Diagnostic déclaratif d'insuffisance en programmation logique avec contraintes

Alexandre Tessier

LIFO, Université d'Orléans, BP 6759
45067 Orléans Cedex 2 (FRANCE)

Résumé : *Il est reconnu que la mise au point des programmes prend une part essentielle du temps dans l'activité du programmeur. Le débogage des programmes logiques avec contraintes est relativement inexploré. Or, pour assurer une large diffusion de CLP auprès des développeurs d'applications, les systèmes doivent intégrer des outils de mise au point.*

De nombreuses techniques de diagnostic déclaratif d'erreur ont été développées pour la programmation logique classique [15] ; mais, il n'est pas possible de les adapter simplement à la programmation logique avec contraintes. De nouveaux fondements théoriques sont nécessaires. Il n'y a plus comme en programmation logique couverture d'une réponse par une réponse calculée plus générale. La relation de couverture pallie cette absence. Toute réponse est couverte par un ensemble (éventuellement infini) de réponses calculées.

Cet article propose une étude du diagnostic déclaratif d'erreur en CLP et plus particulièrement des problèmes liés à l'insuffisance.

On observe que, par rapport aux travaux théoriques sur CLP [11], les implantations pratiques utilisent des solveurs incomplets. Les réponses insatisfiables ne sont pas nécessairement éliminées. Pour des raisons pratiques, il est important d'en tenir compte.

L'article redéfinit la sémantique des programmes logiques avec contraintes en terme d'arbres de preuve utilisant une relation de couverture. [14] montre l'intérêt d'un cadre fondé sur une relation abstraite de conséquence ("entailment" en anglais) plutôt que sur un domaine ou une théorie. La relation de couverture généralise cette relation de conséquence.

Les arbres de preuve permettent de donner à la notion de réponse une définition intrinsèque indépendante de toute règle de calcul. La relation de couverture permet d'exprimer l'idée qu'une contrainte est couverte par un ensemble (éventuellement infini) de contraintes.

La nature inductive de la sémantique est particulièrement bien adaptée à l'étude du diagnostic déclaratif d'erreurs.

Mots-clés : *Diagnostic déclaratif d'erreur, sémantique, programmation logique, contraintes, relation de couverture, vision grammaticale.*

1 Introduction

Le diagnostic déclaratif d'erreurs en programmation logique fut introduit par E. Y. Shapiro [15] sous le nom de *mise au point algorithmique*. D'autres techniques, développées dans [8, 7, 5] se sont inspirées de la méthode de Shapiro. Cet article est motivé par l'extension des concepts de mise au point algorithmique à la programmation logique avec contraintes.

Le diagnostic est déclaratif, ce qui signifie qu'il n'est pas utile pour le programmeur de comprendre le comportement opérationnel du système. En effet, une des grandes forces de la programmation logique avec contraintes est sa nature déclarative. Pour un langage déclaratif (dont la sémantique est indépendante de son modèle d'exécution), il est indispensable de considérer une notion d'erreur déclarative. Il serait incohérent de n'utiliser que des outils de mise au point de bas niveau, basés sur une compréhension du comportement opérationnel du système, alors qu'on donnerait un haut niveau de connaissance du sens déclaratif d'un programme. Le succès d'un outil de mise au point déclarative est directement lié au niveau de déclarativité du langage. De ce point de vue, la programmation logique avec contrainte est bien plus déclarative que la programmation logique classique en rendant inutile l'utilisation du "cut", de la négation par l'échec, du "is", des méta-prédicats tels que "var"... grâce à la notion de contrainte globale et en particulier grâce aux diséquations. Pour ces raisons, un composant essentiel d'un système CLP complet est un outil de mise au point déclarative.

Il est évident qu'un ordinateur ne peut diagnostiquer des erreurs dans un programme sans que lui soit indiqué au moins une partie de ce que devrait calculer une version sans erreur de ce programme. Mais, seule la sémantique déclarative attendue du programme est requise.

En programmation logique avec contraintes les interprétations de Herbrand ne représentent plus la sémantique des programmes. Lors de la mise au point on souhaite rester dans le langage du programme, mais certains éléments du domaine ne sont pas exprimables de façon finie dans celui-ci. Par exemple, le nombre π en CLP($\mathcal{R}$).

En programmation logique, chaque réponse qui est une conséquence du programme est couverte par *une* réponse calculée plus générale. Il n'y a plus couverture unique en programmation logique avec contraintes. Par exemple, en CLP($\mathcal{R}$), le programme : $\{p(x) \leftarrow x < 0\,,\, p(x) \leftarrow x \geq 0\}$ a pour conséquence $true \rightarrow p(x)$, pourtant il n'existe pas de réponse calculée la plus générale. Néanmoins, lors du diagnostic déclaratif d'erreur, on ne peut pas considérer que $true \rightarrow p(x)$ est un symptôme d'insuffisance car cette réponse (au sens déclaratif) est couverte par les deux réponses calculées $x < 0 \rightarrow p(x)$ et $x \geq 0 \rightarrow p(x)$. Une condition suffisante pour la propriété de couverture unique est l'indépendance des contraintes négatives (INC) [3] qui n'est malheureusement vérifiée que par peu de domaines de contraintes intéressants. Parfois, en programmation logique avec contraintes, il n'y a même pas couverture finie si la sémantique des programmes est définie non pas par une théorie mais par un modèle. Par exemple, en CLP($\mathcal{N}$), le programme : $\{ent(x) \leftarrow x = 0\,,\, ent(x) \leftarrow x = y + 1 \;\square\; ent(y)\}$ a pour conséquence $true \rightarrow ent(x)$. L'ensemble des réponses au but $\leftarrow ent(x)$ est $C = \{x = 0, x = 1, \ldots, x = i, \ldots\}$. Pour toute valuation dans

$\mathbb{N}$ qui satisfait $true$ il existe une contrainte de C satisfaite par cette valuation ($true$ est couverte par C), mais il n'existe pas de partie finie C_f de C telle que $true$ soit couverte par C_f.

Les algorithmes de mise au point déclarative habituellement proposés en programmation logique utilisent fortement les propriétés des modèles de Herbrand. Les méthodes classiques ne peuvent donc pas être adaptées simplement à la programmation logique avec contraintes.

Nous commençons par reformuler toutes les bases de la sémantique déclarative des programmes.

L'approche de la sémantique des programmes logiques avec contraintes en terme d'arbres de preuve et de squelettes est une extension de la vision grammaticale de la programmation logique [6]. La relation entre la sémantique déclarative et la sémantique opérationnelle est ainsi mieux expliquée. Cette sémantique est paramétrée par un *critère de rejet* [3] dont le rôle est d'éliminer certaines réponses dont la contrainte n'est jamais satisfaite. La notion de domaine ou de théorie, habituelle pour la sémantique des programmes logiques avec contraintes, devient un paramètre du critère de rejet. Cette sémantique permet de rendre compte de l'incomplétude de certains solveurs de contraintes qui ne peut pas s'exprimer en terme de logique. Par exemple, le solveur de contraintes de CLP($\mathcal{R}$) fournit trois types de réponses : `yes` (la contrainte est satisfiable), `no` (la contrainte est insatisfiable) et `maybe` (il ne peut pas statuer). Il répond `yes` pour $x \times x = 1 \wedge x = 1$ et `maybe` pour $x \times x = 1$, néanmoins $\models \exists x(x \times x = 1 \wedge x = 1) \rightarrow \exists x(x \times x = 1)$; il répond `no` pour $x = 1 \wedge x = 0$ et `maybe` pour $x \times x = 1 \wedge x \times x = 0$, néanmoins $\models \neg\exists x(x = 1 \wedge x = 0) \rightarrow \neg\exists x(x \times x = 1 \wedge x \times x = 0)$.

Le diagnostic déclaratif d'incorrection a pu être traité dans ce cadre [13] mais l'abstraction introduite par le critère de rejet ne permet pas d'exprimer la notion de couverture des réponses par des réponses calculées qui est nécessaire pour le diagnostic d'insuffisance. Pour cette raison nous introduisons une *relation de couverture* entre une contrainte c et un ensemble de contraintes C , noté $c \vdash C$ et lue : c est couverte par C. On retrouve le critère de rejet quand C est vide et la relation de conséquence [14] quand C est un singleton. Nous généralisons ces deux notions et retrouvons les résultats connus dans le cadre de la programmation logique avec contraintes.

La contribution principale de cet article est d'étendre les définitions de symptômes et d'erreurs d'insuffisance de la programmation logique dans un cadre uniforme qui tient compte de l'incomplétude des solveurs de contraintes (les problèmes liés à l'incorrection sont traités dans [13]). Il contribue également à mieux comprendre et comparer les deux familles d'algorithmes existantes en programmation logique [8, 7].

Le reste de cet article est organisé comme suit : La section 2 définit le langage des programmes et quelques notations. La section 3 définit la sémantique opérationnelle, en termes de squelettes, fondée sur le critère de rejet. La section 4 définit la sémantique déclarative, elle présente la relation de couverture et ses propriétés. La nature inductive de la sémantique déclarative est particulièrement adaptée à l'étude du diagnostic déclaratif d'erreur qui fait l'objet de la section 5. La conclusion récapitule les idées principales et les perspectives.

Les preuves se trouvent en détail dans [16].

2 Terminologie et notations

Considérons une fois pour toute quatre ensembles qui déterminent le langage du programme : un ensemble infini de *variables* V ; un ensemble de *symboles de fonctions* Σ ; un ensemble de *symboles de prédicats de contraintes* Π_c ; un ensemble de *symboles de prédicats de programme* Π_p.

Les formules atomiques construites sur V et Π_p de la forme $p(x_1, \ldots, x_n)$ (p : prédicat de programme d'arité n, $x_1, \ldots, x_n$: n variables distinctes) sont appelées *atomes*. Le langage des contraintes CONST est un sous-ensemble du langage du premier ordre construit sur V, Σ et Π_c. Nous supposons qu'il est clos par conjonction et quantification existentielle. Une *contrainte* est une formule de CONST.

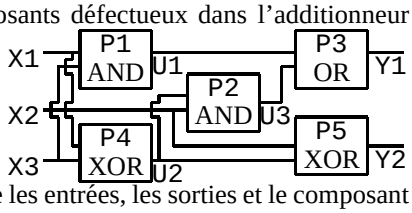
Notations $\tilde{x}$ désigne une séquence de variables distinctes $x_1, \ldots, x_n$. Si F est une formule (construite sur V, Σ, $\Pi_p \cup \Pi_c$) alors $var(F)$ est la séquence des variables libres de F. Si c est une contrainte, $\tilde{x}$ est $x_1, \ldots, x_n$, $\tilde{y}$ sont les variables libres de c n'appartenant pas à $\tilde{x}$ et a est un atome alors $\exists_{\tilde{x}}\, c$ désigne $\exists x_1 \cdots \exists x_n\, c$, $\exists_{-\tilde{x}}\, c$ désigne $\exists_{\tilde{y}}\, c$, $\exists_{-a}\, c$ désigne $\exists_{-var(a)}\, c$.

Une *clause* est un $n+2$-uplet ($0 \leq n$) noté $a_0 \leftarrow c \,\square\, a_1, \ldots, a_n$, où les a_i sont des atomes et c est une contrainte. Un *but* est une clause sans tête de la forme $\leftarrow c \,\square\, a_1, \ldots, a_n$. Étant donné une clause R de la forme précédente, nous notons $head(R) = a_0$, $body(R) = c \,\square\, a_1, \ldots, a_n$, $constraint(R) = c$ et $arity(R) = n$. Un *programme* est un ensemble de clauses. Un *atome contraint* est un couple $a[c]$ où a est un atome et c est une contrainte.

Un but $\leftarrow g$ définit une nouvelle relation, noté ans_g ($ans_g \notin \Pi_p \cup \Pi_c$), d'arité le nombre de variables libres de g. Soit P_{ans} le programme $P \cup \{ans_g(var(g)) \leftarrow g \mid \leftarrow g$ est un but pour $P\}$. Le langage de P détermine le langage de P_{ans} et la donnée de P détermine P_{ans}. Les prédicats ajoutés n'ont aucune occurrence dans les corps de clause de P_{ans}. Ainsi, nous pouvons examiner les réponses à des questions atomiques pour P_{ans} ce qui simplifie les définitions par la suite. Une réponse au but $\leftarrow g$ pour P est une réponse à la question $ans_g(var(g))$ pour P_{ans}.

L'ensemble de cet article est illustré par l'exemple, ci-dessous, de détection de pannes dans un additionneur binaire, dû à A. Colmerauer [4], en Prolog III.

Exemple 1 On cherche à détecter des composants défectueux dans l'additionneur binaire ci-contre qui calcule la somme de trois bits `X1`,`X2`,`X3` sous forme d'un nombre binaire sur deux bits `Y1 Y2`. On ne s'intéresse, dans le programme DETECT1 ci-dessous, qu'au cas où un seul des composants est en panne.



Le prédicat `circuit` exprime la relation entre les entrées, les sorties et le composant défectueux. `C` est la liste des composants (`P1` à `P5`), un composant est défectueux si sa valeur est `1'`, `E` est la liste des entrées (`X1` à `X3`) et `S` est la liste des sorties (`Y1` et `Y2`). `U1`,`U2`,`U3` sont les sorties des composants `P1`,`P2`,`P4`.

```
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]}.
```

Le but ci-dessous fournit les entrées quand P5 est défectueux et la sortie est [1',1'].

```
?- circuit(C,E,S), enumere(E) {C=[0',0',0',0',1'],
   S=[1',1']}.
{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']}
```

Les première et troisième réponses s'expliquent par le fait qu'un composant défectueux n'a pas toujours sa sortie erronée (implications dans le programme).
La relation définie par ce but dans DETECT1$_{ans}$ est appelée go et définie par :

```
go(C,E,S) :- circuit(C,E,S), enumere(E)
  {C=[0',0',0',0',1'],S=[1',1']}.
```

3 Sémantique opérationnelle

Cette section décrit la sémantique opérationnelle d'un programme. Elle se termine par la définition d'arbre SLD utile à l'étude du diagnostic déclaratif d'insuffisance pour la définition de symptôme calculé.

Nous commençons par introduire la notion de squelette qui fait le lien entre la sémantique opérationnelle et la sémantique déclarative. Les squelettes permettent, d'une part, de décrire la SLD-résolution et d'autre part de donner une définition intrinsèque à la notion de réponse. Ainsi de nombreux résultats d'indépendance de la règle de calcul peuvent être montrés simplement [3].

3.1 Squelettes

Un *squelette* est un arbre orienté, étiqueté par $P_{ans} \cup \{?\}$, tel que, pour tout nœud N, $label(N)$ étant son étiquette : le degré de N est $arity(label(N))$ ($arity(?) = 0$) ; si $label(N) = a_0 \leftarrow c \,\square\, a_1, \ldots, a_n$ alors N a n fils $N_1, \ldots, N_n$ tels que, pour tout $i = 1, \ldots, n$, $label(N_i)$ est soit ? soit une clause R_i telle que $head(R_i)$ et a_i ont le même symbole de prédicat. On note, pour tout squelette S, $undef(S)$ l'ensemble des nœuds de S étiqueté par ?, $def(S)$ l'ensemble des autres nœuds et $root(S)$ sa racine. Notons que les clauses de $P_{ans} - P$ n'étiquettent que les racines des squelettes.

Une *fonction de renommage* pour un squelette S, est une fonction $choice$ (des nœuds vers $P_{ans} \cup \{?\}$) telle que, pour tout nœud N de S : si $N = root(S)$ ou $label(N)$=? alors $choice(N)$=$label(N)$; sinon $choice(N)$=$label(N)\theta$, où θ est un renommage tel que : si N est le i^{ème} fils de N' alors le i^{ème} atome de $body(choice(N'))$

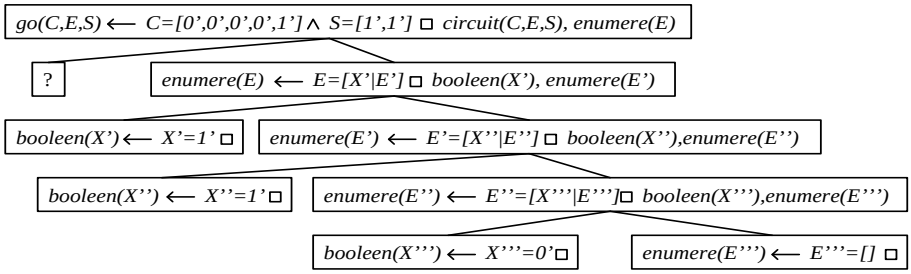
est $head(choice(N))$ et, pour tout nœud N' qui n'appartient pas au sous-arbre de racine N, toute variable de $var(body(choice(N))) - var(head(choice(N)))$ n'a pas d'occurrence libre dans $choice(N')$.

Nous associons à tout squelette S et à toute fonction de renommage $choice$ pour S le *système de contraintes* $const(S, choice) = \{constraint(choice(N)) \mid N \in def(S)\}$. La fonction de renommage ne détermine que le renommage appliqué aux variables du système de contraintes associé au squelette S. Par abus, nous notons parfois ce système $const(S)$. Il est défini à un renommage près.

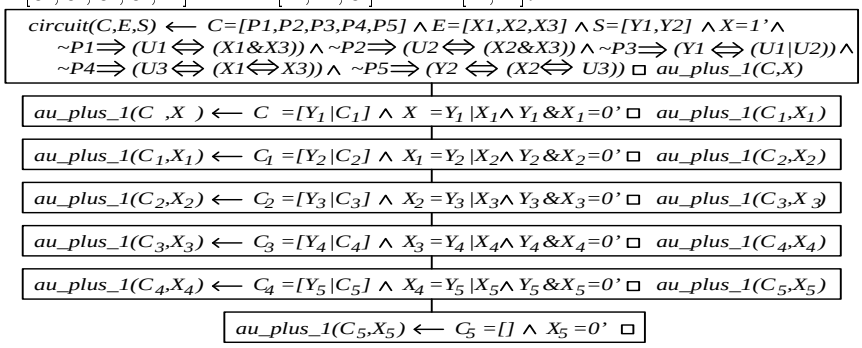
Quand $def(S)$ est fini, on note $conj(S, choice)$ la conjonction des contraintes de $const(S, choice)$ et $AC(S)$ la contrainte $\exists_{-\tilde{x}}\, conj(S, choice)$, où $\tilde{x}$ sont les variables de $head(choice(root(S)))$. $AC(S)$ ne dépend pas de la fonction de renommage mais seulement de $var(head(label(root(S))))$. Elle est appelé la *contrainte associée* à S.

Un *squelette complet* est un squelette S tel que $undef(S) = \emptyset$. Les squelettes finis complets définissent les réponses générales à un programme en ce sens que si S est un squelette fini complet de racine $ans_g(var(g)) \leftarrow g$, alors $P \models AC(S) \rightarrow g$.

Exemple 2 Les clauses du squelette sont renommées pour faciliter la compréhension.



Le système de contrainte est $\{C = [0', 0', 0', 0', 1'] \wedge S = [1', 1'], E = [X'|E'], X' = 1', E' = [X''|E''], X'' = 1', E'' = [X'''|E'''], X''' = 0', E''' = []\}$, sa contrainte associée est $\exists X' \exists E' \exists X'' \exists E'' \exists X''' \exists E''' \, (C = [0', 0', 0', 0', 1'] \wedge S = [1', 1'] \wedge E = [X'|E'] \wedge X' = 1' \wedge E' = [X''|E''] \wedge X'' = 1' \wedge E'' = [X'''|E'''] \wedge X''' = 0' \wedge E''' = [])$ simplifiée par Prolog III en $C = [0', 0', 0', 0', 1'] \wedge S = [1', 1'] \wedge E = [1', 1', 0']$.
Si l'on greffe le squelette suivant à la feuille indéfinie du squelette précédent on obtient un squelette fini complet dont la contrainte associée simplifié par Prolog III est $C = [0', 0', 0', 0', 1'] \wedge E = [1', 1', 0'] \wedge S = [1', 1']$.



3.2 Critère de rejet, réponses, arbre SLD

Parmi les réponses générales, il en existe qui présentent peu d'intérêt. Si $AC(S)$ est toujours fausse alors $AC(S) \rightarrow g$ est toujours vraie indépendamment du programme. Le système cherche à éliminer ces "réponses triviales" grâce à un critère de rejet.

Un *critère de rejet*, noté $\vdash \emptyset$, est une relation sur CONST. Il permet de rendre compte du fonctionnement d'un interprète qui fourni des réponses assorties du commentaire *peut-être*, signalant ainsi que l'algorithme (incomplet) de satisfaction de contraintes ne répond ni *oui* ni *non*. Le critère de rejet abstrait l'interprétation des contraintes. Nous supposons pour simplifier, et ce jusqu'à la fin de cet article, que si c_1 et c_2 sont deux contraintes alors : $c_1 \wedge c_2$ et $c_2 \wedge c_1$ sont identiques ; si $x_1, \ldots, x_n$ sont n variables libres de c_1 mais pas de c_2 alors $\exists x_1 \cdots \exists x_n\, c_1 \wedge c_2$ et $\exists x_1 \cdots \exists x_n\, (c_1 \wedge c_2)$ sont identiques ; si x et y sont deux variables alors $\exists x \exists y\, c_1$ et $\exists y \exists x\, c_1$ sont identiques et $\exists x\, c_1$ et $\exists x \exists x\, c_1$ sont identiques.

Pour toute contrainte c rejetée, noté $c \vdash \emptyset$, $\vdash \emptyset$ a les trois propriétés suivantes : **CONJ** : pour toute contrainte c', $c \wedge c' \vdash \emptyset$; **EXIS** : pour toute variable x, $\exists x\, c \vdash \emptyset$; **RENO** : pour tout renommage θ, $c\theta \vdash \emptyset$.

Le critère de rejet n'est pas quelconque. Il sera le plus souvent défini à partir : soit d'un domaine $\mathcal{D}$, et notée $\vdash_{\mathcal{D}} \emptyset$, dans ce cas $c \vdash_{\mathcal{D}} \emptyset$ si pour toute valuation v dans $\mathcal{D}$ $v(c)$ est fausse ; soit d'une théorie $\mathcal{T}$, et notée $\vdash_{\mathcal{T}} \emptyset$, $c \vdash_{\mathcal{T}} \emptyset$ si pour tout modèle $\mathcal{D}$ de $\mathcal{T}$ $c \vdash_{\mathcal{D}} \emptyset$ (c'est-à-dire $\mathcal{T} \models \neg c$) ; soit d'un algorithme de satisfiabilité de contrainte $\mathcal{A}$, et notée $\vdash_{\mathcal{A}} \emptyset$, $c \vdash_{\mathcal{A}} \emptyset$ si l'algorithme répond non pour la satisfiabilité de c.

Définition 1 *Un squelette S est* rejeté *s'il existe une partie finie C de $const(S)$ telle que $\bigwedge_{c \in C} c \vdash \emptyset$ (remarquons que S fini est rejeté si et seulement si $AC(S) \vdash \emptyset$). Une* réponse *pour le but $\leftarrow g$ est un squelette fini complet S non rejeté enraciné par $ans_g(var(g)) \leftarrow g$. $AC(S)$ est alors une* contrainte réponse *pour $\leftarrow g$.*

Exemple 3 Le squelette fini complet de l'exemple 2 est une réponse pour le but $\leftarrow C = [0', 0', 0', 0', 1'] \wedge S = [1', 1'] \,\square\, circuit(C, E, S), enumere(E)$, le système de contrainte associé n'est pas rejeté. La contrainte réponse est simplifiée par Prolog III en $C = [0', 0', 0', 0', 1'] \wedge S = [1', 1'] \wedge E = [1', 1', 0']$.

Un squelette S' *dérive* d'un squelette S non rejeté par la feuille $N \in undef(S)$ si S' n'est pas rejeté et il existe $R \in P_{ans}$ tel que S' est obtenu à partir de S en greffant $sq(R)$ en N, où $sq(R)$ est le squelette dont la racine étiquetée par R a $arity(R)$ fils étiquetés par ?.

Définition 2 *Un* arbre SLD *pour le but $\leftarrow g$ est un arbre A de racine $sq(ans_g(var(g)) \leftarrow g)$, dont les nœuds sont étiquetés par des squelettes finis non rejetés, tel que : un nœud étiqueté par une réponse S est un nœud* succès *et S est une* réponse calculée *; pour tout nœud M de A étiqueté par S incomplet, soit $N \in undef(S)$, les fils de M correspondent exactement aux squelettes qui dérivent de S par N et sont étiquetés par ces squelettes, si M n'a pas de fils c'est un nœud* échec.

Théorème 3 *Les réponses calculées pour le but $\leftarrow g$ sont exactement les réponses pour le but $\leftarrow g$.*

Nous appelons *règle de calcul* une fonction r qui, pour tout squelette incomplet S donne une feuille de $undef(S)$. La *stratégie standard* correspond à la règle de calcul qui choisie la feuille indéfinie la plus à gauche. L'arbre SLD pour le but $\leftarrow g$ selon la règle de calcul r est l'arbre SLD dans lequel, en tout nœud étiqueté par un squelette incomplet S, la feuille indéfinie choisie est $r(S)$.

Théorème 4 *Les réponses calculées sont indépendantes de la règle de calcul.*

4 Sémantique déclarative

En programmation logique, une substitution réponse au sens déclaratif pour le but $\leftarrow g$ est une substitution θ telle que $P \models g\theta$. Pour toute substitution réponse, il existe *une* substitution réponse calculée plus générale. Ceci devrait se traduire en programmation logique avec contraintes par le fait que pour toute contrainte c telle que $P \models c \rightarrow g$ (ou $P, \mathcal{T} \models c \rightarrow g$ ou encore $P \models_{\mathcal{D}} c \rightarrow g$), il existe *une* contrainte réponse calculée c' telle que c' est "plus générale" que c. Nous savons qu'en générale ceci n'est pas le cas. Néanmoins nous voulons conserver une propriété semblable qui relie les réponses au sens déclaratif et les réponses au sens opérationnel. En fait, toute réponse est couverte par un ensemble (éventuellement infini) de réponses calculées. Cette relation est appelée relation de couverture et étend le critère de rejet introduit dans la section 3.

La sémantique déclarative proposée n'est pas, comme il est usuel, de nature logique. L'abstraction de l'interprétation des contraintes par la relation de couverture ne le permet pas. C'est une sémantique de type point fixe. Malgré tout, on retrouve l'ensemble des résultats connus dans le cas ou la relation de couverture est déduite d'un domaine (ou d'une théorie)

4.1 Relation de couverture

Une *relation de couverture* $\vdash$ est une relation binaire sur CONST $\times$ 2^{CONST}. Elle a, pour toute contrainte c, les cinq propriétés suivantes :
CONJ : s'il existe n contraintes $c_1, \ldots, c_n$ et n ensembles de contraintes $C_1, \ldots, C_n$ tels que $c_i \vdash C_i$, pour tout $i = 1, \ldots, n$, alors $c \wedge c_1 \wedge \cdots \wedge c_n \vdash \{c \wedge c'_1 \wedge \cdots \wedge c'_n \mid c'_i \in C_i, \text{ pour tout } i = 1, \ldots, n\}$ (si $n = 0$ on en déduit $c \vdash \{c\}$) ;
EXIS : s'il existe un ensemble de contraintes C tel que $c \vdash C$ alors $\exists_{\tilde{x}}\, c \vdash \{\exists_{\tilde{x}}\, c' \mid c' \in C\}$, pour toute séquence de variables $\tilde{x}$;
TRAN : s'il existe un ensemble de contraintes C tel que $c \vdash C$ et s'il existe une famille d'ensembles de contraintes $(C'_{c'})_{c' \in C}$ telle que $c' \vdash C'_{c'}$, pour toute contrainte $c' \in C$, alors $c \vdash \bigcup_{c' \in C} C'_{c'}$;
MONO : s'il existe un ensemble de contraintes C tel que $c \vdash C$ alors pour tout ensemble de contraintes C', tel que $C \subseteq C'$, $c \vdash C'$;
RENO : si θ est un renommage et si $c \vdash C$ alors $c\theta \vdash \{c'\theta \mid c' \in C\}$.

La relation de couverture étend le critère de rejet. Une contrainte est rejetée si elle est couverte par l'ensemble vide. On retrouve les trois propriétés (**CONJ**, **EXIS**, **RENO**) du critère de rejet. $(\text{CONST}, \vdash)$ définit ainsi un système de contraintes en un sens analogue à [14, 10].

La relation de couverture n'est, en général, pas quelconque. Elle sera le plus souvent définie à partir : soit d'un domaine $\mathcal{D}$, elle est notée $\vdash_{\mathcal{D}}$, $c \vdash_{\mathcal{D}} C$ si pour toute valuation dans $\mathcal{D}$ qui satisfait c il existe une contrainte de C satisfaite par cette valuation ; soit d'une théorie $\mathcal{T}$, elle est notée $\vdash_{\mathcal{T}}$, $c \vdash_{\mathcal{T}} C$ si pour tout modèle $\mathcal{D}$ de $\mathcal{T}$ $c \vdash_{\mathcal{D}} C$; soit d'un algorithme de satisfiabilité de contrainte $\mathcal{A}$, elle est notée $\vdash_{\mathcal{A}}$ et définie par la plus petite relation vérifiant les cinq propriétés et telle que $c \vdash_{\mathcal{A}} \emptyset$ si l'algorithme répond non pour la satisfiabilité de c.

4.2 Arbres de preuve

La sémantique déclarative est définie inductivement par des règles. Ces règles sont de deux types. Les premières sont issues des clauses du programme et les secondes sont définies par la relation de couverture.

Définition 5 *Pour toute clause renommée $a \leftarrow c \square a_1, \ldots, a_n$ du programme P_{ans} et toutes contraintes $c_1, \ldots, c_n$, nous avons la règle de programme :*

$$\frac{a_1[c_1] \ \cdots \ a_n[c_n]}{a[\exists_{-a}\, c \wedge c_1 \wedge \cdots \wedge c_n]} \qquad \textit{(si } n = 0 \textit{ on déduit l'axiome } \frac{}{a[\exists_{-a}\, c]}\textit{)}$$

Pour tout atome a (du langage augmenté), toute contrainte c et tout ensemble de contraintes C tels que $c \vdash C$, nous avons la règle de couverture :

$$\frac{\{a[c'] \mid c' \in C\}}{a[c]} \qquad \textit{(si } C = \emptyset \textit{ on déduit l'axiome } \frac{}{a[c]}\textit{)}$$

Comme à tout système de règle, il est associé, de manière classique, une notion d'*arbre de preuve* [1]. Les arbres de preuve avec hypothèses sont appelés *arbres de preuve partiels*, les autres sont appelés *arbres de preuve complets*.

Nous distinguons les arbres de preuve utilisant les deux types de règles, qui déterminent la sémantique déclarative, des arbres de preuves utilisant seulement les règles de programmes, qui déterminent la sémantique opérationnelle théorique.

S'il existe un arbre de preuve (partiel ou complet) dont la conclusion est l'atome contraint $a[c]$ et les hypothèses constituent l'ensemble d'atomes contraints A alors nous notons en résumé $A \leadsto_{\vdash} a[c]$. Si de plus cet arbre de preuve n'utilise que des règles de programme alors nous notons en résumé $A \leadsto a[c]$.

Si $\emptyset \leadsto_{\vdash} a[c]$ alors $a[c]$ est appelé une *réponse*[1] à la question a selon $\vdash$. Si $\emptyset \leadsto a[c]$ alors $a[c]$ est appelé une *réponse calculée* à la question a. Remarquons que les atomes contraints de la forme $ans_g(var(g))[c]$ ne peuvent étiqueter que les racines des arbres de preuve n'utilisant que des règles de programme.

Remarque 1 si $c \vdash \emptyset$ alors $\emptyset \leadsto_{\vdash} a[c]$, pour tout atome a. $a[c]$ est alors appelé une *réponse triviale* à la question a selon $\vdash$. Comme nous l'avons précisé dans la section 3, le système cherchera, dans la mesure du possible, à éliminer ces réponses triviales qui sont indépendantes du programme et sans intérêt pour l'utilisateur. Par exemple,

[1] Afin d'éviter la confusion avec les définitions précédentes, on appelle maintenant *squelette réponse* une réponse définie dans la section 3

en CLP($\mathcal{R}$), soit le programme : $\{p(x,y) \leftarrow x > y \ \square \ , \ q(x,y) \leftarrow v = x \times x \wedge w = y \times y \ \square \ p(v,w) \ , \ r(x) \leftarrow x = y \ \square \ p(x,y) \ , \ r(x) \leftarrow x = y \ \square \ q(x,y)\}$ Pour la question $r(x)$ nous obtenons par exemple les réponses $r(x)[x > x]$ et $r(x)[x \times x > x \times x]$. Relativement au domaine $\mathcal{R}$, les deux contraintes de ces réponses sont insatisfiables. Le système CLP($\mathcal{R}$) élimine bien la première, par contre il n'élimine pas la seconde mais la qualifie de "`maybe`" (le solveur est incomplet).

Exemple 4 $go(C,E,S)[C = [0',0',0',0',1'] \wedge S = [1',1'] \wedge E = [0',1',1']]$ est une réponse calculée pour la question $go(C,E,S)$ dont voici une preuve, notée $\boxed{A}$ (les contraintes sont simplifiées pour faciliter la lecture) :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{au\_plus\_1(C_5,X_5)[C_5=[] \wedge X_5=0']}}{au\_plus\_1(C_4,X_4)[C_4=[1'] \wedge X_4=1']}}{au\_plus\_1(C_3,X_3)[C_3=[0',1'] \wedge X_3=1']}}{au\_plus\_1(C_2,X_2)[C_2=[0',0',1'] \wedge X_2=1']} \quad \dfrac{}{booleen(X''')[X'''=1'] \ enumere(E''')[E'''=[]]}}{au\_plus\_1(C_1,X_1)[C_1=[0',0',0',1'] \wedge X_1=1'] \quad booleen(X'')[X''=1'] \quad enumere(E'')[E''=[1']]}}{au\_plus\_1(C\_,X\_)[C\_=[0',0',0',0',1'] \wedge X\_=1'] \quad enumere(E')[E'=[1',1']] \quad booleen(X')[X'=0']}}{circuit(C,E,S)\,[C=[0',0',0',0',1'] \wedge S=[1',1'] \wedge E=[0',1',1'] \quad enumere(E)[E=[0',1',1']]}}{go(C,E,S)[C=[0',0',0',0',1'] \wedge S=[1',1'] \wedge E=[0',1',1']}$$

Il existe une preuve similaire, notée $\boxed{B}$, de la réponse calculée $go(C,E,S)[C = [0',0',0',0',1'] \wedge S = [1',1'] \wedge E = [1',1',0']]$.

De la règle de couverture issue de $\exists A \, (E = [A, 1', \sim A]) \vdash \{E = [0',1',1'], E = [1',1',0']\}$ et de la propriété **CONJ** on déduit pour la réponse $go(C,E,S)[\exists A \, (C = [0',0',0',0',1'] \wedge S = [1',1'] \wedge E = [A,1',\sim A])]$ l'arbre de preuve :

$$\dfrac{\boxed{A} \qquad \boxed{B}}{go(C,E,S)[\ \exists A\ (C=[0',0',0',0',1'] \wedge\ S=[1',1'] \wedge\ E=[A,1',\sim A]\ )]}$$

Étant donné un squelette S et une fonction de renommage $choice$ pour S, nous notons $AP(S, choice)$ l'arbre qui a exactement les mêmes nœuds que $def(S)$ aux étiquettes près, et qui vérifie la propriété suivante : pour tout nœud N de $AP(S, choice)$, soit N' son correspondant dans S, $S_{N'}$ le sous-arbre de S de racine N' et $a = head(choice(N'))$, l'étiquette de N est $a[AC(S_{N'})]$.

Étant donné un arbre de preuve A n'utilisant que des règles de programme, nous notons $SQ(A)$ l'arbre qui a exactement les mêmes nœuds que A aux étiquettes près, et qui vérifie la propriété suivante : pour tout nœud N de $SQ(A)$, soit N' son correspondant dans A, si N' est une hypothèse alors N est étiqueté par ?, sinon l'étiquette de N est la clause d'où est extraite la règle de programme qui relie N' à ses fils.

Lemme 6 *Si S est un squelette fini complet et $choice$ une fonction de renommage pour S alors $AP(S, choice)$ est un arbre de preuve complet (n'utilisant que des règles de programme).*

Lemme 7 *Si A est un arbre de preuve complet de racine $a[c]$ n'utilisant que des règles de programme alors $SQ(A)$ est un squelette fini complet et $AC(SQ(A)) = c$.*

Théorème 8 *$a[c]$ est une réponse calculée pour la question a si et seulement si il existe un squelette fini complet S tels que $head(root(S)) = a$ et $AC(S) = c$.*

Corollaire 9 *$ans_g(var(g))[c]$ est une réponse calculée à la question $ans_g(var(g))$ si et seulement si c est la contrainte associée à un squelette réponse pour le but $\leftarrow g$. c est une contrainte réponse pour le but $\leftarrow g$ si et seulement si $ans_g(var(g))[c]$ est une réponse calculée pour la question $ans_g(var(g))$ et non($c \vdash \emptyset$).*

Le lien entre les arbres de preuves et les squelettes étant clairement établi (à un squelette fini correspond une classe d'équivalence d'arbres de preuve) nous donnons la sémantique déclarative d'un programme sous forme d'un ensemble d'atomes contraints appelé *ensemble succès*. Nous distinguons deux ensembles succès, le premier décrit la sémantique opérationnelle théorique et le second la sémantique déclarative.

Définition 10 *L'ensemble succès associé au programme P et à une relation de couverture $\vdash$ est $SS_\vdash(P) = \{a[c] \mid \emptyset \rightsquigarrow_\vdash a[c]\}$.*
L'ensemble succès associé au programme P est $SS(P) = \{a[c] \mid \emptyset \rightsquigarrow a[c]\}$.

$SS(P)$ (resp. $SS_\vdash(P)$) est l'ensemble défini inductivement par les règles de programme (resp. par les règles de programme et les règles de couverture).

Remarque 2 Si l'on considère le sous-ensemble $SS^{\vdash\emptyset}(P)$ de $SS(P)$ défini par $SS^{\vdash\emptyset}(P) = \{a[c] \mid a[c] \in SS(P)$ et non($c \vdash \emptyset$)$\}$ alors $SS^{\vdash\emptyset}(P) = SS_{RC}(P)$ de [3] si le critère de rejet RC est $\vdash \emptyset$, $SS^{\vdash_{\mathcal{T}}\emptyset}(P) = SS(P, \mathcal{T})$ de [11], $SS^{\vdash_{\mathcal{D}}\emptyset}(P) = lfp(S_P^{\mathcal{D}})$ de [12], $SS^{\vdash_{\mathcal{D}}\emptyset}(P) = SS_3(P, \mathcal{D})$ de [9].

Les ensembles succès peuvent se définir également comme plus petit point fixes d'opérateurs de conséquence immédiate associés aux deux types de règles.

Définition 11 *Soit T_P, $T_\vdash$, $T^{\cup}_{\vdash,P}$ et $T^{\circ}_{\vdash,P}$ les quatre opérateurs de conséquences immédiates (des ensembles d'atomes contraints dans les ensembles d'atomes contraints), associés aux règles de programmes et de couvertures, définis par :*

$$T_P(I) = \{a[c] \mid \text{il existe une règle de programme } \frac{a_1[c_1], \ldots, a_n[c_n]}{a[c]} \text{ telle que } a_i[c_i] \in I, \text{ pour tout } i = 1, \ldots, n\}$$

$$T_\vdash(I) = \{a[c] \mid \text{il existe une règle de couverture } \frac{\{a[c'] \mid c' \in C\}}{a[c]} \text{ telle que } a[c'] \in I, \text{ pour toute contrainte } c' \in C\}$$

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

Lemme 12 *$pppf(T_P) = T_P \uparrow \omega = SS(P)$ et $pppf(T^{\cup}_{\vdash,P}) = SS_\vdash(P)$.*

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 et le plus petit point fixe de $T^{\circ}_{\vdash,P}$ correspond aux arbres de preuves qui utilisent aux profondeurs impaires une règle de programme et aux profondeurs paires une règle de couverture. Le plus petit point fixe de $T_\vdash$ est sans intérêt ici.

Nous montrons, par la suite, que le plus petit point fixe de $T^{\circ}_{\vdash,P}$ est égal, d'une part, à $SS_{\vdash}(P)$ ce qui signifie que les arbres de preuve complets alternant règles de programme et règles de couverture fournissent toutes les réponses et, d'autre part à $T_{\vdash}(SS(P))$ ce qui signifie que les arbres de preuve complets n'utilisant qu'une règle de couverture et cette règle étant utilisée à leur racine fournissent toutes les réponses.

Lemme 13 $SS_{\vdash}(P) = \{a[c] \mid$ *il existe* C *tel que* $c \vdash C$, *et* $a[c'] \in SS(P)$, *pour toute contrainte* $c' \in C\}$

Corollaire 14 $pppf(T^{\cup}_{\vdash,P}) = T_{\vdash}(pppf(T_P)) = T_{\vdash}(SS(P)) = SS_{\vdash}(P)$
$pppf(T^{\cup}_{\vdash,P}) = T^{\cup}_{\vdash,P} \uparrow \omega + 1$
Pour tout atome contraint $a[c]$, *si* $\emptyset \rightsquigarrow_{\vdash} a[c]$ *alors il existe un ensemble de contraintes* C *tel que* $c \vdash C$ *et, pour toute contrainte* $c' \in C$, $\emptyset \rightsquigarrow a[c']$.

C'est-à-dire, pour tout arbre de preuve complet, il existe un arbre de preuve complet de même conclusion qui utilise exactement une règle de couverture, et cette règle de couverture est utilisée à la racine de l'arbre de preuve. Nous retrouvons ainsi les résultats bien connus dans le cas où $\vdash$ est déduite d'un domaine ou d'une théorie (notre définition de réponse est équivalente à celle de [17] quand $\vdash$ est déduite d'un domaine).

L'opérateur $T^{\circ}_{\vdash,P}$ est croissant et a donc un plus petit point fixe. Nous ne définirons pas ici le système de règles associé à cet opérateur, il se déduit facilement des compositions des règles de programmes et des règles de couverture.

Lemme 15 $pppf(T^{\circ}_{\vdash,P}) = pppf(T^{\cup}_{\vdash,P}) = T_{\vdash}(pppf(T_P))$

Ces égalités sont essentielles car elles assurent que les réponses définies à partir de couverture de réponses calculées, d'une part, et les réponses définis en appliquant alternativement règles de programme et règles de couverture, d'autre part, sont exactement toutes les réponses. L'égalité de ces trois ensembles assure la cohérence des définitions et résultats de la section 5.

5 Diagnostic déclaratif d'insuffisance

Cette section étend les travaux sur le diagnostic déclaratif d'erreur en programmation logique basé sur la méthode de Shapiro à la programmation en logique avec contraintes. Les réponses fournies par un programme sont parfois symptômes d'erreurs dans le programme. Le diagnostic d'erreur est la localisation de l'erreur à partir d'un symptôme (réponse fausse ou réponse manquante). Nous ne traitons ici que le cas des réponses manquantes (voir [13] pour le cas des réponses fausses).

Exemple 5 Afin d'illustrer cette section nous donnons le programme DETECT2 qui est une version erronée du programme DETECT1 de l'exemple 1. DETECT2 est identique à DETECT1 à l'exception de la définition de `circuit`:

```
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))}.
```

Nous nous intéresserons au même but. Les réponses fournies par Prolog III sont :

```
?- circuit(C,E,S), enumere(E) {C=[0',0',0',0',1'],
   S=[1',1']}.
{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']}
```

La relation définie par ce but, dans DETECT2$_{ans}$, est encore appelée `go`.

La sémantique attendue du programme est formalisée par un ensemble d'atomes contraints noté I_P.

L'objectif est de déterminer un algorithme qui assiste au mieux le programmeur dans la recherche d'une erreur dans le programme dans le cas ou il manque des réponses à une question. Nous souhaiterions faire appel à l'algorithme dans le cas suivant : les réponses à la question a sont $c_1, \ldots, c_m$, l'atome contraint $a[c]$ est attendu mais c n'est pas couverte par $c_1, \ldots, c_m$.

Nous dirons que le programme P est *finiment incomplet* relativement à I_P pour l'atome a s'il existe un arbre SLD fini pour la question a dont les succès constituent l'ensemble de contraintes C et il existe une contrainte c telle que $a[c] \in I_P$ et non($c \vdash C$). Dans ce cas nous dirons que $a[c]$ est un *symptôme calculé*. Un symptôme calculé est donc un élément de I_P qui n'appartient pas à $T_\vdash(pppf(T_P))$. Nous rappelons que $T_\vdash(pppf(T_P)) = pppf(T^{\cup}_{\vdash,P}) = pppf(T^{\circ}_{\vdash,P})$.

Exemple 6 $go(C,E,S)[\exists A\ (C = [0',0',0',0',1'] \wedge S = [1',1'] \wedge E = [A,1',\sim A])]$ est un symptôme calculé pour le programme DETECT2 (voir exemples 4 et 5).

Définition 16

• P *est* complet *(resp.* suffisant*) pour l'atome contraint* $a[c]$ *si pour toute contrainte* c' *telle que* $c' \vdash \{c\}$ *et* $a[c'] \in I_P$, $a[c'] \in pppf(T^{\circ}_{\vdash,P})$ *(resp.* $a[c'] \in pgpf(T^{\circ}_{\vdash,P})$*).*

• $a[c]$ *est* couvert *par* P *relativement à* I_P *si* $a[c] \in T^{\circ}_{\vdash,P}(I_P)$.

• $a[c]$ *est* complètement couvert *par* P *relativement à* I_P *si pour toute contrainte* c' *telle que* $c' \vdash \{c\}$ *et* $a[c'] \in I_P$, $a[c']$ *est couvert par* P *relativement à* I_P.

• *Une* insuffisance forte *est un atome contraint* $a[c] \in I_P$ *non couvert par* P *relativement à* I_P *(i.e.* $a[c] \in I_P - T^{\circ}_{\vdash,P}(I_P)$*).*

• *Une* insuffisance faible *est un atome contraint* $a[c]$ *non complètement couvert par* P *relativement à* I_P *(i.e. il existe* c' *telle que* $c' \vdash \{c\}$ *et* $a[c']$ *est une insuffisance forte).*

• *L'atome contraint* $a[c]$ *est un* symptôme d'incomplétude *(resp.* d'insuffisance*) de* P *relativement à* I_P *si* $a[c] \in I_P - pppf(T^{\circ}_{\vdash,P})$ *(resp.* $a[c] \in I_P - pgpf(T^{\circ}_{\vdash,P})$*).*

Ces définitions (couvert/complètement couvert, insuffisance forte/faible, symptôme d'insuffisance/d'incomplétude) sont des extensions des définitions existantes en programmation logique [15, 7]. Elles ont l'avantage d'être donnée ici dans un cadre uniforme. Ainsi, elles sont mieux comprises, grâce notamment aux arbres de preuve et à l'opérateur $T^{\circ}_{\vdash,P}$. Le cas particulier de la programmation logique est obtenu en remplaçant les contraintes par des substitutions et la couverture par la couverture unique "moins générale". Le cadre unifié contribue alors à la comparaison de [8] et [7].

Lemme 17 *Si $a[c]$ est une insuffisance forte alors $a[c]$ est une insuffisance faible.*
Il existe une insuffisance forte si et seulement si il existe une insuffisance faible.
Si $a[c]$ est un symptôme d'insuffisance alors $a[c]$ est un symptôme d'incomplétude.
S'il existe un symptôme d'insuffisance alors il existe une insuffisance forte.

Le lemme 17 est essentiel dans la mesure où il exprime les liens entre les différentes notions et montre que s'il existe un symptôme alors il existe une erreur. Le cas particulier où l'erreur est le symptôme fait correspondre insuffisance forte à symptôme d'insuffisance et insuffisance faible à symptôme d'incomplétude.

L'algorithme de diagnostic est utilisé dans le cas ou un symptôme est observé lors d'un calcul. Nous montrons donc qu'un symptôme calculé est un cas particulier de symptôme d'insuffisance (ou d'incomplétude). Ce cas particulier correspond à l'existence d'un arbre SLD fini pour la question concernant l'atome du symptôme.

Lemme 18
1. S'il existe un arbre SLD fini pour la question a dont les contraintes réponses associées à ses succès sont $c_1, \ldots, c_m$ alors il existe un entier k tel que, pour toute contrainte c, si $a[c] \in T_P \downarrow k$ alors $c \vdash \emptyset$ ou il existe $1 \leq i \leq m$ tel que $c = c_i$.
2. Pour tout entier n, si $a[c] \in T^{\circ}_{\vdash,P} \downarrow n$ alors il existe C tel que $c \vdash C$ et, pour toute contrainte $c' \in C$, $a[c'] \in T_P \downarrow n$.
3. Si $a[c] \in T^{\circ}_{\vdash,P} \downarrow \omega$ et il existe un arbre SLD fini pour la question a dont les succès sont $c_1, \ldots, c_m$ alors il existe $C \subseteq \{c_1, \ldots, c_n\}$ tel que $c \vdash C$ (i.e. $a[c] \in SS_{\vdash}(P)$).
4. Si $a[c]$ est un symptôme calculé alors $a[c]$ est un symptôme d'insuffisance.

Par conséquent, s'il y a un symptôme calculé alors il y a une insuffisance forte et une insuffisance faible (indépendamment de la règle de calcul).

Il ressort de cette étude qu'il existe deux familles de définitions concernant les réponses manquantes. Ces deux familles étaient déjà présentes en programmation logique [8, 7] et l'introduction des contraintes a permis d'éclairer les subtilités qui les séparent. C'est grâce à l'opérateur $T^{\circ}_{\vdash,P}$, parce que son plus petit point fixe est égal à $T_{\vdash}(pppf(T_P))$ (prouvé par l'égalité de ces deux ensembles avec $pppf(T^{\cup}_{\vdash,P})$) et parce qu'il n'y a plus la propriété de couverture unique, que l'on saisit pleinement ces différences. Il est intéressant de noter que c'est l'opérateur $T^{\circ}_{\vdash,P}$ qui a été retenu dans les définitions, alors qu'il n'est pas, a priori, le plus naturel pour définir $SS_{\vdash}(P)$. Soulignons qu'en général $pgpf(T^{\circ}_{\vdash,P}) \neq T_{\vdash}(pgpf(T_P)) \neq pgpf(T^{\cup}_{\vdash,P})$.

En programmation logique, sont associé à ces deux familles de définitions deux familles d'algorithmes. Les algorithmes de la famille [8] (insuffisance forte) fournissent une notion d'erreur plus précise que ceux de la famille [7] (insuffisance faible), mais l'interaction l'oracle (la sémantique attendue) y est plus compliquée. En effet, dans le premier cas l'oracle répond aux questions par des substitutions (instanciation de variables) alors que dans le second cas il se contente de répondre par oui ou non. Instancier un atome se traduit en programmation logique avec contraintes par contraindre un atome contraint, c'est-à-dire fournir une contrainte à l'algorithme. Dans certain cas cette contrainte peut être compliqué. Il est plus simple de répondre oui ou non que d'en expliquer les raisons. C'est pour cela que les définitions basée sur "non complètement couvert", semblent mieux adaptée pour CLP.

Nous proposons maintenant un algorithme inspiré de [7] dont la donnée est un symptôme calculé selon la stratégie standard et le résultat est une insuffisance faible. Nous ne cherchons pas ici à prouvé la correction de cet algorithme.

Une forêt de recherche pour $p(\tilde{x})[c]$ est constituée d'un arbre pour chaque clause $R \in P$ d'arité > 0 du paquet de p telle que si $p(\tilde{x}) \leftarrow c_1 \,\square\, b_1, \ldots, b_n$ est une variante de R alors non($c_1 \wedge \exists_{-\tilde{x}}\, c \vdash \emptyset$). La racine de l'arbre est $\langle b_1, c_1 \wedge \exists_{-\tilde{x}}\, c\rangle$. Si $\langle b_i, c_i\rangle$ est un nœud de l'arbre et si $C = \{\bigwedge_{c'' \in const(S)} c'' \mid S$ est un squelette réponse pour b_i et non($AC(S) \wedge c_i \vdash \emptyset$)$\}$ alors pour chaque $c' \in C$, $\langle b_i, c_i\rangle$ a un fils étiqueté par $\langle b_{i+1}, c'\rangle$ si $i < n$ et $\langle \square, c'\rangle$ si $i = n$.

On appelle *question d'incomplétude* pour le nœud N étiqueté par $\langle b_i, c_i\rangle$ la question : "Existe-t-il c' telle que $a[c'] \in I_P$, $c' \vdash \{c_i\}$ et non($c' \vdash \{c'' \mid \langle x, c''\rangle$ étiquette un fils de $N\}$) ?".

Soit $a[c]$ l'entrée de l'algorithme, on pose la question d'incomplétude pour les nœuds étiquetés par $\langle b_i, c_i\rangle$ ($b_i \neq \square$) de la forêt de recherche de $a[c]$. Si la réponse est oui l'algorithme est appelé récursivement avec $b_i[\exists_{-b_i}\, c_i]$. Si la réponse est non pour tous les nœuds alors l'algorithme renvoie $a[c]$ qui est une insuffisance faible.

Exemple 7 Session de diagnostic d'insuffisance pour le programme DETECT2.
Les contraintes présentées dans cet exemple ont été simplifiées par Prolog III.
Symptôme : $go(C, E, S)[C = [0', 0', 0', 0', 1'] \wedge E = [A, 1', \sim A] \wedge S = [1', 1']]$
Existe-t-il une contrainte c telle que $go(C, E, S)[c]$ est attendu, $c \vdash \{\exists A\,(C = [0', 0', 0', 0', 1'] \wedge E = [A, 1', \sim A] \wedge S = [1', 1']\}$ et non($c \vdash \emptyset$) ? `OUI`
Existe-t-il une contrainte c telle que $circuit(C, E, S)[c]$ est attendu, $c \vdash \{\exists A\,(C = [0', 0', 0', 0', 1'] \wedge E = [A, 1', \sim A] \wedge S = [1', 1']\}$ et non($c \vdash \emptyset$) ? `OUI`
Existe-t-il une contrainte c telle que $au\_plus_1(C, X)[c]$ est attendu, $c \vdash \{C = [0', 0', 0', 0', 1'] \wedge X = 1'\}$ et non($c \vdash \{C = [0', 0', 0', 0', 1'] \wedge X = 1'\}$) ? `NON`
Erreur : $circuit(C, E, S)[\exists A\, C = [0', 0', 0', 0', 1'] \wedge E = [A, 1', \sim A] \wedge S = [1', 1']]$
L'algorithme a isolé une définition responsable du symptôme. L'erreur vient du fait que dans la définition de $circuit$ le XOR a été codé par un "NOT OR".

6 Conclusion

Nous avons reformulé la sémantique opérationnelle et la sémantique déclarative des programmes logiques avec contraintes. Nous avons ainsi un cadre formel uniforme dans lequel sont reliées simplement les notions de réponse calculée et de réponse au sens déclaratif, par l'intermédiaire d'une relation abstraite de couverture. La sémantique des programmes peut ainsi être définie en référence à un domaine ou une théorie, et peut même rendre compte d'un algorithme incomplet de test de satisfaction de contraintes. Nous généralisons la sémantique classique de [12].

L'intérêt principal de cette nouvelle sémantique réside principalement dans l'étude du diagnostic déclaratif d'erreurs. Nous avons pu définir les notions de symptôme d'insuffisance et symptôme d'incomplétude ainsi que les deux types d'erreur associés dans un cadre uniforme et général, contribuant de fait à leur comparaison.

Une prochaine étape est, bien entendu, d'élaborer des algorithmes optimisés basés sur ces définitions en vue de les implanter dans les systèmes CLP existant. Ces

algorithmes se comportent comme des sortes de traces intelligentes. Ils interagissent avec l'utilisateur seulement par l'information pertinente, en allant "droit à l'erreur". Ils fournissent une partie de code erronée, la correction s'en trouve ainsi facilitée.

L'algorithme présentée dans la section 5 suppose un symptôme calculé par la stratégie standard. En vertu des lemmes 17 et 18, il serait intéressant d'avoir un algorithme indépendant de la règle de sélection utilisée pour calculer le symptôme. Les deux familles de symptômes et d'erreurs étant définis dans un même formalisme, nous pouvons imaginer en déduire un algorithme qui les fait collaborer, dans la recherche d'une erreur, alliant questions simples et erreur plus précise.

Certains programmes sont de la forme : $go(\tilde{x}) \leftarrow grosse\_contrainte \ \Box$. Nous observons alors un symptôme d'incomplétude $go(\tilde{x})[c]$ puis invoquons un algorithme de diagnostic qui fournit l'erreur $go(\tilde{x})[c]$. Ce n'est pas très intéressant ! Nous pouvons 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 ici. Déboguer la $grosse\_contrainte$ n'a pas de sens dans notre cadre au sens strict : les prédicats de contrainte sont supposés corrects. Concernant ce programme, il existe un symptôme parce que le programmeur a fait une erreur en écrivant la $grosse\_contrainte$. Il devrait être certainement possible de dire plus que : $go(\tilde{x})[c]$ est une erreur. L'objectif est de se donner des moyens formels appropriés basés sur une sémantique des variables de la contrainte.

De plus en plus de systèmes de contraintes utilisent, pour des raisons pratiques, des *approximations* des domaines de valeurs possibles des variables. Les approximations fournies contiennent toutes les réponses mais éventuellement d'autres valeurs (c'est-à-dire que la contrainte réponse est couverte par l'approximation). Elles sont, par exemple, des intervalles sur des domaines continus ($I\!R$: CLP(BNR), Prolog IV) ou sur des domaines (finis) discrets ($\mathcal{FD}$: CHIP). Parmi les travaux en cours, nous essayons de faire évoluer ce cadre pour en tenir compte. Les approximations ont des propriétés analogues à la relation de couverture. En définissant des nouvelles notions de symptômes et d'erreurs, nous profiterons de ces analogies.

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] 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.

[3] 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.

[4] Alain COLMERAUER. An Introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.

[5] 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.

[6] Pierre DERANSART and Jan MAŁUSZYŃSKI. *A Grammatical View of Logic Programming*. MIT Press, 1993.

[7] 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.

[8] Gérard FERRAND. Error Diagnosis in Logic Programming: an adaptation of E. Y. Shapiro's method. *Journal of Logic Programming*, 4:177–198, 1987.

[9] 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.

[10] Roberto GIACOBAZZI, Saumya K. DEBRAY and Giorgio LEVI. A Generalized Semantics for Constraint Logic Programs. In *International Conference on Fifth Generation Computer Systems*, pages 581–591, 1992.

[11] Joxan JAFFAR and Jean-Louis LASSEZ. Constraint Logic Programming. Technical Report 86/73, Department of Computer Science, Monash University, 1986. (Also available in 14^{th} *ACM Symposium on Principles of Programming Languages*, 1987).

[12] Joxan JAFFAR and Michael J. MAHER. Constraint Logic Programming: a survey. *Journal of Logic Programming*, 19-20:503–581, 1994.

[13] François LE BERRE and Alexandre TESSIER. Declarative Incorrectness Diagnosis of Constraint Logic Programs. Technical Report 95/08, LIFO, University of Orléans, 1995.

[14] 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.

[15] Ehud Y. SHAPIRO. *Algorithmic Program Debugging*. ACM Distinguished Dissertation. MIT Press, 1982.

[16] Alexandre TESSIER. Diagnostic Declaratif d'Insuffisance en Programmation Logique avec Contraintes. Technical Report 96/04, LIFO, University of Orléans, 1996.

[17] Pascal VAN HENTENRYCK. Constraint Logic Programming. *Knowledge Engineering Review*, 6(3):151–194, 1991. (Also available as Brown University Technical Report CS-91-05).