, or $pppf\,(T^{\circ}_{\vdash,P}) \supseteq T_{\vdash}(pppf\,(T_P))$ d'après le point précédent et $T_{\vdash}(pppf\,(T_P)) = T^{\cup}_{\vdash,P} \uparrow \omega + 1 = pppf\,(T^{\cup}_{\vdash,P})$, donc $pppf\,(T^{\circ}_{\vdash,P}) \subseteq pppf\,(T^{\cup}_{\vdash,P})$.

■

Nous avons montré que le plus petit point fixe de $T^{\circ}_{\vdash,P}$ est égal à $\mathrm{SS}_{\vdash}(P)$. Cela signifie que tout élément de $\mathrm{SS}_{\vdash}(P)$ enracine un arbre de preuve alternant règles de programme et règles de couverture.

En résumé, les quatre ensembles

1. $pppf\,(T^{\circ}_{\vdash,P})$
2. $pppf\,(T^{\cup}_{\vdash,P})$
3. $T_{\vdash}(pppf\,(T_P))$
4. $\mathrm{SS}_{\vdash}(P)$

sont égaux ce qui exprime

1. le lien entre les réponses opérationnelles et les réponses déclaratives (3.=4.) ;
2. une forme de compositionalité des réponses déclaratives (1.=4.).

Il est important de noter que seule les propriétés de $\vdash$ sont utiles pour montrer ce résultat. Rappelons que les arbres de preuve pour $\Phi(\vdash, P)$ ne sont pas des arbres finis : les règles de couvertures ne sont pas nécessairement finitaires (cas de $\vdash_{\mathcal{D}}$). Ce sont seulement des arbres bien-fondés.

Notons que cette question de compositionalité ne semble pas abordée dans la littérature. Nous verrons dans le chapitre 4 toute l'importance de cette question.

3.5.1 Lien avec la sémantique opérationnelle

On suppose que le critère de rejet RC est la relation unaire $\vdash \emptyset$, i.e. $C \in \mathrm{RC}$ si et seulement si $C \vdash \emptyset$.

Remarque. Les arbres de preuve qui n'utilisent que des règles de programmes de $\Phi(\vdash, P)$ sont exactement les RC-arbres de preuve si $\mathrm{RC} = \emptyset$ (l'ensemble des règles de programmes est $\Phi(\mathrm{RC}, P)$, où $\mathrm{RC} = \emptyset$). On en déduit le lemme qui suit. ◇

Lemme 3.5.10 $\mathrm{SS}(P) = \{a \leftarrow C \mid a \leftarrow C \in \mathit{pppf}(T_P), C \notin \mathrm{RC}\}$.
$a \leftarrow C \in \mathit{pppf}(T_P)$ si et seulement si il existe un squelette fini complet S pour $\leftarrow a$ et une fonction de renommage $reno_S$ pour S et $\leftarrow a$ tels que $\mathrm{AC}(S) = C$.
C est un store réponse au but $\leftarrow a$ si et seulement si $a \leftarrow C \in \mathit{pppf}(T_P)$ et $C \notin \mathrm{RC}$.

Preuve. D'après la remarque. ∎

Lemme 3.5.11 $\mathrm{SS}_\vdash(P) = T_\vdash(\mathrm{SS}(P))$

Preuve.

$\supseteq$ $\mathrm{SS}(P) \subseteq \mathit{pppf}(T_P)$ et $T_\vdash$ est monotone.

$\subseteq$ Soit $a \leftarrow C \in \mathrm{SS}_\vdash(P)$. D'après le lemme 3.5.7, il existe un ensemble de stores R tel que $C \vdash R$ et $\{a \leftarrow C'\}_{C' \in R} \subseteq \mathit{pppf}(T_P)$.
Pour tout $C' \in R$, soit $R_{C'}$ l'ensemble de stores défini par : $R_{C'} = \emptyset$ si $C' \vdash \emptyset$ et $R_{C'} = \{C'\}$ sinon. Il est clair que, pour tout $C' \in R$: $C' \vdash R_{C'}$. Donc d'après **TRAN**, on a $C \vdash \bigcup_{C' \in R} R_{C'}$. D'après le lemme 3.5.10, pour tout $C'' \in \bigcup_{C' \in R} R_{C'}$: $a \leftarrow C'' \in \mathrm{SS}(P)$.
Donc $\mathrm{SS}_\vdash(P) \subseteq T_\vdash(\mathrm{SS}(P))$

∎

Ce lemme exprime la complétude des réponses calculées par rapport à une relation abstraite de couverture. L'ensemble des résultats de complétude des sections précédentes (selon une pré-interprétation, selon une théorie) sont des cas particuliers de celui-ci. On retrouve la couverture finie (par exemple selon une théorie) quand la relation de couverture est compacte :

Définition 3.5.12 Relation de couverture compacte
Une relation de couverture $\vdash$ est compacte si pour tout store C, pour tout ensemble de store R : $C \vdash R$ implique $C \vdash R_f$, où R_f est une partie finie de R.

La relation $\vdash_{\mathcal{T}}$ est compacte parce que la logique du premier ordre est compacte. Notons la similitude avec la notion de compacité en topologie [89].

La correction des réponses calculées est exprimée par le lemme suivant :

Lemme 3.5.13 $\mathrm{SS}(P) \subseteq \mathrm{SS}_\vdash(P)$

Preuve. $\mathrm{SS}(P) \subseteq \mathit{pppf}(T_P) \subseteq \mathrm{SS}_\vdash(P)$ ∎

Chapitre 4

Diagnostic déclaratif d'erreur

> "Of the two aspects of debugging (locating the error and correcting it), the first represents perhaps 95% of the problem."
>
> Glenford J. Myers, The Art of Software Testing [67], page 130.

Parmi les étapes de développement d'un programme, il en existe une inévitable : c'est la localisation des erreurs. Pour des langages de haut niveau, comme les langages de programmation logiques, les techniques traditionnelles de traces deviennent rapidement difficiles à utiliser à cause de la complexité du comportement opérationnel des systèmes (par exemple le backtracking). De plus, il serait incohérent de n'utiliser que des outils de mise au point de bas niveau alors que pour ces langages l'accent est mis sur la sémantique déclarative (i.e. une sémantique indépendante du modèle d'exécution).

> "It is evident that a computer can neither construct nor debug a program without being told, in one way or another, what problem the program is supposed to solve, and some constraints on how to solve it."
>
> Ehud Y. Shapiro, Algorithmic Program Debugging, [88], page 1.

Mais ici, seules des propriétés déclaratives attendues sont requises.

Nous nous intéressons dans ce chapitre à la *correction partielle* des programmes logiques avec contraintes.

Un *symptôme* est le résultat inattendu d'un calcul. Si le résultat d'un calcul est un symptôme, c'est que le calcul a un résultat, c'est donc un calcul fini. Notre objectif est de déterminer les raisons de ce résultat inattendu en ne faisant référence qu'à des propriétés déclaratives attendues pour le programme (par exemple, une interprétation attendue).

On souhaite, par ailleurs, que ces raisons soient modélisées par une partie du code du programme la plus petite possible, plus, éventuellement, des informations expliquant pourquoi cette partie de code est jugée erronée. Ce couple, partie de code et explication, sera appelé *erreur*. Le résultat désiré est : s'il existe un symptôme alors il existe une erreur (la contraposée, à relier aux problèmes de validation, est : s'il n'existe pas d'erreur alors il n'existe pas de symptôme). De plus, nous aimerions avoir une méthode effective qui, étant donné un symptôme localise une erreur. C'est le *diagnostic déclaratif d'erreur*.

Quand il n'existe pas de symptôme, on dit que le programme est *partiellement correct*. Quand il n'existe pas d'erreur, on dit qu'il est *correct*.

Remarque. Notons qu'un programme peut être partiellement correct sans être correct : il existe une erreur dans le programme, mais tout calcul fini produit un résultat attendu. ◇

Il existe une autre grande catégorie de symptômes qui correspondent à la non-terminaison inattendue d'un calcul. Dans ce cas, le symptôme n'est pas le résultat du calcul mais le calcul lui-même. Ce type de symptôme n'est pas traité ici ; mais il existe des travaux dans ce sens, comme par exemple [83] qui étend les travaux de [5] et [25] basé sur la sémantique de [24].

Comme nous avons distingué dans le chapitre 2 deux niveaux de calcul, nous distinguons deux types de symptôme :

- les *symptômes d'incorrection partielle positifs*, appelé plus simplement symptômes positifs ; la notion de calcul qui leur est liée est la dérivation SLD fini ;
- les *symptômes d'incorrection partielle négatifs*, appelé plus simplement symptômes négatifs ; la notion de calcul qui leur est liée est l'arbre SLD.

Soit P un programme et RC un critère de rejet.

Supposons que $a \rightarrow \mathrm{SS}(P,a) \in \mathrm{SS}_{\vee}(P)$ et que $\mathrm{SS}(P,a)$ ne soit pas un $\vee$-store réponse attendue pour $\leftarrow a$. Deux cas peuvent se présenter :

1. au moins un store de $\mathrm{SS}(P,a)$ ne devrait pas faire partie de $\mathrm{SS}(P,a)$;
2. il manque au moins un store dans $\mathrm{SS}(P,a)$.

Dans le premier cas, le store réponse non attendu vient d'une SLD-dérivation. Seule cette SLD-dérivation est concernée pour la recherche de l'erreur. C'est donc une notion de symptôme lié à ce type de calcul fini et il n'est pas nécessaire d'imposer des restrictions quant à la finitude de l'arbre SLD. La notion de symptôme positif correspond à ce type de symptôme.

Dans le second cas, c'est la notion de symptôme négatif qui correspond. Supposons qu'un arbre SLD fini fournit un $\vee$-store réponse qui ne convient pas. C'est qu'il manque au moins une branche succès à l'arbre SLD.

Les symptômes positifs correspondent à la terminologie plus classique de symptôme d'incorrection [58], alors que les symptômes négatifs correspondent à la notion classique de symptôme d'incomplétude [92].

Le changement de terminologie est dû à la nouveauté de notre approche. En plus de fournir un formalisme dans le but de diagnostiquer des erreurs dans les programmes logiques avec contraintes, nous traitons le problème de réponses manquantes comme un problème d'incorrection en considérant une autre notion de réponse. Cette notion de réponse est similaire à celle donné dans [63] pour la sémantique du langage ESHER et par [81] pour la sémantique du langage Prolog IV.

Dans la section 4.1, nous présentons le schéma général du diagnostic déclaratif d'erreur dégagé dans [41, 43]. Dans la section 4.2 nous présentons, vu dans le cadre de la section 4.1, l'état de l'art du diagnostic déclaratif d'erreur (essentiellement pour les programmes logiques purs). Dans la section 4.3, nous présentons un nouveau schéma général pour le diagnostic déclaratif. L'intérêt de ce nouveau schéma est d'étudier le problème des réponses manquantes comme un problème d'incorrection. De plus, on explique enfin complètement des algorithmes connus dans le cadre de la programmation logique qui étaient mal compris parce qu'ils étaient reliés au diagnostic déclaratif d'insuffisance. Dans la section 4.4 on étudie le cas des symptômes positifs, dans la section 4.5 celui des symptômes négatifs. Dans la section 4.6 on voit le problème des réponses manquantes de manière plus classique.

4.1 Symptômes et erreurs pour un système de règles : Premier schéma général

Dans la section 1.2 nous avons introduit les définitions inductives. Nous rappelons ici les définitions de symptômes et d'erreurs et leurs liens dans ce cadre [43, 41]. Dans la section 4.2 nous verrons une première application de ce premier schéma général de diagnostic d'erreur.

Soit E un ensemble et Φ un ensemble de règles sur E.

4.1.1 Symptômes, erreurs et diagnostic

Étant donné un ensemble $I \subseteq E$, supposons que nous souhaitons $ind(\Phi) \subseteq I$, mais que nous constatons qu'il existe un $x \in ind(\Phi) \setminus I$. Dans ce cas, x est appelé un *symptôme* de Φ par rapport à I.

S'il existe un symptôme alors $ind(\Phi) \not\subseteq I$, donc I n'est pas clos par T_Φ, i.e $T_\Phi(I) \not\subseteq I$; par conséquent, il existe une règle $x \leftarrow X \in \Phi$ telle que $X \subseteq I$ mais $x \notin I$. Cette règle $x \leftarrow X$ est appelée une *erreur* de Φ par rapport à I.

Il est facile de voir qu'un élément de E est parfois symptôme à cause d'un autre symptôme, c'est-à-dire, il est la conclusion d'une règle dont une prémisse est un symptôme. Il n'est pas très intéressant de désigner cette règle comme la règle responsable du symptôme. Une erreur est bien plus intéressante : c'est une règle dont la conclusion est un symptôme alors qu'aucune de ses prémisses n'en est un. De plus, s'il n'y a pas d'erreur alors il n'y a pas de symptôme. La conclusion d'une erreur est une sorte de *symptôme minimal*.

Supposons que toutes les règles de Φ soient finitaires.

Étant donné un arbre de preuve enraciné par un symptôme, l'idée d'un algorithme de diagnostic d'erreur est de vérifier si les étiquettes des nœuds de l'arbre sont des symptômes : il existe un nœud étiqueté par un symptôme alors qu'aucun de ses fils ne l'est. La règle qui relie l'étiquette de ce nœud aux étiquettes de ses fils est une erreur. Comme l'arbre de preuve est fini, l'algorithme termine.

4.1.2 Co-symptômes, co-erreurs et diagnostic

Du point de vue dual, un *co-symptôme* de Φ par rapport à $I \subseteq E$ est un élément de $I \setminus coind(\Phi)$. Une *co-erreur* de Φ par rapport à I est un élément x de I tel qu'il n'existe aucune règle $x \leftarrow X \in \Phi$ avec $X \subseteq I$ (i.e. $x \in I \setminus T_\Phi(I)$).

S'il existe un co-symptôme alors $coind(\Phi) \not\supseteq I$, donc I n'est pas supporté par T_Φ, i.e. $T_\Phi(I) \not\supseteq I$; par conséquent, il existe une co-erreur.

Notons que s'il n'y a pas de co-erreur alors il n'y a pas de co-symptôme. De plus, s'il n'existe ni erreur ni co-erreur alors I est un point fixe de T_Φ.

Supposons que toutes les règles de Φ soient finitaires.

Étant donné un co-symptôme x, l'idée d'un algorithme de diagnostic de co-erreur est d'essayer de construire un ∞-arbre de preuve enraciné par x dont l'ensemble des étiquettes est inclus dans I. C'est impossible. Si c'était possible alors l'ensemble X des étiquettes des nœuds de cet arbre serait supporté par T_Φ, donc $X \subseteq coind(\Phi)$, or $x \in X$. On construit une suite d'arbre de preuve "partiel"[1] enracinés par x dont les étiquettes sont dans I. Chaque arbre A_{i+1} de la suite (sauf le

[1] Un arbre de preuve partiel est un arbre de preuve avec hypothèses, i.e. certaines de ses feuilles sont étiquetées par des éléments qui ne sont pas nécessairement des conclusions de règles dont l'ensemble des prémisses est vide. Ces feuilles (ou leurs étiquettes) sont appelées des hypothèses.

premier) est obtenu à partir du précédent A_i : on choisit une hypothèse $N \in dom_{A_i}$ et une règle $lab_{A_i}(N) \leftarrow X \in \Phi$ telle que $X \subseteq I$, $A_{i+1} = graft(A_i, N, A)$, où A est l'arbre de preuve partiel de profondeur 1 enraciné par $lab_{A_i}(N)$ dont l'ensemble des feuilles est X. On trouve une "hypothèse" $x' \in I$ tel qu'il n'existe aucune règle de conclusion x' dont les prémisses sont dans I (sinon on construit un ∞-arbre de preuve enraciné par x). Cet élément x' est une co-erreur. L'algorithme termine si la tentative de construction de l'arbre n'est pas quelconque : par exemple, on peut le construire par niveaux.

4.2 Survol du débogage déclaratif en programmation logique

Nous rappelons, en nous appuyant sur le premier schéma général les principaux travaux sur le diagnostic déclaratif d'erreur.

4.2.1 Symptôme et erreur pour les programmes logiques

On a une première application très simple du schéma général (section 1.2) si l'on considère un programme logique défini P, la base de Herbrand H (i.e. l'ensemble des atomes clos), et le système de règles constitué des instances closes des clauses de P [62, 2]. L'opérateur associé au système de règles est généralement noté T_P.

$pppf(T_P)$ est le plus petit modèle de Herbrand de P et c'est aussi l'ensemble des atomes clos conséquence logique de P. En ce sens, il formalise la *sémantique déclarative* (close) (positive) de P.

Soit I un ensemble d'atomes clos formalisant des propriétés attendues pour le programme P.

Un symptôme de T_P par rapport à I est un atome représentant une réponse close fausse de P et est appelée un *symptôme d'incorrection* de P par rapport à I.

Une erreur est appelée une *incorrection* de P par rapport à I. Trivialement, s'il n'y a pas d'incorrection alors I est un modèle de Herbrand de P.

> *Remarque.* Notons que le schéma général s'applique également si l'on considère d'autres formalisations par point fixe de la sémantique des programmes : par exemple, la sémantique avec variables [40, 32] ou la s-sémantique [15]. ◇

Les propriétés attendues, formalisées par l'ensemble I, ne constituent pas obligatoirement une spécification complète du programme P. En d'autres termes, on désire seulement avoir $pppf(T_P) \subseteq I$, mais l'égalité n'est pas nécessairement attendue : par exemple les propriétés attendues peuvent concerner la forme des atomes, le typage, etc. [70, 78].

Les notions duales de co-symptôme et co-erreur interviennent quand on s'intéresse à la *sémantique négative* (close) du programme P, à cause de l'ensemble d'*échec fini* de P, parce qu'il est inclus dans $H \setminus pgpf(T_P)$.

Un co-symptôme de T_P par rapport à I est appelé un *symptôme d'insuffisance* de P par rapport à I. C'est une notion abstraite qui peut s'appliquer aux réponses manquantes (closes) effectivement calculées, c'est-à-dire au cas où un atome appartient à I mais appartient aussi à l'ensemble d'échec fini de P.

Une co-erreur est appelée *insuffisance* de P par rapport à I. Le terme *insuffisance* [88] (et non pas *incomplétude*) fait ressortir la différence entre $I \subseteq pgpf(T_P)$ et $I \subseteq pppf(T_P)$. Il est intéressant de constater que l'insuffisance est une bonne notion d'erreur : elle explique le symptôme dans le but de corriger le programme. En particulier, une autre notion d'erreur qui expliquerait uniquement la raison pour laquelle il y a échec fini ne serait pas une bonne notion d'erreur car l'échec fini est

une propriété opérationnelle du programme P seul, alors que les bonnes notions de symptômes et d'erreur doivent aussi dépendre des propriétés attendues de P. De plus, il est intéressant qu'elles ne dépendent que de propriétés indépendantes du comportement opérationnel.

Cette approche théorique est bien pratique parce que les notions de symptôme et d'erreur et les relations entre elles sont très claires (et simple) dans ce cadre inductif. De plus, cette approche s'applique aux symptômes effectivement calculés par la résolution SLD.

Dans ce schéma général, avec d'une part le plus petit point fixe de T_P et d'autre part son plus grand point fixe, on peut considérer l'information négative et l'information positive ensemble. Le schéma se généralise aux programmes logiques avec négation (*programmes normaux*) [41, 11] (la sémantique de Fitting [44] en donne une vision logique). A nouveau tout symptôme est relié à une erreur, mais maintenant toute interaction est possible entre les deux types de symptômes et les deux types d'erreurs à travers les négations : un symptôme d'incorrection (respectivement d'insuffisance) peut être relié à une insuffisance (respectivement une incorrection).

On peut également retrouver l'approche de Lloyd [61, 62] basé sur une sémantique logique du complété de Clark [17], la résolution SLDNF et la notion d'*interprétation attendue* I, où l'absence d'erreur signifie que I est un modèle du complété du programme. En fait I modèle du complété de P est un cas particulier de I point fixe de l'opérateur de Fitting associé à P.

Pour le cas des réponses manquantes, une autre notion d'erreur est également étudiée [78, 37, 69] appelée *atome non complètement couvert* (ou *insuffisance faible* [94]). Il s'agit d'un atome éventuellement avec variables dont une instance est un *atome non couvert* (i.e. une insuffisance). Cette notion d'erreur est plus faible (elle comporte moins d'information) que la notion d'insuffisance, mais intervient dans des algorithmes de diagnostics (voir section 4.2.2) qui ont une interaction avec l'oracle moins complexe.

4.2.2 Diagnostic déclaratif pour les programmes logiques

Bien entendu l'ensemble I de propriétés attendues pour le programme n'est pas toujours disponible. Ce n'est qu'une formalisation théorique en terme d'ensemble du concept d'*oracle* [88] i.e. la manière par laquelle l'algorithme de diagnostic peut obtenir de l'information sur les propriétés attendues du programme (formalisées par I).

Considérons un arbre de preuve enraciné par un atome formalisant une réponse fausse. L'étiquette de sa racine appartient à $pppf(T_P)$ mais n'est pas attendue. C'est un symptôme d'incorrection de P par rapport aux propriétés attendues I.

L'algorithme de diagnostic interroge l'oracle sur les nœuds de l'arbre de preuve : l'oracle indique si les étiquettes des nœuds sont attendues (pour simplifier si le nœud est attendu). On montre (par induction sur les arbres de preuve) qu'il existe un nœud qui n'est pas attendu alors que tout ses fils le sont : la racine n'est pas attendue et l'arbre est fini. La règle reliant ce nœud à ces fils est une erreur de T_P par rapport à I. Elle correspond à une *instance de clause incorrecte*, c'est-à-dire une incorrection de P par rapport à I. L'algorithme peut retourner la première erreur détectée, ou encore l'ensemble des erreurs ayant une occurrence dans l'arbre de preuve.

Toute stratégie pour localiser une erreur dans l'arbre de preuve peut être envisagée. Par exemple la stratégie descendante (qui interroge les fils d'un nœuds de gauche à droite en profondeur d'abord) [88, 40, 61, 72]. En général, ces algorithmes s'arrêtent dès que la première incorrection est détectée. Aussi, changer de stratégie peut optimiser le nombre de questions posées à l'oracle[2]. Cela peut

[2]Une des préoccupation les plus importante dans le diagnostic déclaratif est de minimiser le nombre de questions à l'oracle, et que ces questions soient les plus simples possible. Il ne faut pas oublier que le concept théorique

changer également la sortie de l'algorithme, i.e. l'incorrection trouvée. Des stratégies meilleures en moyenne (en nombre de question à l'oracle) que la stratégie descendante (ou ascendante) sont par exemple la stratégie "diviser pour régner" (divide-and-query) [88, 14], ou des stratégies basées sur des heuristiques [79, 75].

Une autre manière de réduire le nombre de questions à l'oracle est d'avoir stocké ses précédentes réponses afin de ne plus l'interroger sur des questions dont la réponse est subsumée par des réponses à des questions précédentes [88, 68, 16].

Une spécification partielle de la sémantique attendue du programme fournie à l'algorithme de diagnostic peut encore réduire le nombre de questions [28, 37]. Drabent et al. [37] suggèrent d'utiliser les assertions, qui peuvent être "attachées" aux symboles de prédicats du programme, comme spécification partielle. Dans [34] une spécification exécutable est utilisée pour générer des tests, localiser des erreurs et guider la correction, par l'intermédiaire de techniques d'inférences déductive et inductive (voir aussi [33, 59]). Notons tout de même qu'une spécification *complète et exécutable* du programme est une hypothèse un peu forte !

L'intérêt du diagnostic déclaratif par rapport aux techniques de traces est maintenant clair :

1. L'utilisateur n'a pas besoin de comprendre le comportement opérationnel du système (les arbres de preuve sont intrinsèques au programme, i.e. ils ne dépendent pas de la stratégie de résolution).

2. L'algorithme suit le chemin direct du symptôme jusqu'à l'erreur (évitant les explorations inutiles dues au backtracking).

3. Il n'affiche que de l'information pertinente (c'est-à-dire indispensable à la localisation de l'erreur).

4. Il aide le programmeur à se poser les bonnes questions sur son programme.

5. Il fournit au programmeur une clause fautive, mais aussi les conditions de son incorrection formalisées par une instance de clause incorrecte.

6. L'ensemble des points précédents facilite la correction du programme.

Pour le problème des réponses manquantes, deux types de diagnostic sont envisagés. Le premier type recherche une insuffisance (un atome non couvert) [88, 40, 61] alors que le second recherche une insuffisance faible (un atome non complètement couvert) [78, 37, 69].

La donnée, pour le premier type d'algorithmes, est un symptôme d'insuffisance, c'est-à-dire un atome attendu formalisant une réponse manquante.

Intuitivement, l'algorithme de diagnostic tente de construire un arbre de preuve enraciné par le symptôme d'insuffisance. Naturellement, la tentative doit échouer (si la construction est équitable). Supposons que l'on ait déjà un arbre de preuve partiel, c'est-à-dire un arbre de preuve dont certaines feuilles sont des hypothèses. L'algorithme choisit une hypothèse de l'arbre de preuve (une feuille non étiquetée par un fait) et demande à l'oracle une instance de clause dont la conclusion est l'hypothèse et les prémisses sont attendues dans le but de "greffer" cette règle à la feuille. Quand aucune instance de clause n'a été trouvée, l'hypothèse est un atome non couvert.

---

d'oracle se traduit dans la réalité par le concept concret de programmeur. Dans la majorité des implantations, c'est le programmeur qui répond aux questions posées sur les propriétés attendues, même quand une spécification partielle de ces propriétés est fournie à l'algorithme de diagnostic : ce dernier peut toujours être amené à poser une question sur la partie non spécifiée des propriétés attendues.

Cette fois toute stratégie n'est pas une bonne stratégie, contrairement aux algorithmes de diagnostic d'incorrection. Une bonne stratégie est, par exemple, de construire l'arbre de preuve partiel par niveaux. D'autres stratégies peuvent être utilisées, guidées par des heuristiques, dans le but de choisir la feuille qui minimisera le nombre de questions. Il faut souligner que l'interaction avec l'oracle est assez compliquée : il doit fournir des substitutions (même s'il est aidé dans cette démarche par l'algorithme de diagnostic).

La donnée, pour le second type d'algorithmes est un symptôme d'incomplétude, i.e. un atome qui a un instance attendue formalisant une réponse manquante.

L'interaction avec l'oracle est plus facile que celle du premier type d'algorithmes. En effet, celui-ci n'a pas à fournir de substitution. Il ne répond que par `oui` ou `non` aux questions d'incomplétude [37, 78]. L'idée est de vérifier si toute instance attendue d'un atome a est instance d'un atome formalisant une réponse au but $\leftarrow a$.

Mais, l'algorithme, quant à lui, est plus compliqué. De plus, il suppose l'existence d'un arbre SLD fini sans co-routinage [69, 93] (ou un arbre SLD fini standard [37]) à partir du symptôme d'incomplétude (dans le but d'avoir des questions d'incomplétude finies). Naish dans [69] recherche des possibilités afin de supprimer cette condition de non co-routinage dans certain cas.

Le Rationnal Debugging [78] invoque des dépendances entre termes et utilise conjointement la sémantique déclarative et la sémantique opérationnelle en utilisant une notion d'"appels inadmissibles".

Les algorithmes de diagnostic décris ci-dessus ont été implantés pour plusieurs systèmes de programmation logique. Les programmes de diagnostic sont souvent des méta-programmes, et leur implantation soulève des problèmes de représentation du programme objet [51, 91]. De ce point de vue, le langage Gödel [52] offre des facilités de manipulation de programmes au niveau objet. Binks a développé le système de diagnostic déclaratif GRADE [14] pour Gödel, écrit en Gödel, incluant des outils pour les types abstraits de données et le co-routinage. Un des composants du système de mise au point NUDE [74], implanté par Naish, est un diagnostic déclaratif. Le système HyperTracer environment utilisant des heuristiques [78] a été implanté pour C-prolog par Calejo [16].

4.2.3 Extensions

Le diagnostic déclaratif a été étendu à des propriétés non déclaratives. Par exemple, dans un contexte de programmation logique concurrente, Abstract Algorithmic Debugging [60] réduit la complexité des questions posées à l'oracle par le biais d'abstractions.

Plus récemment, Abstract Diagnosis [98, 27, 28, 29] étend les méthodes de diagnostic déclaratif en combinant la s-sémantique [15] et les techniques d'interprétation abstraite [30]. La méthode est basée sur la comparaison entre la spécification I_α et $T_{P,\alpha}(I_\alpha)$ pour un $T_{P,\alpha}$ adéquat, où α est un observable [26]. Pour certains observables (par exemple l'ensemble des réponses calculées à une profondeur n) les auteurs donnent une méthode de diagnostic d'incorrection indépendante des symptômes. Ils donnent également une méthode de diagnostic d'incomplétude pour une grande classe de programme incluant les programmes acceptables [4] (en tirant parti de l'unicité du point fixe[3] de T_P). Le diagnostic n'est généralement pas effectif parce que I est infini, mais le devient si une sémantique abstraite finie adéquate est considérée [29].

[3] Ils supposent que tout programme que l'on souhaite écrire a un unique point fixe (c'est le cas de tous les programmes proposés dans [90] par exemple). De plus, ils supposent que les versions erronées de ces programmes ont également un unique point fixe. Il se trouve que l'expérimentation montre que dans la majorité des cas ces hypothèses s'avèrent vérifiées.

Le diagnostic déclaratif à été également étendu à d'autres types de langage de programmation : des langages logico-fonctionnels comme NUE-Prolog [73], Escher [63]. Une caractéristique typique du cadre du diagnostic en Escher est sa simplicité, principalement parce que le calcul n'est pas décrit explicitement par un arbre, mais par des équations et un calcul déterministe.

D'autres langages ont aussi été considérés : les langages fonctionnels paresseux [76, 71] et même des langages impératifs [46] incluant un sous-ensemble de Pascal. Dans le schéma de [72], Naish décrit un cadre pour la mise au point déclarative de programmes logiques orientés objet.

Des travaux récents étendent les diagnostic déclaratif à la programmation logique avec contraintes. La principale difficulté est que les interprétations de Herbrand ne représentent plus la sémantique des programmes. De plus, peu de langages de contraintes ont la propriété d'indépendance des contraintes négatives (INC [65]), ce qui complique le diagnostic d'insuffisance (l'algorithme proposé dans la section 4.2.2 ne peut convenir si INC n'est pas vérifiée). Naish annonce dans [72] un prototype implanté en CLP($\mathcal{R}$) basé sur son schéma (malheureusement, je n'ai pas trouvé d'autres références à ces travaux).

Cette thèse est une contribution à l'extension à la programmation logique avec contraintes. [95] fournit un cadre formel inductif basé sur la vision grammaticale [32] pour étendre le diagnostic déclaratif au langages de PLC. [58] (réponses fausses) abstrait l'interprétation des contraintes par un critère de rejet pour tenir compte de l'incomplétude des solveurs de contraintes. [92] étend le critère de rejet par une relation de couverture et définit les insuffisances et les insuffisance faibles dans un cadre inductif unique [42].

Un dernier point est le "problème de présentation" [63] qui consiste à trouver des moyens de présenter à l'oracle (souvent le programmeur) des questions à l'origine longues et complexes sous une forme convenablement simplifiée pour que celui-ci puisse y répondre correctement. Ce point est au moins aussi important en programmation logique avec contraintes qu'il l'est en programmation logique pure.

4.3 Symptômes et erreurs pour une relation bien fondée : deuxième schéma général

Dans la section 4.1 nous avons décrit les notions de symptômes, symptôme minimal et erreur du premier schéma général [43] basé sur les définitions inductives [1].

Il se peut que l'on éprouve parfois des difficultés à exprimer un système de règles qui conduise à une notion de symptôme minimal acceptable. Il se peut aussi que l'on dispose d'une bonne notion de symptôme minimal, mais que les systèmes de règles pour cette notion soient compliqués et non intuitifs.

Nous proposons dans cette section une autre approche du schéma général du diagnostic déclaratif basé sur le fait que : un algorithme de diagnostic est essentiellement la recherche d'un symptôme minimal selon une relation bien fondée.

Par exemple, supposons que l'on ait une relation binaire sur les états de calcul et que la restriction de cette relation à un sous-ensemble des états de calcul soit bien fondée (la notion de relation bien fondée est intimement liée à la notion de calcul fini, preuve de terminaison, etc.). Alors le schéma que nous proposons s'applique immédiatement. De plus, si le sous-ensemble des états de calcul considéré est fini, alors il suffit, pour montrer que la relation est bien fondée, de montrer que sa clôture transitive est irréflexive. En général, c'est très simple.

Soit E un ensemble. Soit R une relation binaire bien fondée sur E. Soit $I \subseteq E$ l'ensemble des éléments de E attendus. Supposons que $I \neq E$, c'est-à-dire $E \setminus I \neq \emptyset$.

Définition 4.3.1 Symptôme pour une relation bien fondée
Un symptôme de R par rapport à I est un élément de $E \setminus I$.

Comme R est bien fondée, $E \setminus I$ admet au moins un élément minimal (selon R^+).

Définition 4.3.2 Symptôme minimal pour une relation bien fondée
Un symptôme minimal de R par rapport à $I \subset E$ est un élément minimal de $E \setminus I$ (selon R^+).

Lemme 4.3.3 *S'il existe un symptôme de R par rapport à I alors il existe un symptôme minimal de R par rapport à I.*

Preuve. R^+ est un ordre bien fondé. ∎

Si à chaque symptôme minimal on sait associer une notion d'erreur alors: s'il n'existe pas d'erreur, il n'existe pas de symptôme.

4.3.1 Diagnostic d'erreur pour une relation bien fondée

Le diagnostic est aussi simple que le schéma lui-même.

Un algorithme de diagnostic évident consiste à construire une suite $x_0, x_1, \ldots, x_i, \ldots$, de symptômes tels que pour tout i $(x_{i+1}, x_i) \in R$, i.e. une suite décroissante de symptômes. La suite est obligatoirement finie, puisque R est bien fondée.

Si l'on construit une suite (finie) décroissante de symptômes dont le dernier élément n'a pas de prédécesseur qui soit un symptôme alors ce dernier élément est un symptôme minimal.

4.3.2 Liens avec les définitions inductives

Soit R une relation binaire bien fondée sur E. Soit Φ_R l'ensemble de règles sur E défini par: pour tout $x \in E$, soit $X \subseteq E$ l'ensemble des prédécesseurs de x, $x \leftarrow X$ est une règle de Φ_R.

Remarque.

1. Le système de règles Φ_R est non ambiguë, i.e. chaque élément de E est la conclusion d'au plus une règle (exactement une dans notre cas).

   L'ensemble défini inductivement par Φ_R est l'ensemble E tout entier.

2. Dans la section 1.2.1 nous avons défini les arbres de preuve pour des systèmes de règles finitaires. Nous avons profité de la définition des domaines d'arbre standard pour définir l'ensemble des nœuds d'un arbre de preuve. Mais ce qui est important c'est la structuration en arbre des nœuds plutôt que la nature même des nœuds. On peut définir les arbres de preuve même pour des systèmes non finitaires (cependant, leurs domaines d'arbre ne peuvent plus être standards quand l'ensemble des prémisses des règles ne sont plus dénombrables). Un arbre de preuve est un arbre étiqueté bien fondé (sans branche infinie) tel que l'étiquette de chaque nœud est la conclusion d'une règle dont les prémisses sont les étiquettes des fils du nœuds.

   On montre toujours que l'ensemble des racines d'arbres de preuve est l'ensemble défini inductivement par les règles.

3. On remarque que pour Φ_R, tout élément de E enracine un unique arbre de preuve. De plus, comme le système est non ambiguë, on constate que l'on peut prendre comme domaine d'arbre d'un arbre de preuve l'ensemble de ces étiquettes et chaque nœud correspond à son étiquette (cela devient faux si l'on veut définir les ∞-arbres de preuve).

   Les arbres de preuve pour Φ_R sont éventuellement infinis si les règles ne sont pas finitaires, mais il ne doivent pas êtres confondus avec les ∞-arbres de preuves : toute branche d'un arbre de preuve est finie.

$\diamond$

Les notions de symptôme et symptôme minimal de R par rapport à $I \subset E$ correspondent aux notions de symptôme et symptôme minimal de Φ_R par rapport à I de façon tout à fait évidente.

Il semble que les définitions par rapport à une relation bien fondée soient un cas particulier des définitions par rapport à un ensemble de règles : quand celui-ci est non ambiguë.

En fait, il s'agit de deux formalismes équivalents.

Soit Φ un ensemble de règles sur E. Soit $adp(\Phi)$ l'ensemble des arbres de preuve pour Φ.

Soit R_Φ la relation binaire sur $adp(\Phi)$ définie par : $(A', A) \in R_\Phi$ si et seulement si il existe $i \in \mathbb{N}$ tel que A' est l'arbre de preuve enraciné en i dans A. R_Φ est bien fondée (les branches des arbres de preuve sont finies).

Soit $I \subseteq E$ tel que $ind(\Phi) \not\subseteq I$. C'est à dire qu'il existe au moins un symptôme de Φ par rapport à I. Les symptômes de Φ par rapport à I sont des éléments de $ind(\Phi)$.

Soit I_{R_Φ} l'ensemble des arbres de preuves enracinés par un élément de $ind(\Phi) \cap I$. D'une part, $I_{R_\Phi} \subseteq adp(\Phi)$. D'autre part, $adp(\Phi) \backslash I_{R_\Phi} \neq \emptyset$ puisque $ind(\Phi) \backslash I \neq \emptyset$ ($adp(\Phi) \backslash I_{R_\Phi}$ est l'ensemble des arbres de preuve enracinés par les symptôme de Φ par rapport à I). Donc, il existe des symptômes de R_Φ par rapport à I_{R_Φ}.

Un symptôme minimal de R_Φ par rapport à I_{R_Φ} est enraciné par un symptôme de Φ par rapport à I et est tel que tout ses prédécesseurs sont enracinés par des éléments de I, i.e. il est enraciné par un symptôme minimal de Φ par rapport à I.

Réciproquement tout symptôme minimal de Φ par rapport à I enracine un symptôme minimal de R_Φ par rapport à I_{R_Φ} puisqu'un symptôme minimal correspond à un symptôme qui enracine un arbre de preuve dont tous les éléments étiquetant les fils de sa racine (ils enracine des "sous-arbre de preuve") sont dans I.

C'est simplement dû au fait que deux arbres enracinés par le même élément et qui ont les mêmes "sous-arbres" sont identiques.

En fait, les deux formalismes ont le même pouvoir d'expression parce que les principes de preuve par induction (structurelle [6]) et de preuves par induction bien fondée sont équivalent : "l'un simule l'autre". Ce qui se montre facilement dans notre cadre.

Soit Φ un ensemble de règles sur E. Le principe de *preuve par induction structurelle* d'une propriété P sur les éléments de $ind(\Phi)$ consiste à vérifier que, pour toute règle $x \leftarrow X \in \Phi$, si les éléments de X vérifient P alors x vérifie P.

Soit R une relation binaire bien fondée sur E. Le principe de *preuve par induction bien fondée* d'une propriété P sur les éléments de E consiste à vérifier que, pour tout $x \in E$, si tout prédécesseur de x vérifie P alors x vérifie P.

En utilisant les deux techniques consistant à passer de R à Φ_R et de Φ à R_Φ, on montre que toute preuve par induction structurelle se ramène à une preuve par induction bien fondée et réciproquement.

Nous avons donc deux schémas généraux équivalents pour le diagnostic déclaratif. Il est intéressant de disposer du second, car une relation bien fondée simple peut parfois aboutir à un système de règle inutilement compliqué comme nous le verrons par la suite (section 4.5).

De plus, la notion de relation bien fondée est intrinsèquement liée à la notion de calcul fini. Par exemple, dans le chapitre 2 tout calcul est décrit par un arbre ; quand le calcul est fini, la réciproque de sa relation de parenté est bien fondée. Mais il existe d'autres relations bien fondées sur les états de ces calculs finis, et nous verrons que certaines conduisent à des notions intéressantes de symptôme minimal.

Remarque. Notons que nous n'avons pas défini la notion d'erreur pour le schéma basé sur une relation bien fondée. Il y a une explication simple : nous n'avons pas défini la structure à partir de laquelle se définit la relation bien fondée.
Dans le premier schéma on pouvait donner une notion d'erreur parce qu'on donnait un système de règles Φ (une erreur est une règle de Φ).
Les algorithmes de diagnostic recherchent en fait un symptôme minimal. C'est une fois que ce symptôme minimal est trouvé qu'on l'explique par une notion d'erreur issue par exemple d'un système de règles ou d'un programme. ◇

Remarque. Nous n'avons pas cherché à définir de notion duale (co-symptôme) pour une relation bien fondée. C'est parce que le problème des réponses manquantes sera traité comme celui des réponses fausses par un algorithme du type recherche d'incorrection. En fait, une réponse manquante peut s'expliquer par une incorrection d'un système de règle. Ce système de règles est toutefois compliqué, mais heureusement, il existe une relation bien fondée simple qui lui "correspond". C'est à partir de cette relation bien fondée que nous rechercherons dans le programme les causes d'une réponse manquante. ◇

4.4 Correction partielle positive : réponses fausses

On suppose un programme P et un critère de rejet RC fixés.

Rappelons que la notion de symptôme est liée à la notion de calcul fini dont le résultat est inattendu. Dans cette section nous allons examiner le cas des symptômes pour le premier niveau de calcul : les dérivations SLD.

Considérons une SLD-dérivation succès pour le but $\leftarrow a$ dont l'état final S est telle que $a \leftarrow \mathrm{AC}(S, a)$ n'est pas attendu.

L'état complet S peut être construit à partir des sous-squelettes greffés en ses nœuds. D'après le Lemme 2.4.2 chacun de ses sous-squelettes est une réponse.

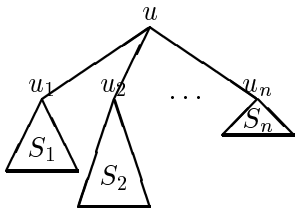
Soit $<$ la relation binaire sur l'ensemble des réponses définie par $S' < S$ si il existe $i \in \mathbb{N}$ tel que S' est la réponse enracinée en i dans S (voir figure 4.1).

Lemme 4.4.1 *$<$ est bien fondée.*

Preuve. Les réponses sont finies et si $S' < S$ alors $card(dom_{S'}) < card(dom_S)$, où $card(X)$ est le cardinal de l'ensemble X. ■

Étant donné un ensemble de réponses attendues, d'après le schéma général de la section 4.3, on déduit immédiatement les définitions de symptôme et symptôme minimal.

Soit S la réponse



pour tout $i = 1, \ldots, n$, $S_i < S$.
$(S = graft(\cdots(graft(graft(sq(u), 0, S_1), 1, S_2), \ldots), n-1, S_n))$

Figure 4.1 : Relation $<$ sur les réponses

Il est bien entendu que l'on ne peut décemment exiger de l'utilisateur que sa formalisation des propriétés attendues pour le programme P soit un ensemble de réponses attendues (i.e. une ensemble de squelettes finis complets non rejetés). Ce qu'il faut c'est disposer d'un critère déclaratif qui permette de décider si une réponse est attendue. À l'origine de la mise en œuvre d'une méthode de diagnostic, on a un symptôme calculé. Ce symptôme a surgi parce que l'utilisateur a posé un but $\leftarrow a$ et que le système a fourni un store réponse C pour le but $\leftarrow a$ jugé anormal par l'utilisateur. Il est anormal parce que $a \leftarrow C$ n'est pas attendu alors qu'il appartient à $\mathrm{SS}(P)$. L'utilisateur doit savoir déterminer si les éléments de $\mathrm{SS}(P)$ sont attendus (c'est ainsi qu'il observe des symptômes en testant son programme). $\mathrm{SS}(P)$ est l'ensemble succès associé au type de calcul considéré : les dérivations SLD.

Les propriétés attendues sont formalisées par un objet de même nature que $\mathrm{SS}(P)$, c'est-à-dire, un ensemble d'atomes contraints, noté I. L'ensemble I doit être fermé par renommage. En effet, si S est une réponse pour $p \in \Pi_p$ d'arité n alors pour toute séquence de variables disjointes $x_1, \ldots, x_n$, par définition, $\mathrm{AC}(S, p(x_1, \ldots, x_n))$ est un store réponse pour $\leftarrow p(x_1, \ldots, x_n)$. C'est-à-dire qu'il existe une forme d'équivalence des stores réponses modulo les variables du but : si C est un store réponse pour $\leftarrow a$ alors $C\theta$ est un store réponse pour $\leftarrow a\theta$. Remarquons que $a \leftarrow \mathrm{AC}(S, a)$ ne dépend que de S.

Remarque. Il est important de noter que des propriétés attendues formalisées en termes d'atomes contraints attendus sont différentes de propriétés attendues formalisées en termes de squelettes finis complets attendus. En effet, les propriétés attendues doivent avoir certaines propriétés leur permettant d'être les propriétés attendues d'un programme, i.e. il doit exister un programme (idéal) qui vérifie ces propriétés. Il est conseillé qu'un "sous-squelette" d'un squelette attendu soit également attendu, sinon les propriétés attendues seraient en contradiction avec le principe même de construction des squelettes (cf. réponses manquantes).
Si l'on considère un diagnostic déclaratif, la notion de squelette attendu n'est plus pertinente, mais on peut retenir du squelette le store associé.
C'est pourquoi les propriétés attendues sont formalisées en termes d'atomes contraints attendus. D'une part, cela permet de déterminer si une réponse est attendue. D'autre part, dans le cadre du diagnostic, on ne cherche qu'à aider le programmeur pour que tous les calculs finis fournissent un store réponse jugée correcte, par ce dernier, vis-à-vis du but posé.

En fait on peut considérer qu'un programme ayant la propriété que tout calcul fini fournit un résultat attendu est d'un certain point de vue satisfaisant (même s'il contient des erreurs), c'est ce qu'on appelle la *correction partielle*. Cette propriété n'est qu'en partie satisfaisante, car toute extension du programme peut faire surgir des symptômes dus à des erreurs du programme d'origine. C'est pourquoi les méthodes de validation au sens preuve de correction (voir la section **??**) quand elles peuvent être mise en œuvre sont un complément indispensable au diagnostic. ◇

La sémantique partielle attendue formalisée par I étant dorénavant fixée, l'oracle peut déterminer si une réponse S pour $\leftarrow a$ est attendue : S est attendue si $a \leftarrow \mathrm{AC}(S, a) \in I$. Cela permet de préciser la notion de symptôme.

Définition 4.4.2 Symptôme d'incorrection partielle positive
Un symptôme d'incorrection partielle positive *de P par rapport à I est un atome contraint $a \leftarrow C \in \mathrm{SS}(P) \setminus I$.*
Pour alléger le texte on l'appelle symptôme positif.

La cause d'un symptôme de $<$ par rapport à I est un symptôme minimal, mais quelle est la cause du symptôme minimal ?

Soit S un symptôme minimal. Soit $\mathit{clause}(\mathit{lab}_S(\mathit{root}(S))) = a \leftarrow C \,\square\, a_1 \cdots a_n$. S est un symptôme donc $a \leftarrow \mathrm{AC}(S, a) \in \mathrm{SS}(P) \setminus I$. Il est minimal donc si $S_1, \ldots, S_n$ sont les réponses enracinées en $i-1$ dans S alors $a_i \leftarrow \mathrm{AC}(S_i, a_i) \in I$. Le symptôme minimal vient du fait que $a \leftarrow \exists_{-a}(C \wedge \bigwedge_{i\in\{1,\ldots,n\}} \mathrm{AC}(S_i, a_i)) \in \mathrm{SS}(P) \setminus I$ alors que chaque $a_i \leftarrow \mathrm{AC}(S_i, a_i) \in \mathrm{SS}(P) \cap I$ (voir le lemme 2.4.3). La clause $a \leftarrow C \,\square\, a_1 \cdots a_n$ est l'origine du symptôme et le store $\bigwedge_{i\in\{1,\ldots,n\}} \mathrm{AC}(S_i, a_i)$ en est une explication.

Nous associons maintenant une notion d'erreur à tout symptôme minimal.

Définition 4.4.3 Incorrection partielle positive
Une incorrection partielle positive *(ou plus simplement une* incorrection positive*) de P par rapport à I est un $n+1$-uplet $\langle a \leftarrow C \,\square\, a_1 \cdots a_n, C_1, \ldots, C_n\rangle$, où $a \leftarrow C \,\square\, a_1 \cdots a_n \in P$ et les C_i sont des stores, tel que $\{a_i \leftarrow C_i \mid i \in \{1, \ldots, n\}\} \subseteq I$, mais $a \leftarrow \exists_{-a}(C \wedge C_1 \wedge \cdots \wedge C_n) \notin I$.*

Lemme 4.4.4 *S'il existe un symptôme positif de P par rapport à I alors il existe une incorrection positive de P par rapport à I.*

Preuve. Il existe un symptôme positif si et seulement si il existe un symptôme. Il existe un symptôme si et seulement si il existe un symptôme minimal. S'il existe un symptôme minimal alors il existe une incorrection positive. ■

La réciproque du lemme 4.4.4 est, en général, fausse. Il est facile de trouver des exemples où la clause d'une incorrection positive n'intervient dans aucun calcul fini. Par contre, un corollaire intéressant est sa contraposée :

Corollaire 4.4.5 *S'il n'existe pas d'incorrection positive alors il n'existe pas de symptôme positif.*

D'où l'intérêt des méthodes effectives de validation (quand elles existent).

Remarque. Les notions de symptôme positif et d'incorrection positive de P par rapport à I sont exactement les notions de symptôme et d'erreur de $\Phi(\mathrm{RC}, P)$ par rapport à I. Nous verrons dans la section 4.5 qu'un système de règles qui donnerait les mêmes notions de symptômes et d'erreurs qu'une relation bien fondée simple peut être extrêmement compliqué.
La raison pour laquelle nous avons préféré définir une relation bien fondée plutôt que réutiliser le système de règles $\Phi(\mathrm{RC}, P)$ réside dans la généralisation possible des définitions quand la sémantique déclarative fait référence à une pré-interprétation des contraintes ou une relation de couverture, comme nous le constaterons plus tard. ◇

4.4.1 Diagnostics d'incorrection positive

On fait appel au diagnostic d'incorrection positive quand on observe un symptôme positif. C'est-à-dire, une dérivation SLD U pour le but $\leftarrow a$ a abouti au store réponse C et $a \leftarrow C$ n'est pas attendu. L'état final de U est une réponse S qui est un symptôme de $<$.

Pour simplifier, dans la suite on dit qu'une réponse S pour $\leftarrow a$ est (non) attendue si $a \leftarrow \mathrm{AC}(S, a)$ est (non) attendu.

Soit S une réponse. Soit $<_S$ la restriction de $<$ à l'ensemble $\mathrm{SA}(S) = \{S' \mid \text{il existe } N \in dom_S, S'$ est enracinée en N dans $S\}$.
$<_S$ est bien fondée.
On note $>_S$ la relation ${<_S}^{-1}$.

Lemme 4.4.6 $(\mathrm{SA}(S), >_S)$ *est un arbre bien fondé (fini).*

Preuve. Triviale. La racine de $(\mathrm{SA}(S), >_S)$ est S. Il est facile de voir d'après la définition de $<_S$ que

- pour tout $S' \in \mathrm{SA}(S)$, $S {>_S}^* S'$;
- $(S, S) \notin >_S$;
- pour tout $S' \in \mathrm{SA}(S) \setminus \{S\}$, il existe un unique S'' tel que $S'' >_S S'$: si S' est enraciné en $N \cdot i$ dans S alors S'' est la réponse enracinée en N dans S.

$(\mathrm{SA}(S), >_S)$ est bien fondé puisque $<_S$ est bien fondée (voir section 1.1), il est même fini puisque $\mathrm{SA}(S)$ est fini. ■

Cet arbre s'oriente facilement en se donnant l'ensemble d'ordres $\{<_{(S,S')}\}_{S' \in \mathrm{SA}(S)}$ tel que, pour tout nœud $S' \in \mathrm{SA}(S)$, si S' est enraciné en N dans S, si N a n fils et S'_i est la réponse enracinée en $N \cdot (i-1)$, $i = 1, \ldots, n$, dans S alors $S'_j <_{(S,S')} S'_k$ si $j < k$, pour tout $j, k = 1, \ldots, n$.
De plus, si l'on étiquette cet arbre par l'ensemble des atomes contraints avec la fonction d'étiquetage $lab_{(\mathrm{SA}(S),>_S)}$ définie par $lab_{(\mathrm{SA}(S),>_S)}(S') = a' \leftarrow \mathrm{AC}(S', a')$, où $a' = head(clause(lab_{S'}(root(S'))))$, alors on retrouve un arbre similaire à un arbre de preuve pour $\Phi(\mathrm{RC}, P)$.

Intuitivement le diagnostic consiste à chercher un symptôme minimal dans l'arbre $(\mathrm{SA}(S), >_S)$, mais l'algorithme s'exprime de manière encore plus simple sur la réponse S directement.
Comme $S'' {<_S}^* S$ si et seulement si S'' est enraciné en un nœud de dom_S dans S, tout parcours des nœuds de $(\mathrm{SA}(S), >_S)$ revient à un parcours des nœuds de S en considérant les arbres enracinés en ces nœuds.

Soit S une réponse pour $\leftarrow a$ telle que $a \leftarrow \mathrm{AC}(S, a) \notin I$. On dit qu'un nœud $N \in dom_S$ est attendu si la réponse enracinée en N dans S est attendue. L'algorithme consiste à vérifier pour

$go1(x,y) \leftarrow x = 1+1 \wedge y = 1+1$ est un symptôme positif de FIB' par rapport à I_{fib}.
Ce symptôme positif est calculé par la réponse S_1 :

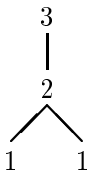


Figure 4.2 : Symptôme positif pour le programme FIB'

chaque nœud de dom_S s'il est attendu. Il existe un nœud $N' \in dom_S$ non attendu alors que tous ces fils sont attendus. Soit S' la réponse enracinée en N' dans S. Soit $clause(lab_{S'}(root(S'))) = a' \leftarrow C' \,\square\, a'_1 \cdots a'_n$. Soit S'_i la réponse enracinée en $i-1$ dans S', $i = 1, \ldots, n$. Il est clair que $\langle a' \leftarrow C' \,\square\, a'_1 \cdots a'_n, \mathrm{AC}(S'_1, a'_1), \ldots, \mathrm{AC}(S'_n, a'_n)\rangle$ est une incorrection positive.

Illustrons le diagnostic d'incorrection positive sur le petit exemple calculant la fonction de Fibonacci, en considérant une version erronée du programme FIB de l'exemple 1.3.2 (cet exemple est similaire à celui traité dans [58]).

Exemple 4.4.1 Fibonacci
Soit FIB' le programme :

$0 : fib(x,y) \leftarrow x = 0 \wedge y = 1 \,\square\, \varepsilon$
$1 : fib(x,y) \leftarrow x = 1 \wedge y = 1 \,\square\, \varepsilon$
$2 : fib(x,y) \leftarrow 1 < x \wedge x = x_1 + 1 \wedge x = x_2 + 1 \wedge y = y_1 + y_2 \,\square\, fib(x_1,y_1) fib(x_2,y_2)$
$3 : go1(x,y) \leftarrow x = y \,\square\, fib(x,y)$
$4 : go2(x,y) \leftarrow x = 1+1 \,\square\, fib(x,y)$

Supposons que le critère de rejet soit $\mathrm{RC}_{\mathcal{N}}$, où $\mathcal{N}$ est la pré-interprétation des contraintes dont le domaine est $\mathbb{N}$ avec l'interprétation habituelle pour les symboles de Σ et Π_c.

Remarque. On peut remarquer que l'ensemble des $\mathcal{D}$-atomes de la forme $fib(n_1, n_2)$ du plus petit $\mathcal{N}$-modèle de FIB' est $\{fib(n_1,n_2) \mid \text{si } n_1 = 0 \text{ alors } n_2 = 1 \text{ sinon } n_2 = 2^{n_1-1}\}$. ◇

L'ensemble d'atomes contraints qui formalise les propriétés attendues de FIB' est noté I_{fib}. Il s'agit de l'ensemble $\mathrm{SS}(FIB)$ de l'exemple 2.7.1.
Le store-réponse $\exists x_1 \exists x_2 \exists y_1 \exists y_2 (x = y \wedge 1 < x \wedge x = x_1 + 1 \wedge x = x_2 + 1 \wedge y = y_1 + y_2 \wedge x_1 = 1 \wedge y_1 = 1 \wedge x_2 = 1 \wedge y_2 = 1)$, écrit plus simplement $x = 1+1 \wedge y = 1+1$, pour le but $\leftarrow go1(x,y)$ est jugé non attendu, c'est-à-dire que $go1(x,y) \leftarrow x = 1+1 \wedge y = 1+1$ n'appartient pas à l'ensemble d'atomes contraints I_{fib} : c'est un symptôme positif de FIB' par rapport à I_{fib}.
La réponse S_1 concernée est présentée par la figure 4.2.
Supposons que l'on interroge les nœuds de S_1 selon le parcours préfixe alors la session de diagnostic serait :

$go1(x,y) \leftarrow x = 1+1 \wedge y = 1+1$ attendu ?	`NON`
$fib(x,y) \leftarrow x = 1+1 \wedge y = 1+1$ attendu ?	`NON`
$fib(x,y) \leftarrow x = 1 \wedge y = 1$ attendu ?	`OUI`
$fib(x,y) \leftarrow x = 1 \wedge y = 1$ attendu ?	`OUI`

Incorrection positive :

$$\langle \quad fib(x,y) \leftarrow 1 < x \wedge x = x_1 + 1 \wedge x = x_2 + 1 \wedge y = y_1 + y_2 \,\square\, fib(x_1,y_1) fib(x_2,y_2),\ x_1 = 1 \wedge y_1 = 1,\ x_2 = 1 \wedge y_2 = 1 \quad \rangle$$

On remarque que l'on peut toujours éviter la première question puisque l'entrée de l'algorithme est un symptôme. De plus, dans cet exemple, on peut éviter la dernière question si l'on a stocké la question et la réponse qui précède. Par conséquent, il a suffit de deux questions posées à l'oracle pour localiser une erreur dans le programme.

Voici, à titre d'exemple, le début de la trace fournie par `clp(FD)` [35] :

```
| ?- go1(X,Y).
      1    1 Call: go1(_31,_32) ?
      2    2 Call: (external) _31#=_32 ?
      2    2 Exit: (external) _31#=_31 ?
      3    2 Call: fib(_31,_31) ?
      4    3 Call: (external) _31#=0 ?
      4    3 Exit: (external) 0#=0 ?
      5    3 Call: (external) 0#=1 ?
      5    3 Fail: (external) _112#=1 ?
      3    2 Redo: fib/2 ?
      4    3 Call: (external) _31#=1 ?
      4    3 Exit: (external) 1#=1 ?
      5    3 Call: (external) 1#=1 ?
      5    3 Exit: (external) 1#=1 ?
      3    2 Exit: fib(1,1) ?
      1    1 Exit: go1(1,1) ?

X = 1
Y = 1 ? ;
      3    2 Redo: fib/2 ?
      4    3 Call: (external) _31#=_135+1 ?
      4    3 Exit: (external) 1..268435455#=0..268435454 +1 ?
      5    3 Call: (external) 1..268435455#=_144+1 ?
      5    3 Exit: (external) 1..268435455#=0..268435454 +1 ?
      6    3 Call: (external) 1..268435455#=_153+_154 ?
      6    3 Exit: (external) 1..268435455#=0..268435455 +0..268435455 ?
      7    3 Call: fib(0..268435454,0..268435455) ?
      8    4 Call: (external) 0..268435454#=0 ?
      8    4 Exit: (external) 0#=0 ?
      9    4 Call: (external) 0..1#=1 ?
      9    4 Exit: (external) 1#=1 ?
      7    3 Exit: fib(0,1) ?
     10    3 Call: fib(0,0) ?
     11    4 Call: (external) 0#=0 ?
     11    4 Exit: (external) 0#=0 ?
     12    4 Call: (external) 0#=1 ?
```

```
 12    4 Fail: (external) _532#=1 ?
 10    3 Redo: fib/2 ?
 11    4 Call: (external) 0#=1 ?
 11    4 Fail: (external) 0#=1 ?
 10    3 Redo: fib/2 ?
 11    4 Call: (external) 0#=_555+1 ?
 11    4 Fail: (external) 0#=0..268435455 +1 ?
 10    3 Fail: fib/2 ?
  7    3 Redo: fib/2 ?
  8    4 Call: (external) 0..268435454#=1 ?
  8    4 Exit: (external) 1#=1 ?
  9    4 Call: (external) 0..2#=1 ?
  9    4 Exit: (external) 1#=1 ?
  7    3 Exit: fib(1,1) ?
 10    3 Call: fib(1,1) ?
 11    4 Call: (external) 1#=0 ?
 11    4 Fail: (external) _546#=0 ?
 10    3 Redo: fib/2 ?
 11    4 Call: (external) 1#=1 ?
 11    4 Exit: (external) 1#=1 ?
 12    4 Call: (external) 1#=1 ?
 12    4 Exit: (external) 1#=1 ?
 10    3 Exit: fib(1,1) ?
  3    2 Exit: fib(2,2) ?
  1    1 Exit: go1(2,2) ?

X = 2
Y = 2 ?
```

On peut comparer l'efficacité du diagnostic déclaratif (deux questions) pour aider l'utilisateur à corriger son programme (il fournit une clause incorrecte ainsi que des contraintes expliquant son incorrection), avec l'efficacité de la trace sur ce petit exemple (pour le but `fib(n,Y)`, le nombre de lignes affichées par la trace croît de manière géométrique par rapport à la valeur de `n`).

4.4.2 Diagnostic d'incorrection positive selon une relation de couverture $\vdash$

Dans la section 3.5, nous avons décrit la sémantique déclarative du programme selon une relation de couverture. Il est naturel d'envisager une sémantique attendue de même nature.

Soit $\vdash$ une relation de couverture. Le critère de rejet RC est défini par la relation unaire $\vdash \emptyset$.

Les propriétés attendues sont toujours formalisées par un ensemble d'atomes contraints I. On suppose maintenant que l'ensemble d'atomes contraints attendus I est tel que si $a \leftarrow C \in I$ et $C' \vdash \{C\}$ alors $a \leftarrow C' \in I$.

La notion de symptôme positif est légèrement modifiée puisqu'on remplace $\mathrm{SS}(P)$ par $\mathrm{SS}_{\vdash}(P)$.

Définition 4.4.7 Symptôme positif
Un symptôme (d'incorrection partielle) positif *de P par rapport à I selon $\vdash$ est un atome contraint de* $\mathrm{SS}_{\vdash}(P) \setminus I$.

On remarque que les symptômes positifs de P par rapport à I sont des symptômes positifs de P par rapport à I selon $\vdash$. En effet, $\mathrm{SS}(P) \subseteq \mathrm{SS}_{\vdash}(P)$.

Pour distinguer les deux notions de symptômes, un élément de $\mathrm{SS}(P) \setminus I$ est appelé un *symptôme positif calculé.*

La définition d'incorrection positive demeure inchangée (elle est déterminée par la sémantique attendue et ne fait pas référence à la sémantique de fait). On rappelle la définition :

Définition 4.4.8 Incorrection positive
Une incorrection positive *de P par rapport à I est un $n+1$-uplet $\langle a \leftarrow C \square a_1 \cdots a_n, C_1, \ldots, C_n \rangle$, où $a \leftarrow C \square a_1 \cdots a_n \in P$ et les C_i sont des stores, tel que $\{a_i \leftarrow C_i \mid i \in \{1, \ldots, n\}\} \subseteq I$, mais $a \leftarrow \exists_{-a}(C \wedge C_1 \wedge \cdots \wedge C_n) \notin I$.*

Lemme 4.4.9 *S'il existe un symptôme positif de P par rapport à I selon $\vdash$ alors il existe une incorrection positive de P par rapport à I.*

> *Preuve.* On remarque que $\langle a \leftarrow C \square a_1 \cdots a_n, C_1, \ldots, C_n \rangle$ est une incorrection positive si, pour tout $i = 1, \ldots, n$, $a_i \leftarrow C_i \in I$ et $a \leftarrow \exists_{-a}(C \wedge C_1 \wedge \cdots \wedge C_n) \notin I$. C'est-à-dire, si $a \leftarrow \exists_{-a}(C \wedge C_1 \wedge \cdots \wedge C_n) \in T^{\cup}_{\vdash,P}(I) \setminus I$. Or, $\mathrm{SS}_{\vdash}(P) = \mathit{pppf}(T^{\cup}_{\vdash,P})$. D'où le résultat. ■

Par conséquent, on retrouve de manière différente le lemme 4.4.4 pour les symptômes positifs calculés dans ce cadre plus général.

Nous montrons que, partant d'un symptôme, on peut décrire une famille d'algorithmes qui généralise celle décrite en section précédente.

Soit S une réponse.

Dans la section précédente, nous avons considéré la relation bien fondée $<_S$ pour définir la notion de symptôme minimal.

On a vu qu'il suffit de considérer S, la relation de parenté de S et la réponse enracinée en chaque nœud de S pour l'algorithme de diagnostic d'incorrection positive.

Pour déterminer si la réponse S' enracinée en un nœud N' de S est attendue, on interroge l'oracle sur un atome contraint associé à S' (ou de manière équivalente à N'). Dans la section précédente, c'était $a' \leftarrow \mathrm{AC}(S', a')$ (S' est une réponse pour $\leftarrow a'$).

Nous changeons maintenant l'atomes contraint associé à nœud de S.

L'atome contraint associé au nœud $N \in \mathit{dom}_S$ est noté $a_N \leftarrow C_N$.

Ces atomes contraints doivent vérifier la condition suivante : Il existe une fonction de renommage reno_S pour S telle que, pour tout nœud $N \in \mathit{dom}_S$, $a_N = \mathit{head}(\mathit{reno}_S(N))$ et si $N_1, \ldots, N_n$ sont les fils de N et C'_N le store de la clause $\mathit{reno}_S(N)$ alors $C_N \vdash \{\exists_{-a}(C'_N \wedge \bigwedge_{i \in \{1,\ldots,n\}} \exists_{-a_{N_i}} C_{N_i})\}$.

On constate tout de suite que si l'on fixe une fonction de renommage reno_S pour S et si l'on associe au nœud N l'atome contraint $a_N \leftarrow \mathrm{AC}(S_N, a_N)$, où S_N est la réponse enracinée en N dans S et $a_N = \mathit{head}(\mathit{reno}_S(N))$, alors la propriété précédente est vérifiée (lemme 2.4.3 et **REFL**).

On peut remarquer un autre cas particulier : si l'atome contraint associé au nœud $N \in \mathit{dom}_S$ est $a_N \leftarrow \exists_{-a_N} \mathit{const}(S, \mathit{reno}_S)$. À nouveau la condition est vérifiée : pour tout $N \in \mathit{dom}_S$:
$\exists_{-a_N} \mathit{const}(S, \mathit{reno}_S) = \exists_{-a_N}(C'_N \wedge \bigwedge_{i \in \{1,\ldots,n\}} \exists_{-a_{N_i}} \mathit{const}(S, \mathit{reno}_S))$ (section 1.3), où C'_N est le store de la clause $\mathit{reno}_S(N)$ et les N_i sont les fils de N dans S.

La relation de parenté de S est bien fondée (S est une réponse). Un nœud $N \in \mathit{dom}_S$ est attendu si et seulement si $a_N \leftarrow C_N$ est attendu.

Supposons que $a_\varepsilon \leftarrow C_\varepsilon$ soit un symptôme positif.

Du second schéma de diagnostic et de la relation de parenté de S on déduit la notion de (nœud) symptôme et la notion de (nœud) symptôme minimal.

Il est évident que si $N \in dom_S$ est un symptôme minimal et N a n fils $N_1, \ldots, N_n$ dans S alors $\langle reno_S(N), C_{N_1}, \ldots, C_{N_n} \rangle$ est une incorrection positive.

Ce cadre plus général est intéressant. Par exemple, supposons que l'oracle (l'utilisateur) soit en mesure de fournir des stores qui renforcent le store réponse $\mathrm{AC}(S, a_\varepsilon)$ de manière que $a_\varepsilon \leftarrow \mathrm{AC}(S, a_\varepsilon)$ soit toujours un symptôme alors on obtient une incorrection positive "plus précise" On remarque que la clause de l'incorrection positive trouvée n'est pas forcement la même selon le choix des atomes contraints associés aux nœuds de la réponse si celle-ci fait intervenir plusieurs clauses incorrectes.

Nous avons montré que l'algorithme de la section 4.4.1 est un cas particulier.

On retrouve aussi l'algorithme de [58] si les stores des l'atomes contraints associés aux nœuds sont construit à partir du store associé à S et une fonction de renommage pour S.

Supposons que l'on dispose d'une $\mathcal{D}$-interprétation attendue I pour le programme et que la relation de couverture soit correcte pour $\mathcal{D}$ (en particulier le critère de rejet est correct pour $\mathcal{D}$).

Le programme est sensé axiomatiser la $\mathcal{D}$-interprétation. S'il existe des symptômes positifs c'est que ce n'est pas le cas.

On retrouve alors le cadre de la section 2 de [58]. Mais ici la présentation est différente (on se passe de la notion de "store témoin") parce qu'on a choisi pour la sémantique déclarative le cadre général de la section 3.5 plutôt que le cadre particulier de la section 3.2. $a \leftarrow C$ est un symptôme s'il existe une valuation v telle que $v_{\mathcal{D}}(C) = \mathit{true}$ et $v_I(a) = \mathit{false}$. $\langle a \leftarrow C \,\square\, a_1 \cdots a_n, C_1, \ldots, C_n \rangle$ est une incorrection positive si et seulement si il existe une valuation v telle que $v_I(a_i \leftarrow C_i) = \mathit{true}$ pour tout $i = 1, \ldots, n$ mais $v_I(a \leftarrow \exists_{-a}(C \wedge \bigwedge_{i \in \{1,ldots,n\}} C_i)) = \mathit{false}$.

Quand l'oracle est l'utilisateur, c'est généralement en fonction d'une $\mathcal{D}$-interprétation attendue I qu'il répond aux questions posées par l'algorithme de diagnostic. Cela est possible car les critères de rejet des systèmes sont toujours corrects pour la pré-interprétation $\mathcal{D}$ sous-jacente.

4.5 Correction partielle négative : réponses manquantes

On suppose un programme P et un critère de rejet RC fixés.

Le cas des symptômes dûs à des réponses manquantes est propre à l'indéterminisme des langages de programmation relationnelle (par exemple la programmation logique avec contraintes).

Supposons que pour le but $\leftarrow a$ on obtient le $\vee$-store réponse R. La paire (a, R) est symptôme d'une erreur dans le programme si R n'est pas un $\vee$-store réponse attendu pour $\leftarrow a$. On peut distinguer deux cas :

1. il existe au moins un store en trop dans R ;
2. il manque au moins un store dans R.

Le premier cas se ramène à la recherche d'une incorrection positive. Le store qui ne devrait pas être dans R vient d'une réponse S, et c'est dans la dérivation SLD qui calcule S que se trouve le problème.

Par contre le second cas ne se ramène pas à l'étude d'une dérivation SLD. C'est pour l'arbre SLD que quelque chose ne va pas : il lui manque au moins une branche.

Dans cette section on s'intéresse au diagnostic d'erreur dans le programme quand le $\vee$-store réponse pour un but n'est pas attendu. D'après les remarques précédentes, on suppose qu'il n'est pas

attendu parce qu'il est incomplet. Dans le cas contraire, on invoquera les techniques de diagnostic d'erreur de la section 4.4.

Différentes approches peuvent être envisagée pour le diagnostic déclaratif dans le cas de réponses manquantes.

Nous avons fait le choix de ne détailler que celle qui semble la plus intéressante : c'est le pendant pour la programmation logique avec contraintes de la recherche d'insuffisance faible en programmation logique pure. En fin de section et dans la section 4.6, nous discutons des alternatives possibles.

Pour les insuffisances faibles, l'algorithme de [37] suppose l'existence d'un arbre SLD fini standard. De plus, lors de la recherche de l'insuffisance faible, l'algorithme suit de très près la stratégie standard (pour l'ordre des questions). Aussi, il n'est pas capable de trouver toutes les insuffisances faibles contenues dans l'arbre SLD, et on ne peut mettre en œuvre des optimisations du type diviser pour régner. [37] ne peut donner à l'avance l'ensemble des questions susceptibles d'être posées à l'oracle. C'est pourquoi, leur algorithme est l'unique preuve que s'il existe un symptôme alors il existe une erreur. Dans [92], nous avons, dans le cadre de la programmation logique avec contraintes, défini un système de règles qui permet de prouver, indépendamment de l'algorithme, que l'existence d'un symptôme implique celle d'une erreur (de plus, nous avons défini les symptômes d'incomplétude, les symptômes d'insuffisance, les insuffisances faibles et les insuffisance à partir de cet unique système de règles, ce qui facilite leur comparaison et la compréhension des liens entre ces notions). Malheureusement, l'algorithme proposé, largement inspiré de celui de [37] souffre des mêmes défauts, bien que la condition d'existence d'un arbre SLD standard ait été généralisée à la condition d'existence d'un arbre SLD sans co-routinage. Dans la suite, nous expliquons enfin complètement ces algorithmes qui ne sont en fait que des instances de la famille d'algorithmes plus générale que nous proposons.

4.5.1 Compléments de sémantique opérationnelle

On commence par des compléments de sémantique opérationnelle. Ces compléments ne sont utiles que pour cette section. C'est pourquoi ils n'apparaissent qu'ici, bien qu'ils soient directement reliés au chapitre 2.

On définit la notion de store réponse pour une paire $C \square A$, où C est un store et A est une suite finie d'atomes.

Définition 4.5.1 Store réponse pour $C \square A$
Un store réponse *pour* $C \square a_1 \cdots a_n$ *est* $C \wedge C_1 \wedge \cdots \wedge C_n$, *où* $C_1, \ldots, C_n$ *sont* n *stores tels que, pour tout* $i = 1, \ldots, n$, C_i *est un store réponse pour* $\leftarrow a_i$ *et* $(C \wedge C_1 \wedge \cdots \wedge C_n) \notin \mathrm{RC}$.
On note $R(C \square A)$ *l'ensemble des stores réponses pour* $C \square A$.

En particulier, on peut remarquer que si $C \notin \mathrm{RC}$ alors $R(C \square \varepsilon) = \{C\}$. On remarque aussi que si $C \in \mathrm{RC}$ alors pour toute suite finie d'atomes A, on a $R(C \square A) = \emptyset$. Enfin, on constate que $R(\emptyset \square a) = \mathrm{SS}(P, a)$.

Ceci nous permet d'énoncer le lemme de compositionalité suivant, plus général que ceux déjà énoncés dans le chapitre 2 :

Lemme 4.5.2 *Pour tout store* C, *pour toutes suites finies d'atomes* A_1 *et* A_2, *pour tout atome* a :

$$R(C \square A_1 \cdot a \cdot A_2) = \bigcup_{C' \in R(C \square a)} R(C \wedge C' \square A_1 \cdot A_2)$$

Preuve. Soit $\{a_i\}_{i\in I}$ la famille d'atomes de la suite A_1. Soit $\{a_j\}_{j\in J}$ la famille d'atomes de la suite A_2.
Les éléments de $R(C \,\square\, A_1 \cdot a \cdot A_2)$ sont les stores $C \wedge C_1 \wedge C' \wedge C_2$ tels que

- pour tout $i \in I$, C_i est un store réponse pour $\leftarrow a_i$ et $C_1 = \bigwedge_{i\in I} C_i$;
- C' est un store réponse pour $\leftarrow a$;
- pour tout $j \in J$, C_j est un store réponse pour $\leftarrow a_j$ et $C_2 = \bigwedge_{j\in J} C_j$;
- $C \wedge C_1 \wedge C' \wedge C_2 \notin \mathrm{RC}$.

Les éléments de $\bigcup_{C'\in R(C \,\square\, a)} R(C \wedge C' \,\square\, A_1 \cdot A_2)$ sont les stores $C \wedge C' \wedge C_1 \wedge C_2$, où

- C' est un store réponse pour $\leftarrow a$ tel que $C \wedge C' \notin \mathrm{RC}$;
- pour tout $i \in I$, C_i est un store réponse pour $\leftarrow a_i$ et $C_1 = \bigwedge_{i\in I} C_i$,
- pour tout $j \in J$, C_j est un store réponse pour $\leftarrow a_j$ et $C_2 = \bigwedge_{j\in J} C_j$,
- $C \wedge C' \wedge C_1 \wedge C_2 \notin \mathrm{RC}$.

On constate que l'unique différence est que dans le second cas on a l'hypothèse supplémentaire $C \wedge C' \notin \mathrm{RC}$. Or les propriétés du critère de rejet garantissent que, pour tout store C_0, si $C_0 \in \mathrm{RC}$ alors, pour tout store C'_0, $C_0 \wedge C'_0 \in \mathrm{RC}$. Donc les deux ensembles sont égaux. ∎

Nous continuons à donner des compléments sur la sémantique opérationnelle. On suppose maintenant qu'une règle de calcul r est fixée.

Intuitivement, un état incomplet S peut être vu de deux manières différentes selon que l'on s'intéresse à S et $r(S)$, ou à S, $r(S)$ et les frères de $r(S)$. Afin de modéliser ces deux visions, on fait la somme disjointe de l'ensemble des états avec lui-même.

Soit $E_0 = \bigcup_{p\in\Pi_p}\{S \mid sq(p) \hookrightarrow_r^p {}^* S, S \text{ incomplet}\}$. Soit $E_1 = \bigcup_{p\in\Pi_p}\{S \mid sq(p) \hookrightarrow_r^p {}^+ S\}$.
Soit $E_0 \oplus E_1$ la somme disjointe de E_0 et E_1. Cet ensemble est isomorphe à $(\{0\}\times E_0)\cup(\{1\}\times E_1)$ et nous confondons ces deux ensembles, pour faciliter les notations. Par conséquent, on représente les éléments de $E_0 \oplus E_1$ par des paires (b, S), où $b \in \{0, 1\}$, $S \in E_0$ si $b = 0$ et $S \in E_1$ si $b = 1$.
On appelle *b-état* tout élément de $E_0 \oplus E_1$. Un élément représenté par $(0, S)$ est un 0-*état*, et un élément représenté par $(1, S)$ est un 1-*état*.

Lemme 4.5.3 *Si $(1, S)$ est un 1-état alors il existe un unique S_0 tel que $S_0 \hookrightarrow_r S$.*

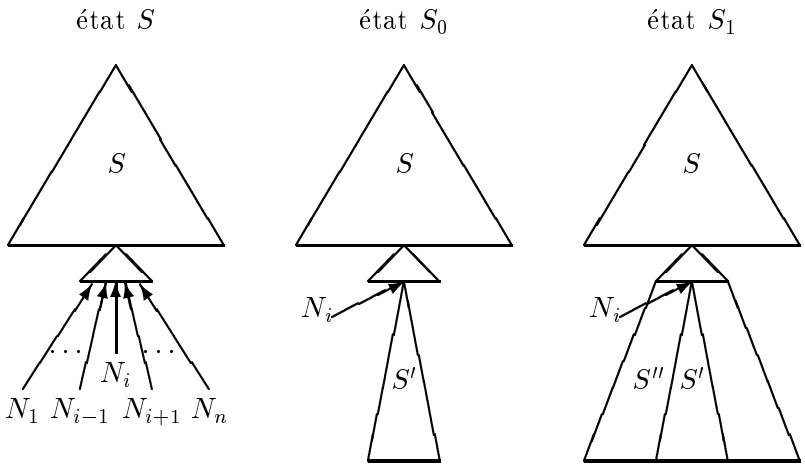
Preuve. Soit $p \in \Pi_p$ tel que S est un état pour p. Toute dérivation SLD qui contient S est une dérivation SLD pour p. D'après la définition de E_1, $S \in dom_r^p$ et S n'est pas la racine de $(dom_r^p, \hookrightarrow_r^p)$, donc S_0 est le père de S dans $(dom_r^p, \hookrightarrow_r^p)$. ∎

Dans la suite, si $(1, S)$ est un b-état alors on appelle *père* de S l'état S_0 tel que $S_0 \hookrightarrow_r S$. Le père de S est noté $father(S)$.

Définition 4.5.4 Prolongement d'un b-état
Soit $(0, S)$ un 0-état. S' est un prolongement *de $(0, S)$ si*

- *il existe un état complet S'' tel que $S' = graft(S, r(S), S'')$,*
- *S' est un état.*

Soit $(1, S)$ un 1-état. S' est un prolongement *de $(1, S)$ si*



$N_i = r(S)$, $N_1, \ldots, N_{i-1}, N_{i+1}, \ldots, N_n$ sont les frères de N_i
S' est un état complet (éventuellement infini)
S'' est un ensemble de $n-1$ états complets (éventuellement infinis)

Figure 4.3 : Prolongement d'un b-état.

- $dom_S \subseteq dom_{S'}$,
- *pour tout* $N \in def(S)$, $lab_S(N) = lab_{S'}(N)$,
- $undef(father(S)) \setminus \{r(father(S))\} = undef(S')$,
- S' *est un état.*

On note $prol_b(S)$ *l'ensemble des prolongements de* (b, S).

Intuitivement, on obtient un prolongement de $(0, S)$ en greffant une réponse en $r(S)$ dans S, et on obtient un prolongement de $(1, S)$ en greffant une réponse en chaque fils de $r(father(S))$ dans S.

On peut remarquer que, pour tout atome a dont le symbole de prédicat est p, $prol_0(sq(p)) = success(a)$.

Lemme 4.5.5 *Soit* $(1, S)$ *un* 1*-état.*

$$prol_1(S) = \{S' \in prol_0(father(S)) \mid lab_{S'}(r(father(S))) = lab_S(r(S_0))\}$$

Preuve. Soit $S_0 = father(S)$.

$\subseteq$ Soit $S' \in prol_1(S)$. D'après la définition $dom_S \subseteq dom_{S'}$. Soit S'' l'état enraciné en $r(S_0)$ dans S'. S'' est un état complet puisque $undef(S_0) \setminus \{r(S_0)\} = undef(S')$. D'une part, pour tout $N \in def(S)$ on a $lab_S(N) = lab_{S'}(N)$, d'autre part, $undef(S_0) \setminus \{r(S_0)\} = undef(S')$, donc $S' = graft(S_0, r(S_0), S'')$. Donc $S' \in prol_0(S_0)$.

$\supseteq$ Soit $S' \in \{S' \in prol_0(S_0) \mid lab_{S'}(r(S_0)) = lab_S(r(S_0))\}$. Il existe un état complet S'' tel que $S' = graft(S_0, r(S_0), S'')$ et $lab_{S''}(\varepsilon) = lab_{S'}(r(S_0)) = lab_S(r(S_0))$. Donc $dom_S \subseteq dom_{S'}$ et pour tout $N \in def(S)$, $lab_S(N) = lab_{S'}(N)$. Comme S'' est complet alors $undef(S_0) \setminus \{r(S_0)\} = undef(S')$. Donc $S' \in prol_1(S)$.

■

Lemme 4.5.6 *Soit S un état incomplet non initial. Si $r(S)$ n'a pas de frère indéfini alors $prol_0(S) = prol_1(S)$.*

Preuve. Rappelons qu'un état initial est de la forme $sq(p)$ où $p \in \Pi_p$ (voir section 2.5.1). La preuve est similaire à celle du lemme 4.5.5 : on montre que tout élément de $prol_0(S)$ vérifie les propriétés de la définition de $prol_1(S)$ et réciproquement. ■

Remarque. Soient $(1, S)$ un 1-état. Si $r(father(S))$ n'a pas de fils dans S alors $prol_1(S) = \{S\}$ (S est de la forme $graft(father(S), r(father(S)), sq(u))$, où u est le nom d'un fait). ◇

Lemme 4.5.7 *Soit $(0, S)$ un 0-état.*

$$prol_0(S) = \bigcup_{S \hookrightarrow_r S'} prol_1(S')$$

En d'autres termes, si p est le symbole de prédicat associé à $r(S)$ dans S et si $X = \{u \in cn(P, p) \mid graft(S, r(S), sq(u))$ est un état$\}$ alors

$$prol_0(S) = \bigcup_{u \in X} prol_1(graft(S, r(S), sq(u)))$$

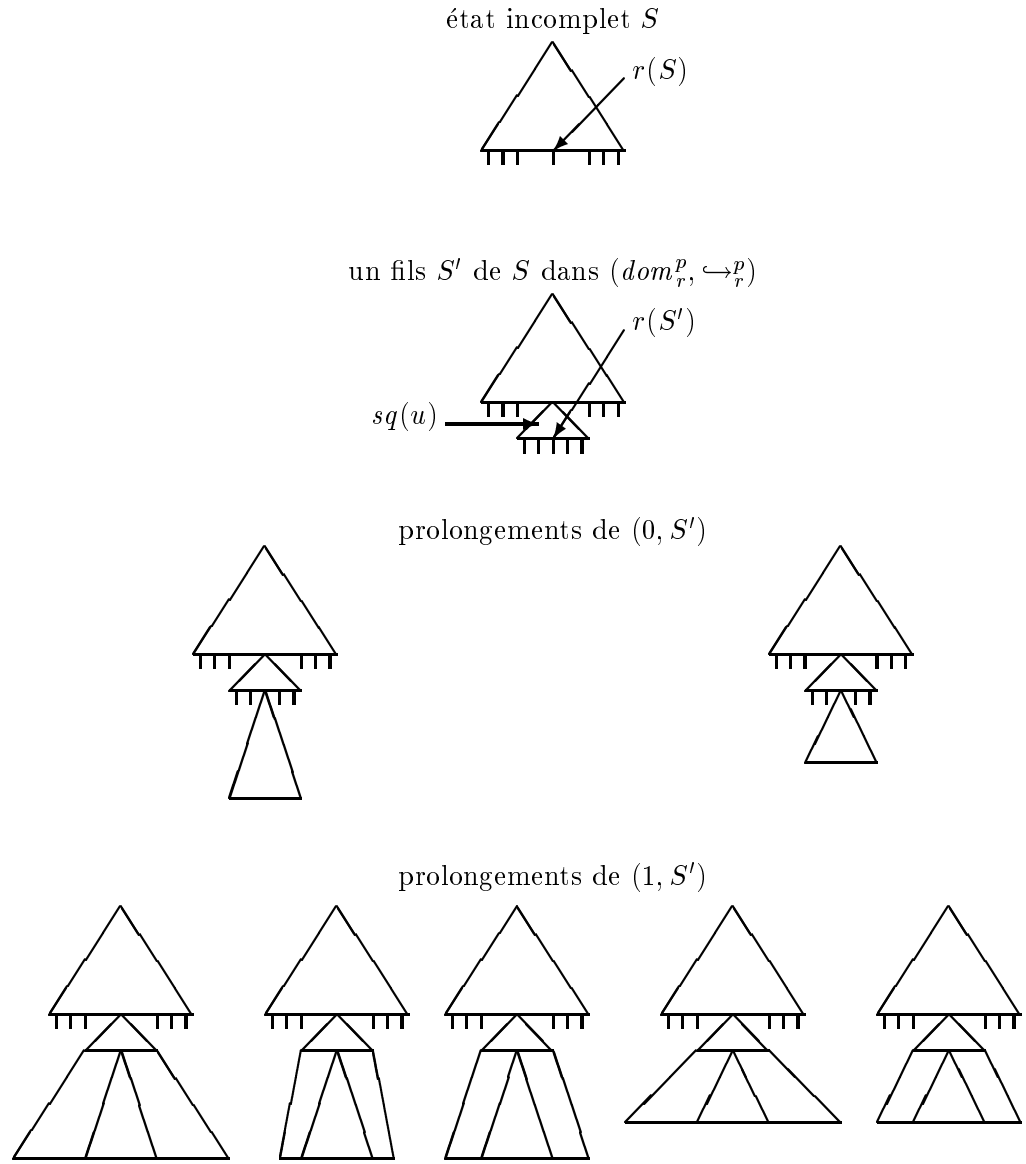
Preuve.

$\subseteq$ Soit $S' \in prol_0(S)$ et $u = lab_{S'}(r(S))$. S' est un état donc $graft(S, r(S), sq(u))$ est un état. Le lemme 4.5.5 montre que $S' \in prol_1(graft(S, r(S), sq(u)))$.

$\supseteq$ Le lemme 4.5.5 montre que $prol_1(graft(S, r(S), sq(u))) = \{S' \in prol_0(S) \mid lab_{S'}(r(S)) = u\}$. Donc $\bigcup_{u \in X} prol_1(graft(S, r(S), sq(u))) \subseteq prol_0(S)$

■

Le lemme précédent montre une forme de compositionalité (des états éventuellement infinis) de la sémantique opérationnelle.



L'ensemble des prolongements de $(0, S)$ est la réunion
des prolongements de $(1, S')$, où S' est un fils de S ($S' \hookrightarrow_r^p S$).

Les prolongements de $(1, S')$ sont les prolongements de $(0, S)$
qui ont le nœud $r(S)$ étiqueté par u.

Figure 4.4 : Compositionalité des prolongements d'un b-état

On suppose maintenant que la règle de calcul r est sans *co-routinage*.

Soit $T = (dom_r^p, \hookrightarrow_r^p)$ un arbre SLD pour p selon la règle de calcul sans co-routinage r.

T (i.e. p et r) est fixé jusqu'à la fin de cette section. Dans la suite, on ne s'intéressera qu'à des b-états de la forme (b, S) avec $S \in dom_r^p$. Aussi, pour simplifier, on suppose maintenant que le terme b-état désigne un élément (b, S) avec $S \in dom_r^p$:

Lemme 4.5.8 *Soit (b, S) un b-état ($S \in dom_r^p$).*

$$prol_b(S) \subseteq dom_r^p$$

De plus, pour tout $S' \in prol_b(S)$,

- *si $b = 1$ et $r(father(S))$ n'a pas de fils dans S alors $S' = S$;*
- *sinon S' est un descendant de S dans $(dom_r^p, \hookrightarrow_r^p)$.*

Preuve. Le lemme 4.5.5 montre que, pour tout 1-état $(1, S)$, $prol_1(S) \subseteq prol_0(father(S))$. Or $father(S) \in dom_r^p$ donc $(0, father(S))$ est un 0-état. Par conséquent, il suffit de montrer que, pour tout 0-état $(0, S)$, $prol_0 S \subseteq dom_r^p$.
Soient $(0, S)$ un 0-état et $S' \in prol_0(S)$. Il existe un état complet S'' tel que $S' = graft(S, r(S), S'')$. Comme $S \in dom_r^p$, il existe un préfixe U d'une dérivation SLD selon r pour p dont le dernier état est S. On montre que l'on peut construire un préfixe U' d'une dérivation SLD selon r pour p dont le dernier état est $graft(S, r(S), S'')$:

1. U est un préfixe de U' ;
2. à partir du dernier état S de U (on remarque qu'une feuille indéfinie de S, différente de $r(S)$, ne peut être sélectionnée avant que l'état greffé en $r(S)$ ne soit complet : la règle est sans co-routinage), on définit la fin U'' de $U' = U \cdot U''$ de la manière suivante :
   (a) le premier état de U'' est $graft(S, r(S), sq(lab_{S''}(\varepsilon)))$;
   (b) si S''' est un état de U'' différent de S' alors, il existe N tel que $r(S''') = r(S) \cdot N$, l'état suivant S''' dans U'' est $graft(S''', r(S'''), sq(lab_{S''}(N)))$.
   (c) le dernier état de U'' est S'.

   U' est bien défini et U' est préfixe d'une branche de $(dom_r^p, \hookrightarrow_r^p)$; son dernier état est S'.

Donc $S' \in dom_r^p$. De plus, c'est un descendant de S dans $(dom_r^p, \hookrightarrow_r^p)$.
Il reste à montrer que, pour tout 1-état $(1, S)$, si $S' \in prol_1(S)$ alors $S' = S$ ou S' est un descendant de S. On sait déjà que S' est un descendant de $father(S)$. On a vu que si $r(father(S))$ n'a pas de fils dans S alors $prol_1(S) = \{S\}$. Il est facile de voir que si $r(father(S))$ a un fils dans S alors $S \notin prol_1(S)$. ∎

Le lemme précédent montre une propriété tout à fait remarquable des arbres SLD sans co-routinage. Cette propriété est fausse en général si l'arbre est avec co-routinage.

De plus, la réciproque du lemme est fausse. C'est-à-dire, si pour tout état incomplet S d'un arbre SLD, $prol_0(S)$ est un sous-ensemble des nœuds de l'arbre alors l'arbre n'est pas obligatoirement sans co-routinage : il peut avoir une branche échec avec co-routinage. En fait, on peut seulement en déduire que ses branches succès sont sans co-routinages.

Corollaire 4.5.9 *Si l'arbre T est fini alors, pour tout b-état (b, S), $prol_b(S)$ est un ensemble fini d'états finis.*

Preuve. $prol_b(S)$ est fini car dom_r^p est fini. Un état de $prol_b(S)$ est fini car tout état d'un arbre SLD fini est fini. ∎

Soit $<_T$ la relation binaire sur l'ensemble des b-états définie par

- pour tout 0-état $(0, S)$, si $S \hookrightarrow_r^p S'$ alors :
$$(1, S') <_T (0, S)$$
- pour tout 1-état $(1, S)$, si $r(father(S))$ à un fils dans S (i.e. $prol_1(S) \neq \{S\}$) alors :
$$(0, S) <_T (1, S)$$
- pour tout 1-état $(1, S)$, si $r(father(S))$ à un fils dans S alors pour tout $S' \in prol_0(S)$:
$$(1, S') <_T (1, S)$$

Cette relation est bien définie d'après le lemme 4.5.8.

Lemme 4.5.10 *Si l'arbre T est fini alors la relation $<_T$ est bien fondée.*

Preuve. On constate que si $(b'', S'') <_T (b', S') <_T (b, S)$ alors S'' est un descendant de S dans T. Comme T est fini, $<_T$ est une relation bien fondée. ∎

À tout 0-état $(0, S)$ est associée la paire $C \square a$, où C et a sont tels qu'il existe une fonction de renommage $reno_S$ pour S, $C = \exists_{-a} const(S, reno_S)$ et a est l'atome associé à $r(S)$ par $reno_S$.

De la même manière, à tout 1-état $(1, S)$ est associée la paire $C \square a_1 \cdots a_n$, où C et les a_i sont tels qu'il existe une fonction de renommage $reno_S$ pour S, $C = \exists_{-var(a_1 \cdots a_n)} const(S, reno_S)$, $r(father(S))$ à n fils dans S, les a_i sont les atomes associés aux $r(father(S)) \cdot (i-1)$ par $reno_S$ dans S.

La paire $C \square A$ associée à un b-état est définie à un renommage près.

Notons que si S est un état incomplet non initial tel que $r(S)$ n'a pas de frère alors les paires associées à $(0, S)$ et $(1, S)$ sont les mêmes. Ce n'est pas sans rapport avec le lemme 4.5.6 et le lemme suivant :

Lemme 4.5.11 *Soit $(0, S)$ un 0-état. Soit $C \square a$ la paire associé à $(0, S)$.*

$$\begin{aligned} R(C \square a) = \{ \quad & \exists_{-a} const(S', reno_{S'}) \ \mid \\ & S' \in prol_0(S), \\ & S' \text{ est fini} \\ & reno_{S'} \text{ est une fonction de renommage pour } S', \\ & a \text{ est l'atome associé à } r(S) \text{ par } reno_{S'} \text{ dans } S' \} \end{aligned}$$

Soit $(1, S)$ un 1-état. Soit $C \square a_1 \cdots a_n$ la paire associé à $(1, S)$. Soit $N_1, \ldots, N_n$ les fils de $r(father(S))$ dans S.

$$\begin{aligned} R(C \square a_1 \cdots a_n) = \{ \quad & \exists_{-var(A)} const(S', reno_{S'}) \ \mid \\ & S' \in prol_1(S), \\ & S' \text{ est fini} \\ & reno_{S'} \text{ est une fonction de renommage pour } S', \\ & a_i \text{ est l'atome associé à } N_i \text{ par } reno_{S'} \text{ dans } S', i = 1 \ldots, n \} \end{aligned}$$

Preuve. Le corollaire 4.5.9 assure que les états de $prol_b(S)$ sont finis. Il suffit de suivre les définitions en remarquant qu'un état complet fini est une réponse. ∎

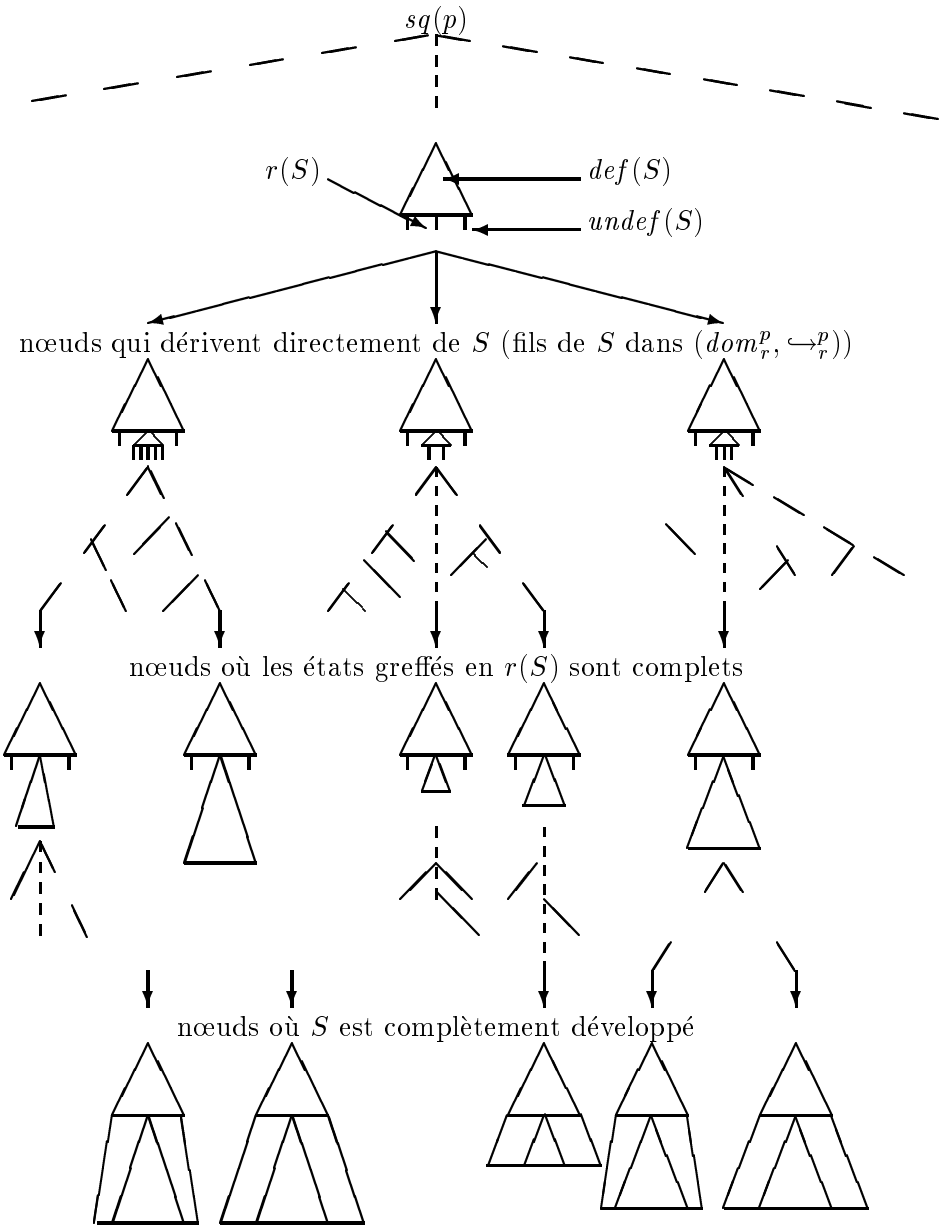


Figure 4.5 : Relation $<_T$ sur les nœuds de l'arbre SLD T

4.5.2 Diagnostic d'incorrection négative

Les préliminaires sur la sémantique opérationnelle étant donnés, nous traitons le problème du diagnostic d'erreur pour le cas des réponses manquantes.

On suppose fixé un arbre SLD fini $T = (dom_r^p, \hookrightarrow_r^p)$ sans co-routinage pour p selon la règle de calcul r jusqu'à la fin de cette section.

Supposons que l'on dispose d'un moyen de savoir si un b-état est attendu en fonction des propriétés attendues du programme.

Le schéma général de la section 4.3 fournit immédiatement la notion de symptôme et symptôme minimal de $<_T$ par rapport à la notion de b-état attendu.

La relation "b-état attendu" doit vérifier des propriétés similaires aux propriétés vérifiées par les "b-états calculés".

Cette propriété est une propriété de fermeture de la relation "b-état attendu" par les deux derniers points de la définition de $<_T$:

Exigences sur la relation "b-état attendu"

On exige que pour tout 1-état $(1, S)$, si tout prédécesseur de $(1, S)$ par $<_T$ est attendu alors $(1, S)$ est attendu.

On déduit de l'exigence de fermeture ci-dessus que les symptômes minimaux de $<_T$ sont des 0-états. Par conséquent, la notion de 0-état minimal est entièrement déterminée par les 0-états symptômes. On pourrait penser que la notion de 1-état est inutile. En fait, la distinction entre 0-état et 1-état facilite la compréhension de la méthode et permettra par la suite de définir une notion d'erreur associée à un symptôme minimal simple et intuitive.

Des arguments similaires à ceux développés pour le diagnostic de réponses fausses nous amène à ne pas exiger que les propriétés attendues du programme soient formalisées par un ensemble de b-états attendus. En fait, il suffit que les propriétés attendues de P permettent de déterminer si un b-état est attendu.

Un b-état (b, S) est attendu si sa paire associée est $C \square A$ et $C \square A \rightarrow R(C \square A)$ est attendue. $C \square A \rightarrow R(C \square A)$ est la couverture locale d'atomes associée au b-état (b, S). La question est de déterminer si l'ensemble des stores réponses pour $C \square A$ convient. On peut noter que la propriété $C \square A \rightarrow R(C \square A)$ attendu ne dépend que de $C \square A$. Les propriétés attendues sont formalisées par un ensemble de couvertures locales d'atomes attendues.

On suppose, bien-entendu, que la relation "couverture locale d'atomes attendue" vérifie certaines propriétés :

Conditions sur la relation "Couverture locale d'atomes attendue"

Pour tout store C, la couverture locale d'atomes $C \square \varepsilon \rightarrow \{C\}$ est attendue (on remarque $R(C \square \varepsilon)$ ne dépend pas du programme).

Pour tout store C, pour tout ensemble de stores R, pour toute famille d'ensembles de stores $\{R_{C'}\}_{C' \in R}$, pour toutes suites finies d'atomes A_1 et A_2, pour tout atome a, si $C \square a \rightarrow R$ est attendu et, pour tout $C' \in R$, $C \wedge C' \square A_1 \cdot A_2 \rightarrow R_{C'}$ est attendu alors $C \square A_1 \cdot a \cdot A_2 \rightarrow \bigcup_{C' \in R} R'_C$ est attendu.

Il faut montrer que sous la condition précédente, la relation "b-état attendu" sous-jacente vérifie l'exigence de fermeture par les deux derniers points de la définition de $<_T$.

Lemme 4.5.12 *Pour tout* 1*-état* $(1, S)$*, si les prédécesseurs de* $(1, S)$ *par* $<_T$ *sont attendus alors* $(1, S)$ *est attendu.*

Preuve.

- Si $r(father(S))$ n'a pas de fils dans S alors $(1, S)$ ne peut pas être un symptôme puisque pour tout store C, $C \square \varepsilon \rightarrow C$ est attendu.
- Si $r(father(S))$ a un fils dans S, sa paire associée est de la forme $C \square A_1 \cdot a \cdot A_2$ (ou a correspond à $r(S)$). Les prédécesseurs de $(1, S)$ sont attendus, donc $C \square a \rightarrow R(C \square a)$ est attendu et, pour tout $C' \in R(C \square a)$, $R(C \wedge C' \square A_1 \cdot A_2)$ est attendu. D'après la condition sur les couvertures locales d'atomes attendues, $C \square A_1 \cdot a \cdot A_2 \rightarrow \bigcup_{C' \in R(C \square a)} R(C \wedge C' \square A_1 \cdot A_2)$ est attendue. Or $R(C \square A_1 \cdot a \cdot A_2) = \bigcup_{C' \in R(C \square a)} R(C \wedge C' \square A_1 \cdot A_2)$, donc $C \square A_1 \cdot a \cdot A_2 \rightarrow R(C \square A_1 \cdot a \cdot A_2)$ est attendue. D'où $(1, S)$ n'est pas un symptôme.

■

Corollaire 4.5.13 *Si* (b, S) *est un symptôme minimal de* $<_T$ *alors* $b = 0$.

Les symptômes minimaux sont donc des 0-états.

Définition 4.5.14 Symptôme d'incorrection partielle négatif
Une paire $C \square a$ *est un* symptôme négatif *si* $C \square a \rightarrow R(C \square a)$ *n'est pas attendu.*
De plus, s'il existe un arbre SLD fini T *sans co-routinage tel que* $C \square a \rightarrow R(C \square a)$ *est la couverture locale d'atomes associée à un* 0*-état alors* $C \square a$ *est un symptôme négatif calculé.*

Il est très intéressant que les propriétés attendues soient les plus simples possibles. Ici, comme la seule notion utile est celle de 0-état attendu, les questions à l'oracle sont plus simples :

- d'une part, elles sont atomiques car la paire associée à un 0-état ne contient qu'un seul atome ;
- d'autre part, si A contient plusieurs atomes, la question pour $C \square A$ contient éventuellement plus de variables que si l'on se ramène à une question pour chaque atome de la suite A.

À un symptôme minimal $(0, S)$, on peut associer une définition de prédicat erronée : c'est la définition du prédicat p' associé à $r(S)$. Mais le cadre trop général ne permet pas d'exprimer une notion d'erreur simple et intuitive qui explique la raison pour laquelle la définition de p' est responsable du symptôme.

De plus, nous ne voulons pas définir deux types de propriétés attendues pour un programme : des atomes contraints pour le problème des réponses fausses et des couvertures locales d'atomes pour le problème des réponses manquantes.

Nous allons utiliser uniquement la relation "atome contraint attendu" de la section 4.4. Pour cela on fait appel à une relation de couverture.

4.5.3 Diagnostic d'incorrection négatif selon une relation de couverture $\vdash$

Soit $\vdash$ une relation de couverture. Le critère de rejet est définit par la relation unaire $\vdash \emptyset$, i.e. $C \in \mathrm{RC}$ si et seulement si $C \vdash \emptyset$.

Les propriétés attendues du programme sont formalisées par l'ensemble I d'atomes contraints attendus. Rappelons que I est fermé par la propriété : si $a \leftarrow C \in I$ alors pour tout store C' tel que $C' \vdash \{C\}$, $a \leftarrow C' \in I$

On précise maintenant comment on détermine si une couverture locale d'atomes est attendue en fonction de l'ensemble d'atomes contraints attendus I.

Une couverture locale d'atomes $C \square A \rightarrow R(C \square A)$ est attendue si pour tout C' tel que $C' \vdash \{C\}$ et $a \leftarrow C' \in I$, $a \in A$, on a $C' \vdash R(C \square A)$.

On rappelle qu'un b-état (b, S) est attendu si sa couverture locale d'atomes associée est attendue.

On remarque en particulier que $(0, sq(p))$ est attendu si, pour tout store C' tel que $a \leftarrow C' \in I$, on a $C' \vdash \mathrm{SS}(P, a)$, où $p \in \Pi_p$ et a est un atome dont le symbole de prédicat est p.

Dans la suite, on note $A \leftarrow C \in I$ si pour tout atome a de la suite finie A, l'atome contraint $a \leftarrow C$ appartient à I.

Montrons que la relation "b-état attendu" sous-jacente vérifie toujours les exigences précédentes. Il suffit de vérifier qu'elle vérifie les conditions de la relation "couverture locale d'atomes attendue"

Lemme 4.5.15 *Pour tout store C, pour toutes suites finies d'atomes A_1 et A_2, pour tout atome a,*

1. *$C \square \varepsilon \rightarrow \{C\}$ est attendu.*
2. *Si $C \square a \rightarrow R(C \square a)$ est attendu et, pour tout $C' \in R(C \square a)$, $C \wedge C' \square A_1 \cdot A_2 \rightarrow R(C \wedge C' \square A_1 \cdot A_2)$ est attendu alors $C \square A_1 \cdot a \cdot A_2 \rightarrow R(C \square A_1 \cdot a \cdot A_2)$ est attendu.*

Preuve.

1. Il suffit d'appliquer les définitions.
2. On suppose que $C \square a \rightarrow R(C \square a)$ est attendu et, pour tout $C' \in R(C \square a)$, $C \wedge C' \square A_1 \cdot A_2 \rightarrow R(C \wedge C' \square A_1 \cdot A_2)$ est attendu.
   Soit C_0 un store tel que $C_0 \vdash \{C\}$ et $A_1 \cdot a \cdot A_2 \leftarrow C_0 \in I$. Montrons que $C_0 \vdash R(C \square A_1 \cdot a \cdot A_2)$.
   Pour tout $C_1 \in R(C \square a)$ on a $C_0 \wedge C_1 \vdash \{C \wedge C_1\}$ (car $C_0 \vdash \{C\}$ et **CONJ**) et $A_1 \cdot A_2 \leftarrow C_0 \wedge C_1 \in I$ (car $A_1 \cdot A_2 \leftarrow C_0 \in I$).
   $C_0 \vdash \{C\}$ et $C_0 \vdash R(C \square a)$ donc $C_0 \vdash \{C \wedge C_1 \mid C_1 \in R(C \square a)\}$ (**CONJ** et $C_0 \wedge C_0 = C_0$ d'après les équivalences entre stores de la section 1.3).
   On a $C_0 \vdash \{C \wedge C_1\}_{C_1 \in R(C \square a)}$ et, pour tout $C_1 \in R(C \square a)$, $C_0 \wedge C_1 \vdash R(C \wedge C_1 \square A_1 \cdot A_2)$ donc $C_0 \vdash \bigcup_{C_1 \in R(C \square a)} R(C \wedge C_1 \square A_1 \cdot A_2)$ (**TRAN**).
   Donc $C_0 \vdash R(C \square A_1 \cdot a \cdot A_2)$ (lemme 4.5.2).

■

Corollaire 4.5.16 *Pour tout b-état (b, S), si (b, S) est un symptôme alors $b = 0$.*

La relation de couverture permet d'associer une notion d'erreur à tout symptôme minimal.

Définition 4.5.17 Couvert, Complètement couvert
Soit C un store et a un atome.
L'atome contraint $a \leftarrow C$ est couvert *par P relativement à I si $a \leftarrow C \in T^{\circ}_{\vdash,P}(I)$.*
La paire $C \square a$ est complètement couverte *par P relativement à I si, pour toute store C' telle que $C' \vdash \{C\}$ et $a \leftarrow C' \in I$, on a $a \leftarrow C'$ est couvert par P relativement à I.*

On remarque que $a \leftarrow C$ *couvert* par P par rapport à I signifie : il existe un ensemble de stores R tel que $C \vdash R$, pour tout $C' \in R$, il existe une clause renommée $a \leftarrow C^{C'} \square a_1^{C'} \cdots a_{n_{C'}}^{C'}$, il existe $n_{C'}$ stores $C_1^{C'}, \ldots, C_{n_{C'}}^{C'}$ tels que, pour tout $i = 1, \ldots, n_{C'}$, $a_i^{C'} \leftarrow C_i^{C'} \in I$ et $C = \exists_{-a}(C^{C'} \wedge \bigwedge_{i=1,\ldots,n_{C'}} C_i^{C'})$.

Définition 4.5.18 Incorrection partielle négative
Une incorrection négative *est une paire $C \square a$ non complètement couverte par P relativement à I.*

Lemme 4.5.19 *Si le 0-état $(0, S)$ est minimal alors sa paire associée $C \square a$ est une incorrection négative.*

Preuve. Soit $(0, S)$ un symptôme minimal. On note $C \square a$ la paire associée à $(0, S)$.
$(0, S)$ est un symptôme donc $C \square a \rightarrow R(C \square a)$ n'est pas attendu. Ce qui signifie qu'il existe un store C_1 tel que $C_1 \vdash \{C\}$ et $a \leftarrow C_1 \in I$, mais $C_1 \nvdash R(C \square a)$.

Supposons que $a \leftarrow$ soit couvert par P par rapport à I. C'est-à-dire, il existe un ensemble de stores R tel que

- $C_1 \vdash R$;
- pour tout $r \in R$, il existe un nom de clause u_r tel que $a \leftarrow C^{u_r} \square a_1^{u_r} \cdots a_{n_{u_r}}^{u_r}$ (on note A^{u_r} la suite d'atomes $a_1^{u_r} \cdots a_{n_{u_r}}^{u_r}$) est un renommage de $clause(u)$ et il existe n_{u_r} stores $C_1^{u_r}, \ldots, C_{n_{u_r}}^{u_r}$ tels que
  - $r = \exists_{-a}(C^{u_r} \wedge \bigwedge_{i \in \{1,\ldots,n_{u_r}\}} C_i^{u_r})$,
  - pour tout $i = 1, \ldots, n_{u_r}$, $a_i^{u_r} \leftarrow C_i^{u_r} \in I$.

Lemme intermédiaire. Pour chaque $r \in R$ on a $C \wedge C^{u_r} \square A^{u_r} \rightarrow R(C \wedge C^{u_r} \square A^{u_r})$ attendu.

Preuve.

1. Si $C \wedge C^{u_r} \nvdash \emptyset$ alors $C \wedge C^{u_r} \square A^{u_r}$ est la paire associée d'un 1-état $(1, S^{u_r})$ tel que $(1, S^{u_r}) <_T (0, S)$ (S est le père de S^{u_r}) car $var(C) \subseteq var(a)$ d'après la définition de paire associée à un 0-état. $(1, S^{u_r})$ est attendu (car $(0, S)$ est minimal) donc $C \wedge C^{u_r} \square A^{u_r} \rightarrow R(C \wedge C^{u_r} \square A^{u_r})$ est attendu.
2. Si $C \wedge C^{u_r} \vdash \emptyset$ alors soit C' tel que $C' \vdash C \wedge C^{u_r}$ (i.e. $C' \vdash \emptyset$) et $A^{u_r} \leftarrow C' \in I$. On a $R(C \wedge C^{u_r} \square A^{u_r}) = \emptyset$ (car $C \wedge C^{u_r} \vdash \emptyset$) donc $C' \vdash R(C \wedge C^{u_r} \square A^{u_r})$. D'où $C \wedge C^{u_r} \square A^{u_r} \rightarrow R(C \wedge C^{u_r} \square A^{u_r})$ est attendu.

■

Pour conclure, on veut une contradiction, à savoir $C_1 \vdash R(C \square a)$.
Il suffit, pour chaque $r \in R$, que $C_1 \wedge r \vdash R(C \square a)$ (car $C_1 \vdash \{C_1\}$ (**REFL**), $C_1 \vdash R$ donc $C_1 \wedge C_1 \vdash \{C_1 \wedge r\}_{r \in R}$ (**CONJ**) ; or $C_1 \wedge C_1 = C_1$ (section 1.3) ; de $C_1 \vdash \{C_1 \wedge r\}_{r \in R}$ et, pour tout $r \in R$, $C_1 \wedge r \vdash R(C \square a)$ on déduit $C_1 \vdash R(C \square a)$ (**TRAN**)).
Or $C_1 \vdash \{C\}$ et $r \vdash \{r\}$ donc $C_1 \wedge r \vdash \{C \wedge r\}$ (**CONJ**).
Donc il suffit que $r \wedge C \vdash R(C \square a)$. (on remarque que $var(r) \cup var(C) \subseteq var(a)$.)
Or, on a $C \wedge C^{u_r} \wedge \bigwedge_{i \in \{1,\ldots,n_{u_r}\}} C_i^{u_r} \vdash \{C \wedge C^{u_r}\}$ (**CONJ**$_g$) et $A^{u_r} \leftarrow C \wedge C^{u_r} \wedge \bigwedge_{i \in \{1,\ldots,n_{u_r}\}} C_i^{u_r} \in I$ (car chaque $a_i^{u_r} \leftarrow C_i^{u_r} \in I$ et $C \wedge C^{u_r} \wedge \bigwedge_{i \in \{1,\ldots,n_{u_r}\}} C_i^{u_r} \vdash \{C_i^{u_r}\}$ (**CONJ**$_g$) donc $a_i^{u_r} \leftarrow C \wedge C^{u_r} \wedge \bigwedge_{i \in \{1,\ldots,n_{u_r}\}} C_i^{u_r} \in I$).

Donc il suffit que $C \wedge C^{u_r} \wedge \bigwedge_{i\in\{1,\ldots,n_{u_r}\}} C_i^{u_r} \vdash R(C \wedge C^{u_r} \square A^{u_r})$.
De plus, $r = \exists_a(C \wedge C^{u_r} \wedge \bigwedge_{i\in\{1,\ldots,n_{u_r}\}} C_i^{u_r})$ donc il suffit que $r \wedge C \vdash R(C \wedge C^{u_r} \square A^{u_r})$ (**EXIS**$_g$ et équivalence entre stores de la section 1.3 ($var(C) \subseteq var(a)$)).
On vérifie enfin, pour tout $r \in R$, que $R(C \wedge C^{u_r} \square A^{u_r}) \subseteq R(C \square a)$:

- Si $C \wedge C^{u_r} \not\vdash \emptyset$ alors du lemme 4.5.7 on déduit $R(C \wedge C^{u_r} \square A^{u_r}) \subseteq R(C \square a)$;
- Si $C \wedge C^{u_r} \vdash \emptyset$ alors $R(C \wedge C^{u_r} \square A^{u_r}) = \emptyset$ donc $R(C \wedge C^{u_r} \square A^{u_r}) \subseteq R(C \square a)$.

Par conséquent, $a \leftarrow C_1$ couvert par P par rapport à I implique $C_1 \vdash R(C \square a)$.
Donc $a \leftarrow C_1$ n'est pas couvert par P par rapport à I ; d'où $C \square a$ est non complètement couvert par P par rapport à I. ■

Corollaire 4.5.20 *S'il existe un symptôme négatif calculé alors il existe une incorrection négative. L'absence d'incorrection négative implique l'absence de symptôme négatif calculé.*

Remarque. Dans [92], nous obtenons un résultat équivalent et de plus nous montrons le résultat plus général : s'il existe un symptôme d'incomplétude (symptôme négatif) alors il existe une insuffisance faible (incorrection négative).
Cet article montre les résultats en utilisant les relations (voir section 3.5) entre les opérateurs T_P, $T_\vdash$, $T^\circ_{\vdash,P}$ et $T^\cup_{\vdash,P}$. Il établit les liens entre les symptômes d'incomplétude, les symptômes d'insuffisance, les insuffisances faibles et les insuffisances (fortes). Malheureusement, le cadre de cet article ne permet pas d'expliquer la famille d'algorithmes de recherche d'insuffisance faible dont nous donnons des instances dans la section suivante. En fait, ces algorithmes ne trouve leur explication que lorsqu'on constate qu'ils sont de simples algorithmes de recherche d'incorrection (mais sur un système de règles fort compliqué qui n'a pas été donné ici). Ceci est parfaitement montré ici grâce à la relation bien fondée entre les b-états.
Le nouveau schéma de diagnostic basé sur une relation bien fondée montre ici tout son intérêt. Il est probable qu'il existe bien d'autres exemples, éventuellement pour d'autres types de langages de programmation (impératifs, fonctionnels, etc.), où ce schéma éclaircirait des algorithmes de diagnostic déclaratifs, ou permettrait d'en découvrir des nouveaux.
Il est intéressant de noter qu'un algorithme de diagnostic recherche avant tout un symptôme minimal. C'est ensuite, que l'on associe à ce symptôme minimal une notion d'erreur. Ceci est nettement mieux expliqué par une relation bien fondée que par un système de règles où la distinction entre le symptôme minimal et l'erreur qui lui est associé n'est pas mise en avant. ◇

L'intérêt de la relation de couverture est double :

1. On se ramène à des propriétés attendues exprimées en termes d'atomes contraints attendus. Ainsi, les propriétés attendues requises pour le diagnostic d'incorrection positive et le diagnostic d'incorrection négative sont les mêmes. C'est agréable pour automatiser une partie du diagnostic en utilisant par exemple une spécification partielle de la sémantique attendue.

2. On peut exprimer une "bonne" notion d'erreur associée à la notion de symptôme minimal. Elle est bonne parce qu'elle explique de manière suffisamment fine le symptôme minimal, mais aussi parce qu'elle est facilement compréhensible par l'utilisateur.

4.5.4 Algorithmes

Dans la section 4.3 nous avons décrit une famille d'algorithmes qui consistent à construire une suite décroissante de symptômes jusqu'à trouver un symptôme minimal. L'algorithme que nous proposons dans [92] est une instance de cette famille. Cet algorithme généralise aux arbres SLD sans co-routinage en programmation logique avec contraintes l'algorithme de [37] donné dans le cadre de la programmation logique pure et qui suppose l'existence d'un arbre SLD standard.

Nous donnons enfin l'explication de ce style d'algorithme grâce à la relation bien fondée $<_T$.

Étant donné un 0-état symptôme $(0, S)$, on interroge les 0-états de l'ensemble $\{(0, S') \mid$ il existe une suite finie de 1-état $(1, S_1) \cdots (1, S_m), (0, S') <_T (1, S_m) <_T \cdots <_T (1, S_1) <_T (0, S)\}$. Cet ensemble correspond à la forêt de preuve définie dans [92] et à la forêt de preuve définie dans [37] si T est un arbre SLD standard. Si un des 0-états de l'ensemble est un symptôme alors l'algorithme est appelé récursivement, sinon la paire associée à $(S, 0)$ est une incorrection négative (ou une insuffisance faible pour [92] ou un atome non complètement couvert pour [37]).

Comme l'algorithme est invoqué si $(0, sq(p))$ est un symptôme, c'est par celui-ci que commence la recherche d'un symptôme minimal.

En plus d'avoir expliqué les algorithmes classiques de recherche d'atome non couvert, nous avons découvert une famille plus vaste d'algorithmes qui permet d'envisager des optimisations du type "diviser pour régner" ou bien guidées par des heuristiques. Les seules optimisations qui étaient possibles dans les algorithmes classiques reposaient sur le non-déterminisme de l'ordre pour interroger les nœuds d'une forêt de preuve donnée. Maintenant, c'est le non-déterminisme de l'ordre pour interroger l'ensemble de tous les 0-états qui autorise toute sorte d'optimisations.

Nous donnons maintenant un algorithme de diagnostic d'incorrection négative. Il correspond à l'algorithme de [92]. On remarquera que son expression est ici nettement plus simple, grâce à l'étude qui précède. Enfin sa preuve de correction et complétude a été faite et de manière indépendante de l'algorithme lui-même.

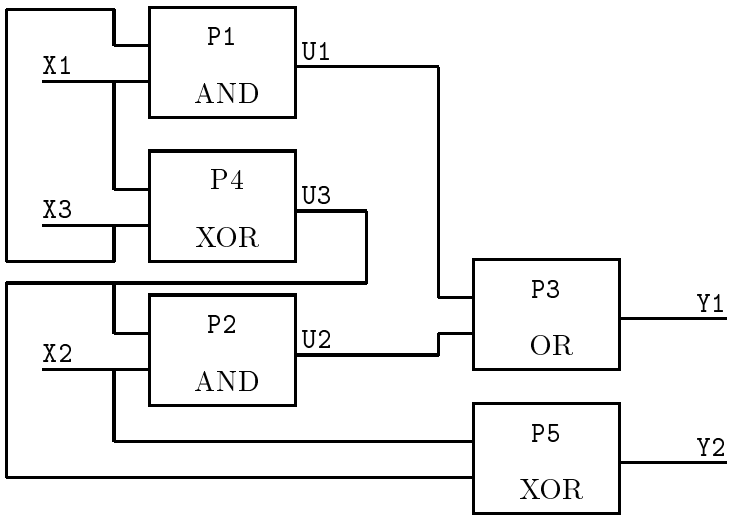
Soit $T = (dom_r^p, \hookrightarrow_r^p)$ un arbre SLD sans co-routinage tel que $(0, sq(p))$ n'est pas attendu. On suppose donné un ordre sur les fils de tout nœud de T, i.e. on oriente l'arbre T (par exemple à partir de l'ordre des clauses dans le programme).

La paire associée à $diagnostic(sq(p))$ est une incorrection négative, où *diagnostic* est la fonction :

```
fonction diagnostic(S : état) retourne état
début fonction
    X ⟵ {S' |  il existe une suite finie de 1-état (1,S1)···(1,Sm),
               (0,S') <T (1,Sm) <T ··· <T (1,S1) <T (0,S)}
    pour i de 1 à card(X)
        soit Si le i^ème état de X trouvé lors du parcours gauche en profondeur de T
    fin pour
    i ⟵ 1
    début boucle
        si i > n alors sortir boucle
        interroger l'oracle pour (0,Si)
        si réponse = NON alors retourner diagnostic(Si)
        i ⟵ i + 1
    fin boucle
    retourner S
fin fonction
```

On illustre l'algorithme par l'exemple suivant (similaire à celui traité dans [92]) dû à Colmerauer [21] en Prolog III utilisant le solveur sur les booléens [8].

Exemple 4.5.1 Détection de pannes dans un additionneur binaire en Prolog III
On cherche à détecter des composants défectueux dans un additionneur binaire qui calcule la somme de trois bits x_1, x_2, x_3 sous forme d'un nombre binaire sur deux bits $y_1 y_2$. Le circuit est le suivant :



On ne s'intéresse qu'au cas où un seul des composants est en panne.

La clause $a \leftarrow C \,\square\, A$ s'écrit en Prolog III (syntaxe marseilleise) :

```
a -> A , C ;
```

(en particulier un fait $a \leftarrow C \,\square\, \varepsilon$ s'écrit `a -> , C ;`).

Le programme DETECT1 exprime la relation entre les entrées, les sorties et le composant défectueux.

```
circuit(C,E,S) -> au_plus_1(C,X) , {C = [P1,P2,P3,P4,P5], E = [X1,X2,X3],
                  S = [Y1,Y2], X = 1', ~P1 => (U1 <=> (X1 & X3)),
                  ~P2 => (U2 <=> (X2 & U3)), ~P3 => (Y1 <=> (U1 | U2)),
                  ~P4 => (U3 <=> ~(X1 <=> X3)), ~P5 => (Y2 <=> ~(X2 <=> U3))} ;

au_plus_1(L,X) -> , {L = [], X = 0'} ;
au_plus_1(L,X) -> au_plus_1(Q,Z) , {L = [Y|Q], X = (Y | Z), (Y & Z) = 0'} ;

booleen(X) -> , {X = 0'} ;
booleen(X) -> , {X = 1'} ;

enumere(L) -> , {L = []} ;
enumere(L) -> booleen(X) enumere(Q) , {L = [X|Q]} ;

go(C,E,S) -> circuit(C,E,S) enumere(E) , {C = [0',0',0',0',1'], S = [1',1']} ;
```

La variable `C` est la liste des composants (`P1` à `P5`), un composant est défectueux si sa valeur est `1'`, la variable `E` est la liste des entrées (`X1` à `X3`) et la variable `S` est la liste des sorties (`Y1` et `Y2`). Les variables existentielles `U1`,`U2`,`U3` de la première clause sont les sorties des composants `P1`,`P2`,`P4` comme indiqué sur le schéma précédent.
Le but `go(C,E,S)` fournit des jeux de tests dans le cas ou le composant `P5` est défectueux (pour simplifier, on impose que la sortie soit `[1',1']`).

> *Remarque.* Il ne faut pas confondre notre objectif qui est de recherché des erreurs dans le programme avec l'exemple qui est un programme qui recherche des composants défectueux dans un additionneur binaire. ◇

Les réponses fournies par Prolog III sont :

```
> go(C,E,S) ;
{C = [0',0',0',0',1'], E = [0',1',1'], S = [1',1']}
{C = [0',0',0',0',1'], E = [1',0',1'], S = [1',1']}
{C = [0',0',0',0',1'], E = [1',1',0'], S = [1',1']}
{C = [0',0',0',0',1'], E = [1',1',1'], S = [1',1']}
```

La dernière réponse s'expliquent par le fait qu'un composant défectueux n'a pas sa sortie erronée pour toutes les entrées possibles. C'est pour cette raison que nous avons mis des implications dans le programme : le composant `P2` fonctionne correctement si $\texttt{P2} = 0'$, i.e. pour tout `X2`,`U3`,`U2` la conjonction des entrées `X2 & U3` est équivalente à la sortie `U2`.

Afin d'illustrer le diagnostic d'incorrection négative nous donnons le programme DETECT2 qui est une autre implantation du problème (erronée évidement).

```
circuit(C,E,S) -> au_plus_1(C,X) , {C = [P1,P2,P3,P4,P5], E = [X1,X2,X3],
                  S = [Y1,Y2], X = 1', ~P1 => (U1 <=> (X1 & X3)),
                  ~P2 => (U2 <=> (X2 & U3)), ~P3 => (Y1 <=> (U1 | U2)),
                  ~P4 => (U3 <=> (~X1 & ~X3)), ~P5 => (Y2 <=> (~X2 & ~U3))} ;

au_plus_1(L,X) -> , {L = [], X = 0'} ;
au_plus_1(L,X) -> au_plus_1(Q,Z) , {L = [Y|Q], X = (Y | Z), (Y & Z) = 0'} ;

booleen(X) -> , {X = 0'} ;
booleen(X) -> , {X = 1'} ;

enumere(L) -> , {L = []} ;
enumere(L) -> booleen(X) enumere(Q) , {L = [X|Q]} ;

go(C,E,S) -> circuit(C,E,S) enumere(E) , {C = [0',0',0',0',1'], S = [1',1']} ;
```

Les réponses fournies par Prolog III pour le but `go(C,E,S)` du programme DETECT2 sont :

```
> go(C,E,S) ;
{C = [0',0',0',0',1'], E = [0',1',0'], S = [1',1']}
{C = [0',0',0',0',1'], E = [1',0',1'], S = [1',1']}
{C = [0',0',0',0',1'], E = [1',1',1'], S = [1',1']}
```

On constate que $(0, sq(\texttt{go(C,E,S)}))$ est un symptôme. $\{\} \square \texttt{go(C,E,S)}$ est non complètement couvert parce qu'il existe un store C' tel que $\texttt{go(C,E,S)} \leftarrow C'$ est non couvert. Par exemple, on peut prendre $C' = \{\texttt{(C=[0',0',0',0',1'], S=[1',1'], E=[0',1',1'])}\}$, ou $C' = \{\texttt{(C=[0',0',0',0',1'], S=[1',1'], E=[1',1',0'])}\}$.

Voyons la session de diagnostic pour le programme DETECT2 et le but `go(C,E,S)`.
Les stores présentés ont été simplifiés par Prolog III.

Symptôme : `go(C,E,S)`

`circuit(C,E,S)` $\square$ `{C = [0',0',0',0',1'], S = [1',1']}`
est-il complètement couvert par
`{C = [0',0',0',0',1'], E = [X1,X2,X3], S = [1',1'], X2|X3 = 1', X1 => X3, X3 => X1, X1|X2 = 1'}`
? `NON`

`au_plus_1(C,X)` $\square$ $\{\texttt{C} = [0', 0', 0', 0', 1']\}$
est-il complètement couvert par
`{ {C = [0',0',0',0',1'], X = 1'} }`
? `OUI`

`enumere(E)` $\square$ $\{\texttt{E} = [\texttt{X1}, \texttt{X2}, \texttt{X3}], \texttt{X2}|\texttt{X3} = 1', \texttt{X1} => \texttt{X3}, \texttt{X3} => \texttt{X1}, \texttt{X1}|\texttt{X2} = 1'\}$
est-il complètement couvert par
`{ {E = [0',1',0']}, {E = [1',0',1']},{E = [1',1',1']} }`
? `OUI`

Incorrection négative : `circuit(C,E,S)` $\square$ `{C = [0',0',0',0',1'], S = [1',1']}`

L'algorithme a isolé une définition responsable du symptôme. L'erreur vient du fait que dans la définition de `circuit` les "XOR" ont été implantés par des "NOT OR".

Sur cet exemple, on constate que $C' \square go(C,E,S)$ est aussi non complètement couvert, où $C' = \{\texttt{(C=[0',0',0',0',1'], S=[1',1'], E=[A,}\sim \texttt{A,1'])}\}$. Dans ce cas on peut rechercher une erreur à partir de $C' \square go(C,E,S)$. En effet, on peut ajouter la clause `go' -> go(C,E,S), C' ;` au programme, et considérer que $\{\} \square$ `go'` est non attendu. Dans ce cas on retrouve l'exemple traité dans [92] qui aboutit à une erreur encore plus précise.

Dans cette section, nous n'avons donné une méthode de diagnostic d'incorrection négative que si l'arbre SLD associé au symptôme négatif calculé est sans co-routinage. Cette condition permet d'assurer que les questions d'incomplétude sont finies, et que l'on peut calculer les prolongements d'un 0-état de manière effective.

Les arbres SLD calculés par un système de PLC standard (standard international ISO de Prolog [31]) peuvent être avec co-routinage. Cependant, en l'absence de méta-prédicat de contrôle (comme `delay`, `freeze`, etc.) les arbres SLD sont sans co-routinage (ils respectent la règle de calcul standard). De plus, on constate que de nombreuses parties, des arbres SLD avec co-routinages calculés, sont sans co-routinage (on peut commencer par interroger ces parties). Et parfois on peut déduire de l'arbre SLD calculé un arbre SLD sans co-routinage.

Maintenant que nous avons bien expliqué les algorithmes de recherche d'incorrection négative, on comprend pourquoi la condition de non co-routinage est nécessaire. On pouvait penser trouver des méthodes pour étendre ces algorithmes à des arbres SLD avec co-routinage en conservant des questions atomiques (qui ne font intervenir qu'un atome). On comprend pourquoi c'est impossible grâce à notre définition claire de l'arbre SLD.

D'autres techniques ont été envisagées, mais celles qui permettent d'avoir des interactions simples avec l'oracle ne fournissent pas de notion d'erreur intéressante.

En revanche, si on s'autorise à poser des questions non atomiques alors on peut trouver d'autres algorithmes.

Par exemple, à l'extrême, on peut considérer que le programme P est une abréviation pour $\mathrm{IFF}(P)$. On présente brièvement une méthode similaire à celle développée dans [63] pour le langage ESCHER.

Dans ce cas, on peut voir la sémantique opérationnelle comme du remplacement de $=$ par $=$:

- le programme est un ensemble d'égalités $a = (C^1 \square a^1_1 \cdots a^1_{n_1}) \vee \cdots \vee (C^m \square a^m_1 \cdots a^m_{n_m})$
- on passe du but

$$B^1 \vee \cdots \vee B^{i-1} \vee (C^i \square a^i_1 \cdots a^i_{j-1} a^i_j a^i_{j+1} \cdots a^i_{n_i}) \vee B^{i+1} \vee \cdots \vee B^l$$

où a^i_j est l'atome sélectionné, et les B^k sont de la forme $C^k \square a^k_1 \cdots a^k_{n_k}$ au but

$$B^1 \vee \cdots \vee B^{i-1} \vee B^{(i,1)} \vee \cdots \vee B^{(i,n)} \vee B^{i+1} \vee \cdots \vee B^l$$

où $a^i_j = (D^1 \square b^1_1 \cdots b^1_{n_1}) \vee \cdots \vee (D^m \square b^m_1 \cdots b^m_{n_m})$ est une égalité du programme (dont les variables locales sont renommées à part) et les $B^{(i,k)}$, $k = 1, \ldots, n$ sont de la forme $C^i \wedge D^h \square a^i_1 \cdots a^i_{j-1} b^h_1 \cdots b^h_{n_h} a^i_{j+1} \cdots a^i_{n_i}$ pour les $h \in \{1, \ldots, m\}$ tels que $C^i \wedge D^h$ n'est pas rejeté.

Le calcul est alors déterministe et de la forme :

$$\begin{array}{c} B^1_1 \\ \| \\ B^2_1 \vee \cdots \vee B^2_{n_2} \\ \| \\ \cdots \\ \cdots \\ \cdots \\ \| \\ B^i_1 \vee \cdots \vee B^i_{n_i} \\ \| \\ B^{i+1}_1 \vee \cdots \vee B^{i+1}_{n_{i+1}} \\ \| \\ \cdots \\ \cdots \\ \cdots \\ \| \\ B^m_1 \vee \cdots \vee B^m_{n_m} \end{array}$$

où chaque B^m_k, $k = 1, \ldots, n_m$, est de la forme $C^m_k \square \varepsilon$ et B^1_1 est de la forme $\emptyset \square a$ ($B^m_1 \vee \cdots \vee B^m_{n_m}$ est le $\vee$-store réponse au but $\leftarrow a$).

La sémantique opérationnelle est simple et le diagnostic déclaratif d'incorrection encore plus. Le $\vee$-store réponse est un symptôme si $a = B^m_1 \vee \cdots \vee B^m_{n_m}$ n'est pas attendu. C'est qu'il existe une égalité $B^i_1 \vee \cdots \vee B^i_{n_i} = B^{i+1}_1 \vee \cdots \vee B^{i+1}_{n_{i+1}}$ non attendue. L'erreur peut se définir à partir de l'égalité du programme qui a permis de passer de l'état i à l'état $i+1$.

Cependant, on voit toute la difficulté de l'interaction avec l'oracle puisque ce dernier doit se prononcer sur des objets très complexes, i.e. il doit déterminer si une égalité de la forme $B_1^i \vee \cdots \vee B_{n_i}^i = B_1^{i+1} \vee \cdots \vee B_{n_{i+1}}^{i+1}$ est attendue, où chaque B_y^z est de la forme $C_y^z \square A_y^z$ où A_y^z est une suite finie d'atomes.

On comprend pourquoi Lloyd s'intéresse tant au problème de présentation (i.e. comment présenter des question longues et complexes à l'utilisateur).

4.6 Insuffisance : réponses manquantes

Nous nous contentons d'une esquisse de l'adaptation des techniques de recherche d'insuffisance connues à la programmation logique avec contraintes. La principale différence avec la programmation logique réside dans le problème de non couverture unique largement discuté dans le chapitre 3.

Cette section se base sur les résultats de [92].

On reprend une partie de la terminologie et des définitions introduites dans la section 4.5.

On suppose un programme P et une relation de couverture $\vdash$ fixés. Le critère de rejet RC est défini par la relation unaire $\vdash \emptyset$.

La sémantique attendue est toujours formalisée par un ensemble d'atomes contraints I.

Le schéma général de la section 4.1 peut s'appliquer à l'opérateur $T^\circ_{\vdash,P}$.

Définition 4.6.1 Symptôme d'insuffisance
Un symptôme d'insuffisance *est un atome contraint* $a \leftarrow C \in I \setminus pgpf(T^\circ_{\vdash,P})$.

Définition 4.6.2 Insuffisance
Une insuffisance *est un atome contraint* $a \leftarrow C$ *non couvert par* P *relativement à* I *(i.e.* $a \leftarrow C \in I \setminus T^\circ_{\vdash,P}(I)$*).*

Définition 4.6.3 Symptôme d'insuffisance calculé
Un symptôme d'insuffisance calculé *est un atome contraint* $a \leftarrow C$ *tel qu'il existe un arbre SLD fini pour le but* $\leftarrow a$ *et* $C \not\vdash \mathrm{SS}(P,a)$ *(*$\mathrm{SS}(P,a)$ *est le* $\vee$*-store réponse pour le but* $\leftarrow a$*).*

Pour que la méthode s'applique aux symptômes effectivement calculés, il faut montrer qu'un symptôme d'insuffisance calculé est un cas particulier de symptôme d'insuffisance. Ce cas particulier correspond à l'existence d'un arbre SLD fini pour le but concernant l'atome du symptôme.

Les trois premiers lemmes sont intermédiaires à la preuve du lemme 4.6.7.

Lemme 4.6.4 *S'il existe un arbre SLD fini pour le but* $\leftarrow a$ *et le* $\vee$*-store réponse associé est* $\mathrm{SS}(P,a) = \{C_1, \ldots, C_m\}$ *alors il existe un entier* k *tel que, pour tout store* C*, si* $a \leftarrow C \in T_P \downarrow k$ *alors* $C \vdash \emptyset$ *(i.e.* C *est rejeté) ou il existe* $1 \leq i \leq m$ *tel que* $C = C_i$*.*

> *Preuve.* On fait correspondre un élément $a \leftarrow C$ de $T_P \downarrow k$ à un squelette S de profondeur $\leq k$ qui a toutes ses feuilles indéfinies à la profondeur k et est tel que $\mathrm{AC}(S,a) = C$.
> Si k' est la profondeur de l'arbre SLD, on prend $k > k'$ et on constate qu'un squelette qui correspond à un élément $a \leftarrow C$ de $T_P \downarrow k$ est soit une réponse, soit tel que $C \vdash \emptyset$. ■

Lemme 4.6.5 *Pour tout entier* n*, si* $a \leftarrow C \in T^\circ_{\vdash,P} \downarrow n$ *alors il existe un ensemble de stores* R *tel que* $C \vdash R$ *et, pour tout store* $C' \in R$*,* $a \leftarrow C' \in T_P \downarrow n$*.*

Preuve. Le lemme se prouve par récurrence sur l'entier n.

1. Si $a \leftarrow C \in T^{\circ}_{\vdash,P} \downarrow 0$ alors il suffit de prendre $R = \{C\}$.
2. Pour l'hérédité de l'hypothèse, on utilise le fait que $T^{\circ}_{\vdash,P} \downarrow n{+}1 = T_{\vdash}(T_P(T^{\circ}_{\vdash,P} \downarrow n))$ et les propriétés **CONJ**, **EXIS** et **TRAN** de la relation de couverture.

■

Lemme 4.6.6 *Si $a \leftarrow C \in T^{\circ}_{\vdash,P} \downarrow \omega$ et il existe un arbre SLD fini pour le but $\leftarrow a$ et $\mathrm{SS}(P,a) = \{C_1, \ldots, C_m\}$ alors il existe $R \subseteq \{C_1, \ldots, C_n\}$ tel que $C \vdash R$ (i.e. $a \leftarrow C \in SS_{\vdash}(P)$).*

Preuve. Le lemme 4.6.5 montre que, pour tout entier n, il existe R_n tel que $C \vdash R_n$ et, pour tout store $C' \in R_n$, $a \leftarrow C' \in T_P \downarrow n$.
Le lemme 4.6.4 assure qu'il existe un entier k tel que si $a \leftarrow C' \in T_P \downarrow k$ alors $C' \vdash \emptyset$ ou il existe $1 \leq i \leq m$ tel que $C' = C_i$.
Il suffit de prendre $R = R_k$. ■

Nous pouvons maintenant énoncé le lemme principal qui établi qu'un symptôme d'insuffisance calculé est un cas particulier de symptôme d'insuffisance.

Lemme 4.6.7 *Si $a \leftarrow C$ est un symptôme d'insuffisance calculé alors $a \leftarrow C$ est un symptôme d'insuffisance.*

Preuve. Soit $a \leftarrow C$ un symptôme d'insuffisance calculé. Si $a \leftarrow C$ n'est pas un symptôme d'insuffisance alors $a \leftarrow C \in pgpf(T^{\circ}_{\vdash,P})$, donc $a \leftarrow C \in T^{\circ}_{\vdash,P} \downarrow \omega$ et $C \vdash \mathrm{SS}(P,a)$ (lemme 4.6.6), or $a \leftarrow C$ est un symptôme d'insuffisance calculé, donc $a \leftarrow C$ est un symptôme d'insuffisance. ■

La méthode généralise celle utilisée en programmation logique (section 4.2), puisqu'on remplace la condition d'existence d'un arbre SLD d'échec fini pour le symptôme par la condition d'existence d'un arbre SLD fini. C'est parce que les deux niveaux de réponses considérés permettent de généraliser la sémantique négative classique liée à l'échec fini (notons qu'il était moins évident de considérer des "disjonctions" de substitutions réponses dans le cadre de la programmation logique).

L'algorithme de diagnostic est basé sur les mêmes principes que l'algorithmes de diagnostic de la section 4.1.2. À cette différence qu'il utilise des règles de couverture éventuellement non finitaires. Les règles de couverture correspondent, ramené au cadre de la section 4.2, à la tentative de construction en parallèle de plusieurs arbres de preuve.

La non finitude potentielle des règles de couverture pose le problème d'effectivité du diagnostic. Si la relation de couverture est compacte (par exemple celle définie à partir d'une théorie) ce problème ne se pose pas. En revanche, si elle n'est pas compacte alors supposons que l'utilisateur ne puisse répondre à une question par une couverture finie alors deux cas sont a envisager :

1. c'est parce qu'il n'existe pas de couverture possible alors on a trouvé une insuffisance ;
2. c'est parce que toute couverture possible est infinie alors on constate que pour obtenir l'atome contraint par un calcul fini, la meilleure solution est peut-être de l'ajouter comme clause du programme, même si l'erreur qui a causé le symptôme ne se situe pas nécessairement dans la définition correspondante.

   Dans tous les cas, si on veut utiliser une règle de couverture non finitaire, la notion d'erreur n'est pas maniable de manière effective.

Cet algorithme demande donc à l'utilisateur de fournir des contraintes pour couvrir ce qui est attendu. On remarque qu'il y a beaucoup plus de contraintes à fournir qu'il y avait de substitutions à fournir en programmation logique : la méthode revient à explorer plusieurs arbres de preuve quand la propriété d'indépendance des contraintes négatives n'est pas vérifiée.

Aucun prototype n'a été réalisé pour le diagnostic d'insuffisance, mais on peut s'attendre à une grande complexité de l'interaction avec l'oracle. C'est pourquoi nous avons privilégié le diagnostic d'incorrection négative.

Malgré tout cette méthode reste une piste intéressante dans le cas où il n'existe pas d'arbre SLD sans co-routinage (ou à partir de l'arbre SLD calculé on n'a pas pu déduire un arbre SLD sans co-routinage).

En comparant les deux types de définitions, nous allons constater qu'une collaboration des deux méthodes pourrait être envisagée.

Un symptôme d'insuffisance calculé est un atome contraint $a \leftarrow C$ vérifiant la propriété :

- $a \leftarrow C \in I$;
- $C \not\vdash \mathrm{SS}(P, a)$, i.e. $C \not\vdash R(C \,\square\, a)$ (**REFL** et **CONJ**).

Donc $a \,\square\, C$ est un symptôme d'incorrection négative.

De la même manière, un symptôme d'incorrection négative calculé est une paire $a \,\square\, C$ vérifiant :

- il existe $C' \vdash \{C\}$;
- $a \leftarrow C' \in I$;
- $C' \not\vdash R(C \,\square\, a)$, i.e. $C' \not\vdash \mathrm{SS}(P, a)$ (sinon $C' \wedge C \vdash R(C \,\square\, a)$ or $C' \vdash \{C' \wedge C\}$ d'où la contradiction).

Donc si l'utilisateur fournit la contrainte C' (celle dont on lui demande si elle existe dans la question d'incomplétude) alors $a \leftarrow C'$ est un symptôme d'insuffisance.

Par conséquent, il semble qu'il soit possible de passer d'un type de symptôme à l'autre. Si cette voie, qui reste à explorer, s'avère fructueuse alors (lors d'une session de diagnostic pour des réponses manquantes) on peut imaginer utiliser le premier algorithme pour les parties de l'arbre SLD sans co-routinage et le second algorithme pour les autres parties.

Le résultat suivant renforce aussi cette perspective :

Lemme 4.6.8 *Il existe une insuffisance si et seulement si il existe une incorrection négative.*

Preuve.

$\Rightarrow$ Soit $a \leftarrow C$ une insuffisance, i.e. $a \leftarrow C \notin I \setminus T^{\circ}_{\vdash,P}(I)$. On a $C' \vdash \{C'\}$, $a \leftarrow C' \in I$ mais $a \leftarrow C'$ non couvert par P par rapport à I, donc $a \,\square\, C'$ est une incorrection négative.

$\Leftarrow$ Soit $a \,\square\, C$ une incorrection négative, i.e. il existe C' tel que $C' \vdash \{C\}$, $a \leftarrow C' \in I$ mais $a \leftarrow C'$ n'est pas couvert par P par rapport à I, donc $a \leftarrow C'$ est une insuffisance.

■

D'où le corollaire suivant, qui s'applique bien entendu aux symptômes calculés :

Corollaire 4.6.9 *S'il existe un symptôme d'insuffisance (respectivement un symptôme d'incorrection négatif) alors il existe une incorrection négative (respectivement une insuffisance).*

Enfin, notons que c'est l'opérateur $T^{\circ}_{\vdash,P}$ qui a été choisit pour les définitions. Soulignons qu'en général : $pgpf(T^{\circ}_{\vdash,P}) \neq pgpf(T^{\cup}_{\vdash,P}) \neq T_{\vdash}(pgpf(T_P))$

Chapitre 5

Conclusion

Deux contributions aux domaines de la programmation logique avec contraintes se dégagent de ce travail :

1. Une reformulation complète de la sémantique opérationnelle et de la sémantique déclarative des programmes logiques avec contraintes en termes de squelettes et d'arbres de preuve.

   Notre cadre permet de rendre compte de l'incomplétude des solveurs de contraintes grâce au critère de rejet pour la sémantique opérationnelle et grâce à la relation de couverture pour la sémantique déclarative. Le critère de rejet abstrait la notion de satisfiabilité d'un store dans une interprétation du langage des contraintes ou dans une théorie dans le langage des contraintes. La relation de couverture étend le critère de rejet pour abstraire la notion de couverture d'un store par un ensemble (éventuellement infini) de stores.

   Cette reformulation dégage les deux niveaux de calcul manipulés en programmation logique avec contraintes : dérivations SLD et arbres SLD ; les premiers peuvent être vu comme des calculs positifs et les seconds comme des calculs négatifs. Cela permet de distinguer deux sémantiques : la sémantique positive et la sémantique négative. En général la sémantique négative se définit à partir des arbres SLD d'échec fini. Nous considérons qu'une bonne généralisation est de la définir à partir des arbres SLD finis en considérant les $\vee$-stores réponses (on retrouve le cas particulier de l'échec fini quand le $\vee$-store réponse est vide).

2. Une étude du diagnostic déclaratif d'erreur en programmation logique avec contraintes mettant à profit la reformulation en termes de squelettes et d'arbres de preuve de la sémantique des programmes logiques avec contraintes.

   Nous avons étudié le cas des réponses fausses (incorrection positive) et le cas des réponses manquantes (incorrection négative et insuffisance). Pour la recherche d'incorrection négative, nous avons contribué à éclaircir les fondements théoriques de l'algorithme que nous proposions dans [92]. Cet algorithme est une double généralisation (à la programmation logique avec contraintes et aux arbres SLD sans co-routinage) d'un algorithme connu pour la programmation logique [68]. De ce point de vue, nous contribuons également à la compréhension formelle des algorithmes de recherche d'atomes non complètement couverts de la programmation logique : ce sont de simples algorithmes de recherche d'incorrections, mais pour un système de règles compliqué.

   Nous avons défini un nouveau schéma général pour le diagnostic basé sur une relation bien fondée, qui n'explique pas directement la notion d'erreur, mais qui rend compte fidèlement

des algorithmes de diagnostic d'incorrection qui ne sont que la recherche d'élément minimaux parmi les symptômes.

Nous n'avons imposé aucune condition sur les programmes, telle que la propriété d'être acceptable [4]. Nous n'avons imposé aucune condition sur le solveur de contraintes, telle que la complétude. La seule condition que nous ayons imposé est l'existence d'un arbre SLD fini sans co-routinage pour le diagnostic déclaratif d'incorrection négative. Cependant, les arbres SLD calculés par les systèmes ont généralement de nombreuses parties sans co-routinage. De plus, nous discutons dans la section 4.6 des méthodes qui permettraient une collaboration judicieuse entre les techniques de recherche d'insuffisance et de recherche d'incorrection négative. Une telle collaboration peut être possible grâce à notre définition des arbres SLD (avant cette définition, il était difficile de l'envisager). Cela permettrait de diagnostiquer des erreurs dans le programme dans le cas de réponses manquantes en conservant, quand c'est possible, une interaction simple avec l'oracle.

Nous avons comparé notre modélisation de la sémantique opérationnelle avec les modèles plus classiques et nous avons montré qu'elle décrit une abstraction du comportement opérationnel des systèmes réels.

Rappelons que nous avons fait le choix de voir un système de PLC comme un vrai langage de programmation et non comme un système de résolution de contraintes uniquement.

Le problème de présentation (voir section 4.2) n'a pas été abordé. C'est un problème qui dépasse largement le cadre du diagnostic. En effet, lors du diagnostic on interroge l'utilisateur pour savoir si le résultat d'un calcul est attendu. Le problème de présentation de ce résultat existe déjà quand on considère l'implantation du système de PLC lui-même, et au delà il existe déjà quand on s'intéresse au système de résolution de contraintes. Des solutions sont en partie connues, par exemple en utilisant des approximations, l'énumération des solutions, l'élimination des variables quantifiées existentiellement, des outils graphiques, des transformations des formules conservant l'ensemble des solutions, etc.

Le niveau de cette reformulation convient pour étudier l'aspect programmation, mais pas encore l'aspect résolution de contraintes.

Par exemple, si le programme est de la forme (où C est un très gros stores)

$$go(x_1, \ldots, x_n) \leftarrow C \,\square\, \varepsilon$$

et qu'on observe un symptôme à l'exécution, les méthodes de recherche d'erreur que nous présentons sont sans intérêt. On peut discuter de la méthodologie de programmation et préférer un programme mieux structuré : en général il est plus facile de découper un problème en sous-problème... mais ce n'est pas notre propos.

Notons que le programme ci-dessus peut à peine être considéré comme un programme. C'est plutôt l'utilisation d'un système de *programmation* logique avec contraintes comme simple système de résolution de contraintes : seule une partie des possibilités du système est utilisée (c'est comme si l'on utilisait un langage de programmation impérative pour calculer la valeur de $(1+\sqrt{5})/2$ alors qu'on dispose d'une calculette[1]).

Chercher des erreurs dans le store C n'a pas de sens dans notre cadre au sens strict : les prédicats de contrainte sont supposés corrects, c'est le programme que nous déboguons non le système. Cependant, s'il y a un symptôme c'est bien qu'une erreur a été commise lors de l'écriture du store C.

[1] Cependant, peu de programmeurs peuvent affirmer n'avoir jamais utilisé un langage de programmation de cette manière.

Il serait donc important de se placer dans un autre cadre pour chercher des moyens formels appropriés afin de diagnostiquer des erreurs dans des systèmes de contraintes purs. C'est-à-dire s'intéresser au diagnostic pour les problèmes de satisfaction de contraintes.

Nous n'avons pas parlé d'implantation des algorithmes décrits dans le chapitre 4. Les implantations réalisées actuellement sont à l'état de prototype. Notamment elles n'intègrent pas d'interface conviviale et sont encore semi-automatiques en ce qui concerne le problèmes des réponses manquantes.

Comme pour la programmation logique, l'implantation de prototypes écrits dans le langage lui-même soulève des problèmes de méta-programmation. La difficulté supplémentaire est la manipulation des contraintes au niveau méta. Cette difficulté intervient par exemple lors du problème de présentation. Elle est en partie simplifiée en Prolog III grâce au prédicat `outc` qui affiche un terme avec les contraintes associées aux variables qui figurent dans ce terme.

On constate que la représentation avec variables du programme objet fonctionne correctement (comme en programmation logique). Par exemple le méta-interprète *vanilla* :

$$
\begin{array}{l}
solve(true) \leftarrow \emptyset \,\square\, \varepsilon \\
solve(x) \leftarrow \emptyset \,\square\, clause(x\ if\ y)\ solve(y) \\
solve(x\ et\ y) \leftarrow \emptyset \,\square\, solve(x)\ solve(y)
\end{array}
$$

où la définition du prédicat *clause* est : pour toute clause $a \leftarrow C \,\square\, a_1 \cdots a_n$ du programme objet, le méta-interprète a un fait $clause(a\ if\ a_1\ et\ \cdots\ et\ a_n\ et\ true) \leftarrow C \,\square\, \varepsilon$. On peut noter que c'est la définition exacte du prédicat `clause` de Prolog III. Ce langage semble être le mieux adapté pour l'implantation de prototypes méta-programmés.

Cette difficulté à manipuler les contraintes au niveau méta, en partie simplifiée en Prolog III, amène à penser que les algorithmes de diagnostic seraient plus simples à implanter dans le système lui-même et non au niveau méta. Un interprète manipulant les contraintes sur les intervalles sur $\mathbb{N}$ et sur $\mathbb{R}$ est en cours de développement au Laboratoire d'Informatique Fondamental d'Orléans sous la direction de Frédéric Benhamou. Nous envisageons d'intégrer nos algorithmes de diagnostic déclaratif d'erreur dans ce système.

Nous n'avons pas parlé de l'optimisation possible de nos algorithmes. Toutes les optimisations décrites dans la section 4.2 sont envisageables. De plus, en tenant compte de l'interprétation sous-jacente des contraintes on pourrait exhiber des heuristiques permettant d'optimiser le nombre de questions ou de déterminer un ordre allant des questions les plus simples aux questions les plus complexes.

Il est important de noter que le diagnostic déclaratif d'erreur n'évite pas tout travail de la part de l'utilisateur et qu'il n'est qu'une aide pour la recherche des erreurs dans le programme (il n'y a pas de miracle !).

Rappelons enfin que ces outils doivent s'intégrer dans un environnement général de mise au point de programme, et qu'ils ne viennent qu'en appui de toutes les autres techniques permettant d'aider à la correction du programme.

Appendice A

Rappels

Cette annexe rappelle les définitions de base utiles dans cette thèse. Les notations et le vocabulaire introduits sont très classiques. Le lecteur trouvera plus de détail sur les différents concepts dans les ouvrages de mathématiques discrètes pour l'informatique (par exemple [53, 6, 99, 66, 82]).

A.1 Relations

Une *relation* R sur $E_1 \times \cdots \times E_n$ est un sous-ensemble de $E_1 \times \cdots \times E_n$ ($\times$ est le produit cartésien).

Étant donné une relation R sur $E_1 \times \cdots \times E_n$, nous notons aussi R la fonction de $E_1 \times \cdots \times E_n$ dans l'ensemble des valeurs de vérité $\{\text{vrai}, \text{faux}\}$ et nous utilisons les notations $(e_1, \ldots, e_n) \in R$ et $R(e_1, \ldots, e_n)$ pour indiquer que $(e_1, \ldots, e_n)$ est un élément de R. Notons que $E_1 \times \cdots \times E_n$ est la relation sur $E_1 \times \cdots \times E_n$ vraie pour tout élément de $E_1 \times \cdots \times E_n$.

Si R est une relation sur E^n nous dirons également que R est une relation n-aire sur E.

Soit R une relation binaire sur E. On note parfois $e_1 \, R \, e_2$ pour $(e_1, e_2) \in R$ et $e_1 \not R \, e_2$ pour $(e_1, e_2) \notin R$. De plus, on dit que :

- R est *réflexive* si pour tout $e \in E$: $(e, e) \in R$;
- R est *irréflexive* si pour tout $e \in E$: $(e, e) \notin R$;
- R est *symétrique* si pour tout $(e_1, e_2) \in E^2$: $(e_1, e_2) \in R$ implique $(e_2, e_1) \in R$;
- R est *antisymétrique* si pour tout $(e_1, e_2) \in E^2$: $(e_1, e_2) \in R$ et $(e_2, e_1) \in R$ implique $e_1 = e_2$;
- R est *transitive* si pour tout $(e_1, e_2, e_3) \in E^3$: $(e_1, e_2) \in R$ et $(e_2, e_3) \in R$ implique $(e_1, e_3) \in R$;
- R est une *relation d'équivalence* si R est réflexive, symétrique et transitive ;
- R est un *ordre* (partiel) si R est réflexive, antisymétrique et transitive ;
- R est un *ordre strict* si R est irréflexive et transitive ;
- R est un *ordre total* (strict) si R est un ordre (strict) et pour tout $(e_1, e_2) \in E^2$, $e_1 \neq e_2$: $(e_1, e_2) \in R$ ou $(e_2, e_1) \in R$.

Soit R une relation sur $E_1 \times E_2$. On note R^{-1} la *relation réciproque* de R sur $E_2 \times E_1$ définie par : pour tout $(e_1, e_2) \in E_1 \times E_2$, $(e_2, e_1) \in R^{-1}$ si et seulement si $(e_1, e_2) \in R$.

Soit R une relation binaire sur E. On appelle *clôture transitive* (respectivement *réflexive transitive*) de R la plus petite relation transitive (respectivement réflexive et transitive) qui contient R. La clôture transitive de R est notée R^+ et sa clôture réflexive transitive est notée R^*.

Soit R une relation binaire sur E. Soit $(e_1, e_2) \in R$. e_1 est un *prédécesseur* de e_2 selon R (et e_2 est un *successeur* de e_1 selon R) s'il n'existe aucun $e \in E$ tel que $(e_1, e) \in R$ et $(e, e_2) \in R$. La relation binaire sur E définie par $\{(e_1, e_2) \in E^2 \mid e_1 \text{ est un prédécesseur de } e_2 \text{ selon } R\}$ est appelée *relation successeur* issue de R (elle est incluse dans R) ; sa relation réciproque est appelée *relation prédécesseur* issue de R.

Soit R une relation d'équivalence sur E. On appelle *classe d'équivalence* de $e \in E$ (modulo R) l'ensemble $\{e' \in E \mid (e', e) \in R\}$. Deux classes d'équivalences sont soit égales soit disjointes ; une classe d'équivalence est entièrement déterminée par un de ses éléments. L'ensemble des classes d'équivalences modulo R est une partition de E notée E/R (le *quotient* de E par R). Afin de simplifier, une classe d'équivalence est notée, en pratique, par n'importe lequel de ses éléments ; le contexte suffit, en général, à établir s'il s'agit de la classe ou de l'élément.

Une relation binaire R sur E est *bien fondée* (ou nœtherienne) si pour tout élément $e_0 \in E$, il n'existe pas de suite infinie $\{e_i\}_{i \in \mathbb{N}}$ d'éléments de E telle que $(e_{i+1}, e_i) \in R$, pour tout $i \in \mathbb{N}$ (i.e. il n'existe pas de suite infinie "décroissante").

La clôture transitive d'une relation bien fondée est un ordre strict bien fondé.

A.2 Mots sur un alphabet

Soit A un ensemble de *lettres* appelé *alphabet*.

L'ensemble des *mots* A^* sur l'alphabet A est l'ensemble des suites finies d'éléments de A. La suite vide est notée ε ; La suite composée des $n > 0$ lettres $a_1, \ldots, a_n$ dans cet ordre est notée $a_1 \cdots a_n$ (n est la *longueur* du mot $a_1, \ldots, a_n$) ; en particulier, la suite composée de l'unique élément $a \in A$ est notée aussi a (en général, le contexte lève toute ambiguïté).

Si $N = a_1 a_2 \cdots a_m$ et $N' = a'_1 a'_2 \cdots a'_n$ sont des éléments de A^*, la *concaténation* de N et N' est la suite finie, notée $N \cdot N'$, définie par : $N \cdot N' = a_1 a_2 \cdots a_m a'_1 a'_2 \cdots a'_n$. N est appelé un *préfixe* (ou facteur de gauche) de $N \cdot N'$.

$(A^*, \cdot, \varepsilon)$ est le monoïde libre engendré par A. Un *langage* sur A^* est un sous-ensemble de A^*.

A.3 Treillis

Soit E un ensemble et $\leq$ un ordre sur E ($(E, \leq)$ est un *ensemble partiellement ordonné*). Soit X un sous-ensemble de E.

On appelle *majorant* de X tout élément $e \in E$ tel que, pour tout $x \in X : x \leq e$. Le dual est la notion de *minorant* de X. Une *borne supérieure* de $X \subseteq E$ est un majorant e de X tel que, pour tout majorant e' de $X : e \leq e'$. La *borne supérieur* de X, si elle existe, est unique et notée $lub(X)$. La notion duale de *borne inférieure* est notée $glb(X)$.

$(E, \leq)$ est un *treillis* si, pour tout $(e_1, e_2) \in E^2$, la borne supérieure et la borne inférieure de $\{e_1, e_2\}$ existent. $(E, \leq)$ est un *treillis complet* si pour tout sous-ensemble X de E : $lub(X)$ et $glb(X)$ existent.

Un sous-ensemble X de E est *dirigé* si tout sous-ensemble fini de X a sa borne supérieure dans X.

Exemple A.3.1 Treillis sur l'ensemble des parties d'un ensemble
Soit E un ensemble. $(2^E, \subseteq)$ est un treillis complet. Si $X \subseteq 2^E$ alors $lub(X) = \bigcup_{x \in X} x$ et $glb(X) = \bigcap_{x \in X} x$. La borne supérieure de 2^E est E et sa borne inférieure est $\emptyset$.

Bibliographie

[1] Peter Aczel. *An Introduction to Inductive Definitions*, chapter 7, pages 739–782. In J. Barwise, editor, Handbook of Mathematical Logic. North-Holland Publishing Company, 1977.

[2] K. R. Apt. *Handbook of Theoretical Computer Science*, volume 2, chapter Logic Programming, pages 493–574. Elsevier, 1990.

[3] K. R. Apt and M. H. Van Emden. Contributions to the Theory of Logic Programming. *Journal of the ACM*, 29(3):841–862, 1982.

[4] K.R. Apt and D. Pedreschi. Reasoning about termination of pure PROLOG programs. *Information and Computation*, 106(1):109–157, 1993.

[5] Krzysztof R. Apt and Dino Pedreschi. Studies in Pure Prolog: Termination. In John W. Lloyd, editor, *Symposium on Computational Logic*, Lecture Notes in Computer Science, pages 150–176. Springer-Verlag, 1990.

[6] A. Arnold and I. Guessarian. *Mathématiques pour l'informatique.* Logique Mathématiques Informatiques. Masson, 1993.

[7] Roberto Bagnara, Marco Comini, Francesca Scozzari, and Enea Zaffanella. The AND-compositionality of CLP computed answer constraints. In Paqui Lucio, Maurizio Martelli, and Marisa Navarro, editors, *Joint Conference on Declarative Programming*, pages 355–366, 1996.

[8] Frédéric Benhamou. Boolean Algorithms in Prolog III. In Alain Colmerauer and Frédéric Benhamou, editors, *Constraint Logic Programming: Selected Research*, chapter 17, pages 307–326. MIT Press, 1993.

[9] Frédéric Benhamou. Heterogeneous Constraint Solving. In Michael Hanus and Mario Rofríguez-Artalejo, editors, *International Conference on Algebraic and Logic Programming*, volume 1139 of *Lecture Notes in Computer Science*, pages 62–76. Springer-Verlag, 1996.

[10] Frédéric Benhamou and Touraïvane. Prolog IV: langage et algorithmes. In *Journées Francophones de Programmation en Logique*, pages 51–65. Teknea, 1995.

[11] Michel Bergère. *Approche déclarative du diagnostic d'erreur pour la programmation en logique avec négation.* PhD thesis, Université d'Orléans, 1991.

[12] Michel Bergère, Gérard Ferrand, François Le Berre, Bernard Malfon, and Alexandre Tessier. La Programmation Logique avec Contraintes revisitée en termes d'arbres de preuve et de squelettes. Technical Report 95/06, LIFO, University of Orléans, 1995.

[13] Michel Bergère, Gérard Ferrand, François Le Berre, Bernard Malfon, and Alexandre Tessier. La Programmation Logique avec Contraintes revisitée en termes d'arbres de preuve et de squelettes. In *Pôle Contraintes et Programmation Logique*. Journées du GDR Programmation du CNRS, 1994.

[14] Dominic Frank Julian Binks. *Declarative Debugging in Gödel*. PhD thesis, University of Bristol, 1995.

[15] Annalisa Bossi, Maurizio Gabrielli, Giorgio Levi, and Maurizio Martelli. The S-Semantics Approach: Theory and Applications. *The Journal of Logic Programming*, 19&20:149–198, 1994.

[16] Miguel Cruz Costa Calejo. *A Framework for Declarative Prolog Debugging*. PhD thesis, Universidade Nova de Lisboa, 1992.

[17] K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[18] Philippe Codognet. Programmation logique avec contraintes : une introduction. *Technique et Science Informatique, Hermes*, 14(6):665–692, 1995. Journées Francophones de Programmation en Logique 1994.

[19] Philippe Codognet and Daniel Diaz. Compiling Constraints in `clp(FD)`. *Journal of Logic Programming*, 27(3):185–226, 1996.

[20] Alain Colmerauer. *Prolog II, Manuel de référence et modèle théorique*, Groupe Intelligence Artificielle Luminy edition, 1982.

[21] Alain Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.

[22] Alain Colmerauer. Combining Constraint domains. Technical Report 3, Acclaim, ESPRIT Project 7195, 1995. Deliverable D2.11/3.

[23] Alain Colmerauer. Specifications of prolog iv. Draft, 1996.

[24] Livio Colussi, Elena Marchiori, and Massimo Marchiori. A Dataflow Semantics for Constraint Logic Programs. In Manuel V. Hermenegildo and S. Doaitse Swierstra, editors, *Symposium on Programming Languages: Implementations, Logics and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 431–448. Springer-Verlag, 1995.

[25] Livio Colussi, Elena Marchiori, and Massimo Marchiori. On Termination of Contraint Logic Programs. In Ugo Montanari and Francesca Rossi, editors, *International Conference on Principles and Practice of Constraint Programming*, volume 976 of *Lecture Notes in Computer Science*, pages 431–448. Springer-Verlag, 1995.

[26] Marco Comini and Giorgio Levi. An Algebraic Theory of Observables. In M. Bruynooghe, editor, *International Symposium on Logic Programming*, pages 172–186. MIT Press, 1994.

[27] Marco Comini, Giorgio Levi, and Giuliana Vitiello. Abstract Debugging of Logic Programs. In Laurent Fribourg and F. Turini, editors, *Logic Program Synthesis and Transformation and Metaprogramming*, volume 883 of *Lecture Notes in Computer Science*, pages 440–450. Springer-Verlag, 1994.

[28] Marco Comini, Giorgio Levi, and Giuliana Vitiello. Declarative Diagnosis Revisited. In John Lloyd, editor, *International Logic Programming Symposium*, Logic Programming, pages 275–287. MIT Press, 1995.

[29] Marco Comini, Giorgio Levi, and Giuliana Vitiello. Efficient Detection of Incompleteness Errors in the Abstract Debugging of Logic Programs. In Mirelle Ducassé, editor, *Automated and Algorithmic Debugging*, pages 159–174. IRISA-CNRS, 1995.

[30] Patrick Cousot and Radhia Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.

[31] Pierre Deransart, AbdelAli Ed-Dbali, and Laurent Cervoni. *Prolog: The Standard, Reference Manual*. Springer-Verlag, 1996.

[32] Pierre Deransart and Jan Małuszyński. *A Grammatical View of Logic Programming*. MIT Press, 1993.

[33] Nachum Dershowitz and Yuh-Jeng Lee. Deductive Debugging. In *Fourth IEEE Symposium on Logic Programming*, pages 298–306, 1987.

[34] Nachum Dershowitz and Yuh-Jeng Lee. Logical Debugging. *Journal of Symbolic Computation*, 15:745–773, 1993.

[35] Daniel Diaz. *Étude de la compilation des langages logiques de programmation par contraintes sur les domaines finis : le système* `clp(FD)`. PhD thesis, Université d'Orléans, 1995.

[36] Mehmet Dincbas, Pascal Van Hentenryck, H. Simonis, and Abderrahmane Aggoun. The Constraint Logic Programming Language CHIP. In *International Conference on Fifth Generation Computer Systems*, pages 249–264, 1988.

[37] Wlodek Drabent, Simin Nadjm-Tehrani, and Jan Małuszyński. Algorithmic Debugging with Assertions. In Harvey Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 501–522. MIT Press, 1989.

[38] François Fages. Programmation Logique avec Contraintes. École jeunes chercheurs, GDR programmation du CNRS, 1995.

[39] François Fages. *Programmation Logique par Contraintes*. ellipses, 1996.

[40] Gérard Ferrand. Error Diagnosis in Logic Programming: an adaptation of E. Y. Shapiro's method. *Journal of Logic Programming*, 4:177–198, 1987.

[41] Gérard Ferrand. The Notions of Symptom and Error in Declarative Diagnosis of Logic Programs. In Peter A. Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 40–57. Springer-Verlag, 1993.

[42] Gérard Ferrand and Alexandre Tessier. Sémantique des Programmes Logiques avec Contraintes fondée sur la Relation de Couverture. In *Pôle Contraintes et Programmation Logique*. Journées du GDR Programmation du CNRS, 1995.

[43] Gérard Ferrand and Alexandre Tessier. Declarative Debugging. *The Newsletter of the European Network in Computational Logic*, 1996. COMPULOG NET.

[44] Melvin Fitting. A Kripke-Kleene Semantics for Logic Programs. *Journal of Logic Programming*, 2:295–312, 1985.

[45] Melvin Fitting. *Computability Theory, Semantics, and Logic Programming*. Number 13 in Oxford Logic Guides. Oxford University Press, 1987.

[46] Peter Fritzson, Tibor Gyimothy, Mariam Kamkar, and Nahid Shahmeri. Generalized Algorithmic Debugging and Testing. *ACM Letters on Programming Languages and Systems*, 1(4):303–322, 1992.

[47] Maurizio Gabbrielli and Giorgio Levi. Modeling answer constraints in Constraint Logic Programs. In Vijay A. Saraswat and K. Ueda, editors, *International Conference on Logic Programming*, pages 238–252. MIT Press, 1991.

[48] Roberto Giacobazzi, Saumya K. Debray, and Giorgio Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *Journal of Logic Programming*, 25(3):191–248, 1995. (Short version available in International Conference on Fifth Generation Computer, pp 581-591, 1992).

[49] Nevin C. Heintze, Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H.C. Yap. *The CLP($\mathcal{R}$) Programmer's Manual Version 1.2*. IBM Thomas J. Watson Research Center, 1992.

[50] Leon Henkin, J. Donald Monk, and Alfred Tarski. *Cylindric Algebras*, volume 64 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, first edition, 1971.

[51] P. M. Hill and John W. Lloyd. Analysis of Meta-Programs. In Harvey Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*. MIT Press, 1989.

[52] P. M. Hill and John W. Lloyd. The Gödel Programming Language. Technical Report CSTR-92-27, Department of Computer Science, University of Bristol, 1992.

[53] Kiyosi Itô. *Encyclopedic Dictionary of Mathematics*, volume 1&2. MIT Press, second edition, 1993.

[54] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In 14^{th} *ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987. (Full paper available as, Department of Computer Science, Monash University, Technical Report 86/73).

[55] Joxan Jaffar and Michael J. Maher. Constraint Logic Programming: a survey. *Journal of Logic Programming*, 19-20:503–581, 1994.

[56] R. A. Kowalski. Predicate Logic as a Programming Language. In *Information Processing*, pages 569–574, 1974.

[57] Jean-Louis Lassez, V. L. Nguyen, and E. A. Sonenberg. Fixed Point Theorems and Semantics: A Folk Tale. *Information Processing Letters*, 14(3):112–116, 1982.

[58] François Le Berre and Alexandre Tessier. Declarative Incorrectness Diagnosis in Constraint Logic Programming. In Paqui Lucio, Maurizio Martelli, and Marisa Navarro, editors, *Joint Conference on Declarative Programming*, pages 379–391, 1996. (Preliminary version available as, LIFO, University of Orléans, Technical Report 95/08).

[59] Yuh-Jeng Lee and Nachum Dershowitz. Debugging Logic Programs Using Specifications. In Peter A. Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 75–84. Springer-Verlag, 1993.

[60] Y. Lichtenstein and Ehud Y. Shapiro. Abstract Algorithmic Debugging. In *Joint International Conference and Symposium on Logic Programming*, pages 512–530. MIT Press, 1988.

[61] John W. Lloyd. Declarative Error Diagnosis. *New Generation Computing*, 5(2):133–154,, 1987.

[62] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987. second edition.

[63] John W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, 1995.

[64] Michael J. Maher. A CLP View of Logic Programming. In *Conference on Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 364–383. Springer-Verlag, 1992.

[65] Michael J. Maher. A Logic Programming view of CLP. In Warren, editor, *International Conference on Logic Programming*, pages 737–753. MIT Press, 1993.

[66] Michel Marchand. *Mathématique discrète.* De Boeck Université, 1989.

[67] Glenford J. Myers. *The Art of Software Testing.* Wiley & Sons, 1979.

[68] Simin Nadjm-Tehrani. *Contributions to the Declarative Approach to Debugging Prolog Programs.* PhD thesis, Linkoping University, 1989.

[69] Lee Naish. Declarative Diagnosis of Missing Answers. *New Generation Computing*, 10(3):255–285, 1992. (Also available as, Melbourne University, Technical Report 88/9).

[70] Lee Naish. *Types in Logic Programming*, chapter Types and the Intended Meaning of Logic Programs, pages 189–216. Logic Programming Series. MIT Press, 1992.

[71] Lee Naish. Declarative Debugging of Lazy Functional Programs. *Australian Computer Science Communications*, 15(1):287–294, 1993. (Also available as, Department of Computer Science, University of Melbourne, Technical Report 92/6).

[72] Lee Naish. A Declarative Debugging Scheme. Technical Report 95/1, Department of Computer Science, University of Melbourne, 1995.

[73] Lee Naish and Timothy Barbour. A Declarative Debugger for a Logical-Functional Language. Technical Report 94/30, Department of Computer Science, University of Melbourne, 1994.

[74] Lee Naish, P. W. Dart, and J. Zobel. The NU-Prolog Debugging Environment. In Giorgio Levi and Maurizio Martelli, editors, *International Conference on Logic Programming*, pages 521–536. MIT Press, 1989.

[75] A. E. Nicholson. Declarative Debugging of the Parallel Logic Programming Language GHC. In *Australian Computer Science Conference*, pages 225–236, 1988.

[76] Henrik Nilsson and Peter Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, 1994.

[77] William J. Older and Frédéric Benhamou. Programming in CLP(BNR). In *Workshop on Principles and Practice of Constraint Programming*, 1993.

[78] Luís Moniz Pereira. Rational Debugging in Logic Programming. In Ehud Y. Shapiro, editor, *International Conference on Logic Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1986.

[79] Luís Moniz Pereira and Miguel Calejo. A Framework for Prolog Debugging. In *International Conference and Symposium on Logic Programming*, pages 481–495. MIT Press, 1988.

[80] PrologIA. *Le manuel de Prolog III*, 1989.

[81] PrologIA. *Le manuel de Prolog IV*, 1996.

[82] Kenneth H. Rosen. *Discrete Mathematics and its Applications.* The Random House/Birkhäuser Mathematics Series. Random House, first edition, 1988.

[83] Salvatore Ruggieri. On Termination of Contraint Logic Programs. In Paqui Lucio, Maurizio Martelli, and Marisa Navarro, editors, *Joint Conference on Declarative Programming*, pages 391–403, 1996.

[84] Vijay A. Saraswat. The Category of Constraint Systems is Cartesian-Closed. In *Logic In Computer Science.* IEEE Press, 1992.

[85] Vijay A. Saraswat. *Concurrent Constraint Programming.* ACM Doctoral Dissertation Awards. MIT Press, 1993. (Also available as, Concurrent Constraint Programming Languages, PhD thesis, Carnegie-Mellon University, 1989).

[86] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *Symposium on Principles of Programming Languages*, pages 333–352. ACM Press, 1991.

[87] Dana S. Scott. Domains for Denotational Semantics. In M. Nielsen and E. M. Schmidt, editors, *Automata, Languages and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer-Verlag, 1982.

[88] Ehud Y. Shapiro. *Algorithmic Program Debugging.* ACM Distinguished Dissertation. MIT Press, 1982.

[89] M. B. Smyth. *Handbook of Logic in Computer Science*, volume 1 Background: Mathematical structures, chapter Topology, pages 641–761. Oxford University Press, 1992.

[90] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* Logic Programming. MIT Press, third edition, 1987.

[91] Alexandre Tessier. Méta-programmation en programmation en logique. Rapport de DEA, LIFO, Université d'Orléans, 1992.

[92] Alexandre Tessier. Declarative Debugging in Constraint Logic Programming. In Joxan Jaffar, editor, *Asian Computing Science Conference*, volume 1179 of *Lecture Notes in Computer Science*, pages 64–73. Springer-Verlag, 1996.

[93] Alexandre Tessier. Declarative Debugging in Constraint Logic Programming. In Michael Hanus, editor, *International Conference on Algebraic and Logic Programming (Poster session)*, pages 38–49. Technical Report 96/9, Aachener Informatik – Berichte, 1996.

[94] Alexandre Tessier. Diagnostic déclaratif d'insuffisance en programmation logique avec contraintes. In Jean-Louis Imbert, editor, *Journées Francophones de Programmation Logique et Programmation par Contraintes*, pages 65–82. HERMES, 1996. (Also available as, LIFO, University of Orléans, Technical Report 96/04).

[95] Alexandre Tessier. Une caractérisation des arbres SLD en programmation logique avec contraintes. In *Pôle Contraintes et Programmation Logique*. Journées du GDR Programmation du CNRS, 1996.

[96] Maarten H. Van Emden. Value Constraints in the CLP scheme. In *International Logic Programming Symposium, post-conference workshop on Interval Constraints*, 1995.

[97] Pascal Van Hentenryck. Constraint Logic Programming. *Knowledge Engineering Review*, 6(3):151–194, 1991. (Also available as, Brown University, Technical Report CS-91-05).

[98] Giuliana Vitiello. *On the Abstract Diagnosis of Logic Programs.* PhD thesis, University of Salerno, 1996.

[99] Nguyen Huy Xuong. *Mathématiques discrètes et informatique.* Logique Mathématiques Informatiques. Masson, 1992.

Notations

$a \leftarrow C \,\square\, A$ Une clause, 27

$\hookrightarrow$ Relation de transition entre états de calcul, 43

$\hookrightarrow_r$ Relation de transition selon la règle de calcul r, 45

$(dom_r^p, \hookrightarrow_r^p)$ Arbre SLD pour le prédicat de programme p selon la règle de calcul r, 47

$\hookrightarrow_r^p$ Relation de transition selon r pour p (relation de parenté de l'arbre SLD), 47

dom_r^p Domaine d'arbre de l'arbre SLD pour p selon r, 47

$\cdot$ Concaténation

$(X_i)_{i \in I}$ Famille indexée par I

$arity(R)$ Arité de la clause R, 27

ATOM Ensemble des atomes

$body(R)$ Corps de la clause R, 27

$clause_P(u)$ Clause dont le nom est u dans le programme P (noté $clause(u)$ quand P est fixé), 28

$cn(P)$ Ensemble des noms des clauses du programme P, 28

$cn(P, p)$ Ensemble des noms des clauses de la définition de p dans P, 28

$coind(\Phi)$ Ensemble défini co-inductivement par Φ, 22

consistent Relation de consistance du store actif, 32

CONST Ensemble des contraintes basiques du langage, 25

$const(S, reno_S)$ Store associé au squelette S et à la fonction de renommage $reno_S$, 36

$def(S)$ Ensemble des nœuds définis du squelette S, 35

dom_T Domaine d'arbre standard d'un arbre orienté étiqueté T, 19

$\exists_{\tilde{x}} F$ Quantification existentielle de F sur les variables de $\tilde{x}$, 25

$\tilde{\exists} F$ Clôture existentielle de F, 25

$\exists_{-\tilde{x}} F$ Clôture existentielle de F sauf sur les variables de $\tilde{x}$, 25

$\exists_{-a} F$ Clôture existentielle de F sauf sur les variables libres de l'atome a, 25

FF(P) Ensemble d'échec fini du programme P

FI(P) Partie "seulement si" du programme P

glb Borne inférieur

$graft(T, N, T')$ Greffe de l'arbre T' sur le nœud N dans l'arbre T, 19

$head(R)$ Tête de la clause R, 27

IF(P) Partie "si" du programme P

IFF(P) Partie "si et seulement si" du programme P

$ind(\Phi)$ Ensemble défini inductivement par Φ, 20

infer Fonction de détermination des contraintes actives et des contraintes passives, 32

lab_T Fonction d'étiquetage d'un arbre orienté étiqueté T, 19

lub Borne supérieur

Π_c Ensemble des symboles de prédicat de contrainte d'un programme, 24

Π_p Ensemble des symboles de prédicat de programme d'un programme, 24

$\mathcal{P}_f(E)$ Ensemble des parties finies de l'ensemble E, 40

$pgpf(T)$ Plus grand point fixe de l'opérateur T, 22

$pppf(T)$ Plus petit point fixe de l'opérateur T, 21

R^+ Clôture transitive de R

R^* Clôture réflexive transitive de R ou ensemble des suites finies d'éléments de R

R^{-1} Relation réciproque de R

RC Critère de rejet du système, 39
- $\mathrm{RC}_{\mathcal{D}}$ Critère de rejet défini par une interprétation $\mathcal{D}$, 39
- $\mathrm{RC}_{\mathcal{T}}$ Critère de rejet défini par une théorie $\mathcal{T}$, 39
- $\mathrm{RC}_{\mathcal{A}}$ Critère de rejet défini par un algorithme de test de satisfaction $\mathcal{A}$, 39

$reno_S$ Une fonction de renommage pour le squelette S, 36

$root((E,R))$ Racine de l'arbre (E,R), 17

Σ Ensemble des symboles de fonction d'un programme, 24

$sq(p)$ Squelette de racine indéfini enraciné par le symbole de prédicat de programme p, 35

$sq(u)$ Squelette enraciné par le nom de clause u dont les fils de sa racine sont indéfinis, 35

$\mathrm{SS}(P)$ Ensemble succès du programme P (succès positifs), 55

$\mathrm{SS}_\vee(P)$ Ensemble $\vee$-succès du programme P (succès négatifs), 55

STORE Ensemble des stores du langage, 26

$store(R)$ Store de la clause R, 27

$success(a)$ Ensemble des réponses pour le but $\leftarrow a$, 50

$T \uparrow \alpha$ Puissance ordinal ascendante α de T

$T \downarrow \alpha$ Puissance ordinal descendante α de T

$undef(S)$ Ensemble des nœuds indéfinis du squelette S, 35

V Ensemble des variables d'un programmes, 24

$var(E)$ Ensemble des variables libres de E, 24

ω Plus petit ordinal limite (> 0)

$\tilde{x}$ Ensemble (en général fini) de variables $\{x_1, \ldots, x_i, \ldots\}$

Index

Adresse postale : Alexandre Tessier, LIFO, Faculté des Sciences, Université d'Orléans, BP 6759, 45067 Orléans Cedex 2, France
Adresse électronique : `Alexandre.Tessier@lifo.univ-orleans.fr`
WWW : `http://www.univ-orleans.fr/SCIENCES/LIFO/tessier/`
FTP : `ftp-lifo.univ-orleans.fr` directory `/pub/Users/tessier/`
Téléphone : +33 2 38 49 46 70 — *Fax* : +33 2 38 41 71 37 — *Telex* : 783 388 F

Approche, en termes de squelettes de preuve, de la sémantique et du diagnostic déclaratif d'erreur des programmes logiques avec contraintes

Résumé

Cette thèse propose une reformulation complète de la sémantique des programmes logiques avec contraintes dans la lignée de la vision grammaticale de la programmation logique [P. Deransart et J. Maluszynski 1993]. La sémantique généralise les sémantiques classiques [J. Jaffar et J-L. Lassez 1987] basées sur une interprétation ou une théorie pour la sémantique du langage des contraintes. De plus, elle tient compte de l'incomplétude des solveurs de contraintes. Les résultats connus sont retrouvés. Cette reformulation est particulièrement bien adaptée pour étudier le diagnostic déclaratif d'erreur [E. Y. Shapiro 1982] des programmes logiques avec contraintes. Diverses notions d'erreurs sont définies et des algorithmes de diagnostics sont proposés. Ils sont comparés avec certaines techniques élaborées pour la programmation logique pure.

Mots Clés

Programmation logique avec contraintes, sémantique opérationnelle, sémantique déclarative, squelettes, arbre SLD, définitions inductives, arbres de preuve, diagnostic déclaratif d'erreur, correction partielle, débogage, validation.

Abstract

This thesis proposes a full reformulation of the Constraint Logic Program semantics following the Grammatical View of Logic Programming [P. Deransart et J. Maluszynski 1993]. Our semantics genralizes classical semantics [J. Jaffar et J-L. Lassez 1987] based on an interpretation or a theory for the contraint language semantics. Moreover, it takes into account incompleteness of the contraint solvers. Known results are revisited. This reformulation is particularly adapted to study Declarative Error Diagnosis [E. Y. Shapiro 1982] of Constraint Logic Programs. Several notions of errors are defined and diagnosis algorithms are proposed. They are compared with some technics developped for pure Logic Programming.

Key Words

Constraint logic programming, operational semantics, declarative semantics, skeleton, SLD tree, inductive definitions, proof trees, declarative error diagnosis, partial correctness, debugging, validation.

Thèse du Laboratoire d'Informatique Fondamentale d'Orléans présentée par Alexandre Tessier.