Méta-Programmation en Programmation en Logique

Alexandre TESSIER

Université d'Orléans

Stage de D.E.A.
— 1992 —

Remerciements

M. Bergère et G. Ferrand ont proposé ce stage de D.E.A., et ont accepté de m'encadrer durant sa réalisation.

Je remercie M. Bergère pour sa disponibilité, sa présence journalière et ses nombreuses explications qui m'ont permis de mener à bien ce travail.
Je remercie G. Ferrand qui, au cours des réunions hebdomadaires, a donné une direction à ce stage et a prodigué d'excellents conseils, et qui, par ses connaissances et sa grande rigueur, est à l'origine des idées développées.

Je les remercie tous deux pour le temps qu'ils ont consacré à ce stage.

Table des matières

1 Introduction **5**
1.1 Définition d'un méta-programme . . . 5
1.2 Méta-programmation en Prolog . . . 5
1.3 Le méta-interprète *vanilla* . . . 7

2 Représentation du programme objet — État de l'art **9**
2.1 Introduction . . . 9
2.2 Représentation typée . . . 9
2.2.1 Pourquoi la logique typée ? . . . 9
2.2.2 Passage en représentation typée . . . 10
2.2.3 Théorèmes . . . 11
2.2.4 Discussion . . . 12
2.3 Représentation close . . . 12
2.3.1 Pourquoi la représentation close ? . . . 12
2.3.2 Passage en représentation close . . . 12
2.3.3 Théorèmes . . . 14
2.3.4 Les prédicats méta-logiques . . . 15
2.3.5 Discussion . . . 15
2.4 Conclusion . . . 16

3 Notations utilisées et rappels sur l'opérateur T **19**

4 Représentation implicite **21**
4.1 Motivations . . . 21
4.2 Représentation implicite du programme P . . . 21
4.3 Théorèmes . . . 22
4.4 Preuves . . . 24
4.4.1 Preuves des lemmes . . . 24
4.4.2 Preuve du théorème 4.1 . . . 28
4.5 Comparaison avec Hill et Lloyd, et Conclusion . . . 31

5 Représentation close **33**
5.1 Motivations . . . 33
5.2 Représentation close du programme P . . . 33
5.3 Théorèmes . . . 34
5.4 Preuves . . . 35
5.4.1 Preuve du théorème 5.1 . . . 35
5.4.2 Preuve du théorème 5.2 . . . 37
5.5 Conclusion . . . 39

6 **Sémantique close, représentation implicite et représentation close** **41**
6.1 Introduction . . . 41
6.2 Le méta-programme $\mathbf{H}_P$ . . . 41
6.3 Théorèmes . . . 41
6.4 Preuves . . . 43
6.4.1 Preuve du théorème 6.1 . . . 43
6.4.2 Justification des remarques . . . 43
6.4.3 Preuve du théorème 6.9 . . . 44
6.4.4 Preuve du théorème 6.10 . . . 45
6.4.5 Preuve des corollaires . . . 45
6.5 Conclusion . . . 45

7 **Représentation quelconque du programme objet** **47**
7.1 Introduction . . . 47
7.2 Définition du langage . . . 47
7.3 Le méta-interprète *Super_vanilla* . . . 48
7.4 Définition de *instance* et *appartenance* . . . 49
7.4.1 Représentation . . . 49
7.4.2 Les méta-interprètes R_v et R_c . . . 50
7.5 Théorèmes . . . 52
7.6 Preuves . . . 52
7.6.1 Preuve du théorème 7.1 . . . 52
7.6.2 Preuve du théorème 7.2 . . . 53
7.7 Le problème de la négation . . . 53
7.7.1 Sémantique négative d'un programme . . . 53
7.7.2 Théorèmes . . . 54
7.7.3 Preuves . . . 54
7.8 Conclusion . . . 55

8 **Conclusion** **57**

Chapitre 1

Introduction

M. Bergère a réalisé un système de mise au point de programmes logiques en Prolog. Ceci l'a conduit à écrire des méta-programmes. Mais la mise en œuvre de méta-programmes pose encore des problèmes de sémantique (et d'implantation).

Aussi, avec G. Ferrand, il a proposé ce stage dont l'objectif est de contribuer à la réalisation de son système, au travers d'une étude théorique de l'état de l'art de la méta-programmation en Prolog.

1.1 Définition d'un méta-programme

Un méta-programme est un programme qui utilise un autre programme (le programme objet) en donnée (ex : interprète, compilateur, débogueur, générateur de programmes ...).

Le langage dans lequel est rédigé le méta-programme est appelé le méta-langage, alors que le langage du programme objet est appelé le langage objet. Le langage objet et le méta-langage peuvent être identiques; on parlera d'interprète méta-circulaire pour désigner un interprète écrit dans le langage qui est interprété. En Prolog, on peut écrire un interprète méta-circulaire[1] de façon beaucoup plus claire et concise que dans d'autres langages (même de haut niveau, ex : LISP).

1.2 Méta-programmation en Prolog

Il est important de donner une sémantique aux méta-programmes si l'on souhaite pouvoir faire des preuves de complétude ou de correction. C'est notre but, et nous pouvons déjà exhiber les problèmes que pose la méta-programmation en logique.

Prolog, par la structure même du langage, est bien adapté à la méta-programmation. Mais, en Prolog, la définition de méta-programme est plus compliquée à cause des possibilités fournies par les interprètes Prolog habituels et les théories sous-jacentes au langage.

Nous pouvons d'une part écrire de façon purement déclarative (ou "logique") un méta-programme utilisant une représentation termale du programme objet. Ce sera longuement discuté par la suite. Nous pouvons aussi nous servir du prédicat méta-logique fourni par les interprètes : le prédicat *clause*. Les programmes ne peuvent plus être alors considér'es comme des ensembles de clauses de la logique du premier

[1] Voir figure 1.3 page 7.

$p(f(X)) \leftarrow f(X)$	ou	$p(f(X)) \leftarrow f(p(X))$
$f(a) \leftarrow$		$f(p(a)) \leftarrow$

Figure 1.1: *méta-programme mélangeant symbole de fonction et symbole de prédicat*

ordre. Les symboles de prédicats du programme objet sont aussi des symboles de fonction du méta-programme. Cela se complique encore si les symboles de fonction du méta-programme sont également des symboles de prédicats du méta-programme comme dans l'exemple de la figure 1.1 page 6, ou encore avec le méta-interprète le plus simple, utilisant le prédicat *call* présenté dans la figure 1.2 page 6 (dans le chapitre 7 nous verrons une sémantique qui peut s'adapter à ces programmes).

Dans le dernier exemple, nous sommes loin de la logique du premier ordre. La méta-variable A, qui est un terme dans la tête de la clause, sera considérée par l'interprète Prolog comme un atome dans le corps de la clause. Ici, ce n'est plus comme dans les exemples précédents un symbole de prédicat identique à un symbole de fonction, c'est une variable dont l'instance peut être un terme et un atome à la fois.

D'autre part, en Prolog, il existe de nombreux prédicats systèmes permettant d'écrire des programmes auxquels on ne peut donner la sémantique logique classique. Par exemple, les prédicats *var*, *constant*, *clause*, *write* ... Ces programmes ont généralement une sémantique non fermée par substitution, mais nous verrons comment, avec la représentation close définie dans [8], nous arrivons à faire des preuves de correction ou de complétude.

Le sujet est donc vaste, et pour ne pas nous égarer, il faut définir ce que nous entendons par méta-programme. Nous ne nous intéresserons qu'aux méta-programmes "purs", c'est à dire aux méta-programmes qui manipulent un programme objet à travers une représentation de ce dernier. Nous étudierons différentes représentations pour le programme objet. Les deux représentations les plus courantes sont la représentation implicite (Chapitre 4) qui consiste à représenter une constante par une constante, une variable par une variable, un symbole de fonction par un symbole de fonction de même arité et un symbole de prédicat par un symbole de fonction de même arité, et la représentation close (Chapitre 5) qui lui est similaire à la différence qu'une variable est représentée par une constante. En représentation close, le programme objet est représenté par des termes clos. Les instances incorrectes d'atomes du programme objet[2] obtenues en représentation implicite sont éliminées; il est, de plus, possible d'obtenir dans les racines d'arbre de preuve des représentations d'atomes dont certaines représentations d'instances ne sont dans aucune racine

[2]Au niveau méta, symboles de prédicats objet et symboles de fonctions ne sont pas differenciés.

$$solve(A) \leftarrow A$$

Figure 1.2: *Méta-interprète avec méta-variable*

En trois clauses :

$$solve(empty) \leftarrow$$
$$solve(x \,\&\, y) \leftarrow solve(x) \wedge solve(y)$$
$$solve(x) \leftarrow clause(x, y) \wedge solve(y)$$

ou en quatre clauses en ajoutant :

$$solve(not\, A) \leftarrow \neg solve(A)$$

Si $p(x,z) \leftarrow q(x,y) \wedge r(y,z)$ est une clause du programme objet alors on trouve le fait $clause(p(x,z),\, q(x,y) \,\&\, r(y,z)) \leftarrow$ dans la définition de $clause$.

Figure 1.3: *Le méta-interprète vanilla*

d'arbre de preuve. Cette représentation du programme objet est donc bien adaptée au débogage[3].

Enfin, dans le chapitre 7, nous étendrons le langage de la programmation en logique pour ressembler à Prolog (point de vue termal), tout en restant déclaratif. Nous définirons alors une nouvelle représentation du programme objet.

Les notations, définitions et théorèmes régulièrement utilisés sont présentés dans le chapitre 3.

Le chapitre 2 présente un extrait de l'état de l'art de la méta-programmation en logique. Il se limite aux travaux de Lloyd; il est à préciser que la majorité des travaux ne se placent pas exactement dans le même cadre que le nôtre, puisque, pour la plupart, ils oublient Prolog et sont basés sur des langages différents[4].

Rappelons que l'aspect dynamique de la méta-programmation (les prédicats *assert* et *retract*) ne nous préoccupe pas ici, nous ne nous penchons que sur l'aspect statique de celle-ci.

1.3 Le méta-interprète *vanilla*

En programmation logique, un méta-programme se distingue de façon tout à fait naturelle. Il s'agit du méta-interprète *vanilla*. C'est la référence en matière de méta-programmation en logique. Il constitue un point de départ conseillé dans l'étude de celle-ci.

Les méta-programmes étudiés seront le plus souvent basés sur le méta-interprète (standard et classique) *vanilla* de la figure 1.3 page 7.

Le méta-interprète *vanilla* peut paraître trop simple, surtout en comparaison des interprètes méta-circulaires qui sont écrits habituellement avec les langages fonctionnels ... Il n'en est rien. C'est un méta-interprète clair et concis, et tout à fait efficace.

[3] L'instance d'une erreur n'est pas toujours une erreur.

[4] Christiansen dans [3]; Lloyd avec le langage Gödel ...

Sa structure sert de base à de nombreux méta-programmes dont on peut voir des exemples dans [12, Chapitre 19] (aussi [8], [11] ...).

vanilla constitue le cœur de notre étude. Il sera chaque fois adapté aux besoins pour faire ressortir son intérêt profond, sa généralité, sa simplicité et sa polyvalence.

Chapitre 2

Représentation du programme objet — État de l'art

2.1 Introduction

Dans un méta-programme, il est nécessaire de représenter le programme objet. En programmation en logique, nous pouvons envisager plusieurs représentations, les plus utilisées étant la représentation implicite et la représentation close (étudiées dans les chapitres 4 et 5).

Lloyd, dans [8], décide d'écrire le méta-programme dans un langage différent. Il choisit la logique typée, et étudie dans ce cadre les deux représentations classiques. Il a également développé le langage Gödel, dans l'objectif de faciliter l'écriture de méta-programmes. Nous constaterons que l'écriture de méta-programmes aboutit parfois à l'écriture de programmes infinis ou de programmes pour lesquels les preuves de correction et de complétude sont longues. Aussi, l'étude de la méta-programmation peut se faire dans un langage mieux adapté, proche de Prolog, entièrement déclaratif, qui pourra être compilé en Prolog (voir par exemple les travaux de Christiansen dans [3]).

2.2 Représentation typée

Nous cherchons ici à donner une sémantique logique au méta-interprète *vanilla* pour pouvoir comparer la sémantique du programme objet et celle du méta-interprète.

2.2.1 Pourquoi la logique typée ?

Lloyd dans [8] propose de travailler dans une logique typée. En effet, dans le méta-interprète *vanilla*, on peut distinguer deux "sortes" de variables :

- Celles qui sont des termes du programme objet, c'est-à-dire celles qui sont dans le domaine de l'interprétation du programme objet. Elles apparaissent dans la définition de *clause*.

- Celles qui sont des termes du méta-programme, c'est-à-dire celles destinées à être instanciées par des atomes (littéraux) ou conjonctions d'atomes (de littéraux) du programme objet. Elles apparaissent dans la définition de *solve*.

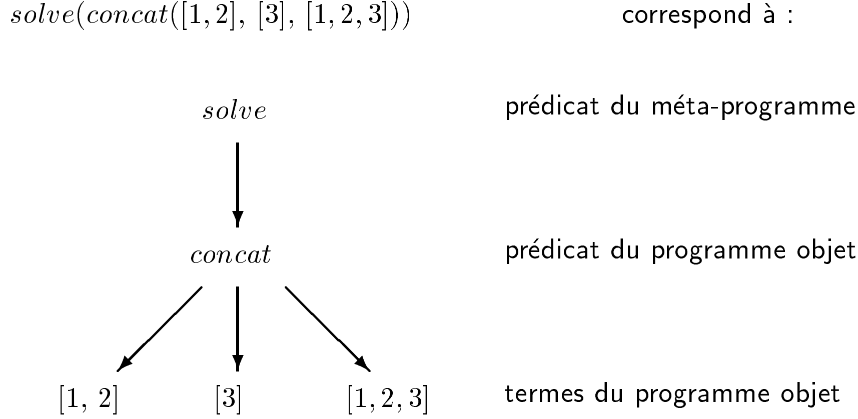


Figure 2.1: *Les différents types d'objets dans le méta-programme.*

On peut donc distinguer trois "couches" dans le méta-langage (voir Figure 2.1 page 10). Elles permettent d'exprimer :

- les termes du langage objet,
- les formules du langage objet qui sont les termes du méta-langage,
- les formules du méta-langage.

2.2.2 Passage en représentation typée

Il existe donc deux "sortes" de termes pour le méta-interprète, les termes du langage objet et les termes représentant des formules du langage objet. Pour pallier ce problème, Lloyd introduit des types, et donne la représentation typée (du programme objet) suivante :

passage d'un langage $\mathcal{L}$ à un langage typé $\mathcal{L}'$ possédant deux types : o (pour objet) et μ (pour méta)[1]

a constante de $\mathcal{L}$	$\longleftrightarrow$	a' constante de type o de $\mathcal{L}'$
x variable de $\mathcal{L}$	$\longleftrightarrow$	x' variable de type o de $\mathcal{L}'$
f fonction n-aire de $\mathcal{L}$	$\longleftrightarrow$	f' fonction n-aire de type $o \times o \times \cdots \times o \rightarrow o$ de $\mathcal{L}'$
p prédicat n-aire de $\mathcal{L}$	$\longleftrightarrow$	p' fonction n-aire de type $o \times o \times \cdots \times o \rightarrow \mu$ de $\mathcal{L}'$

Nous supposons que les applications $a \rightarrow a'$, $x \rightarrow x'$, $f \rightarrow f'$ et $p \rightarrow p'$ sont toutes bijectives.

[1] Les notions de logique typée et langage du premier ordre typé suffisantes à la compréhension se trouvent dans [9].

Nous ajoutons au langage $\mathcal{L}'$ une constante : *empty* de type μ, les symboles de fonctions : &, if et *not* de types $\mu \times \mu \to \mu$, $\mu \times \mu \to \mu$ et $\mu \to \mu$. Le langage $\mathcal{L}'$ contient les deux symboles de prédicats *solve* et *clause* de types μ. Le passage des termes et atomes du langage $\mathcal{L}$ dans le langage $\mathcal{L}'$ est défini par induction de façon évidente ... Si F et G sont des formules de $\mathcal{L}$ représentées par F' et G' dans $\mathcal{L}'$ alors $\neg F$, $F \wedge G$ et $F \leftarrow G$ sont représentées respectivement par les termes $not\, F'$, $F' \,\&\, G'$ et $F' \, if \, G'$ de types μ.

Soit $\mathbf{V}$ le programme normal typé, basé sur $\mathcal{L}'$, suivant :

$$
\begin{array}{l}
solve(empty) \leftarrow \\
\forall_\mu \, x, y \; solve(x \,\&\, y) \leftarrow solve(x) \wedge solve(y) \\
\forall_\mu \, x \; solve(not \, x) \leftarrow \neg solve(x) \\
\forall_\mu \, x, y \; solve(x) \leftarrow clause(x \, if \, y) \wedge solve(y)
\end{array}
$$

Si P est un programme normal basé sur le langage $\mathcal{L}$, nous noterons $\mathbf{V}_P$ le programme constitué de $\mathbf{V}$ et des clauses : $\forall_o \, x'_1 \ldots x'_k \, clause(A' \, if \, Q') \leftarrow$, pour chaque clause $A \leftarrow Q$, avec les variables $x'_1 \ldots x'_k$, du programme P ($Q' = empty$ si Q est vide).

2.2.3 Théorèmes

Soient P un programme normal et Q un but normal avec les variables $x'_1, \ldots, x'_k$. Soit $\mathbf{V}_P$ le programme tel qu'il a été défini plus haut.

Théorèmes

1. $comp(P)$ est consistant `ssi` $comp(\mathbf{V}_P)$ est consistant.
2. $\{x_1/t_1, \ldots, x_k/t_k\}$ est une réponse correcte pour $comp(P) \cup \{\leftarrow Q\}$ `ssi` $\{x'_1/t'_1, \ldots, x'_k/t'_k\}$ est une réponse correcte pour $comp(\mathbf{V}_P) \cup \{\leftarrow solve(Q')\}$.
3. $\leftarrow Q$ est conséquence logique de $comp(P)$ `ssi` $\leftarrow solve(Q')$ est conséquence logique de $comp(\mathbf{V}_P)$.

Corollaires

1. si $\{x'_1/t'_1, \ldots, x'_k/t'_k\}$ est une réponse calculée pour $\mathbf{V}_P \cup \{\leftarrow solve(Q')\}$ alors $\{x_1/t_1, \ldots, x_k/t_k\}$ est une réponse correcte pour $comp(P) \cup \{\leftarrow Q\}$.
2. si $\mathbf{V}_P \cup \{\leftarrow solve(Q')\}$ a un arbre-SLDNF d'échec fini alors $\leftarrow Q$ est conséquence logique de $comp(P)$.

Théorèmes

1. $\{x_1/t_1, \ldots, x_k/t_k\}$ est une réponse calculée pour $P \cup \{\leftarrow Q\}$ `ssi` $\{x'_1/t'_1, \ldots, x'_k/t'_k\}$ est une réponse calculée pour $\mathbf{V}_P \cup \{\leftarrow solve(Q')\}$.
2. $P \cup \{\leftarrow Q\}$ a un arbre-SLDNF d'échec fini `ssi` $\mathbf{V}_P \cup \{\leftarrow solve(Q')\}$ a un arbre-SLDNF d'échec fini.

Le détail des preuves se trouve dans [8].

2.2.4 Discussion

La représentation typée peut permettre de donner une sémantique aux méta-programmes "purs" basés sur le méta-interprète *vanilla* (voir [12, Chapitre 19] pour de nombreux exemples). Elle permet d'éliminer, par le typage des termes, les instances "impropres" obtenues en logique classique et les problèmes que cela cause (comme nous le verrons dans le chapitre suivant, ex : $solve(concat(concat(x, y, z)\,\&x, u\,if\ v, w))$ est une instance de $solve(concat(x, y, z)))$.

Par le typage des termes, nous assurons qu'une variable de type o sera instanciée par la représentation d'un terme du langage $\mathcal{L}$, et qu'une variable de type μ sera instanciée par la représentation d'une formule du langage $\mathcal{L}$.

Mais, dans le chapitre 4, nous montrerons des bonnes propriétés sur le méta-interprète *vanilla* sans types (c'est à dire celui programmé en Prolog, puisqu'on ne peut pas spécifier d'unification typée), dans la mesure où l'on ne s'intéresse qu'à des réponses pour $\mathbf{V}_P \cup \{solve(Q') \leftarrow\}$, Q' étant la représentation d'un atome Q de $\mathcal{L}$.

Il est regrettable que Lloyd n'explique pas pourquoi il s'est intéressé à la logique typée, comme nous avons tenté de le faire dans la section 2.2.1, surtout au regard des résultats obtenus dans le chapitre 4. Grâce au typage, le méta-interprète *vanilla* devient correct (en plus d'être complet); mais peut-être a-t-il voulu conserver un formalisme proche de celui exploité pour les programmes normaux (conséquence logique du complété) sans alourdir par les instances impropres qui rendent les preuves difficilement abordables.

D'un point de vue plus opérationnel, nous pourrons implanter un langage basé sur l'expression de clauses en logique typée (qui pourra être compilé en Prolog, il suffit de spécifier l'unification typée), avec lequel il sera aisé d'écrire des méta-programmes corrects, mais aussi de le prouver.

2.3 Représentation close

Nous présentons ici la deuxième partie de [8], concernant la représentation close.

2.3.1 Pourquoi la représentation close ?

Lloyd dans son article s'attache fortement à l'aspect SLDNF (voir même opérationnel) de la méta-programmation en logique.

Nous connaissons la différence qu'il existe entre la logique et Prolog. Pouvons-nous, sous certaine condition, donner une sémantique logique aux prédicats comme *var*, *constant* ... équivalente ou proche de leur sémantique opérationnelle ? C'est essentiellement dans ce cadre que Lloyd étudie la représentation close.

2.3.2 Passage en représentation close

Lloyd manipule une représentation close du programme objet dans une logique typée basée sur un langage $\mathcal{L}'$. $\mathcal{L}'$ contient trois types : o, μ et η.

Passage du langage $\mathcal{L}$ au langage typé $\mathcal{L}'$:

$$a \text{ constante de } \mathcal{L} \quad \longleftrightarrow \quad a' \text{ constante de type } o \text{ de } \mathcal{L}'$$

$$x \text{ variable de } \mathcal{L} \quad \longleftrightarrow \quad x' \text{ constante de type } o \text{ de } \mathcal{L}'$$

$$f \text{ fonction n-aire de } \mathcal{L} \quad \longleftrightarrow \quad f' \text{ fonction n-aire de type } o \times o \times \cdots \times o \to o \text{ de } \mathcal{L}'$$

$$p \text{ prédicat n-aire de } \mathcal{L} \quad \longleftrightarrow \quad p' \text{ fonction n-aire de type } o \times o \times \cdots \times o \to \mu \text{ de } \mathcal{L}'$$

Nous supposons que les applications $a \to a'$, $x \to x'$, $f \to f'$ et $p \to p'$ sont toutes bijectives.

Nous ajoutons au langage $\mathcal{L}'$ une constante : *empty* de type μ, les symboles de fonctions : &, if, *not*, *positive*, *negative* et *resultant* de types $\mu \times \mu \to \mu$, $\mu \times \mu \to \mu$, $\mu \to \mu$, $\mu \to \eta$, $\mu \to \eta$ et $\mu \times \mu \times \mu \times \mu \to \eta$. Le langage $\mathcal{L}'$ contient les symboles de prédicats *solve*, *succeed*, $fail$, *select*, *derive* et *clause* de types $\mu \times \mu$, $\mu \times \mu$, μ, $\mu \times \mu \times \eta \times \mu \times \mu$, $\eta \times \mu \times \mu$ et μ. Le passage des termes, atomes et formules du langage $\mathcal{L}$ dans le langage $\mathcal{L}'$ est défini par induction de façon évidente ...

Soit **G** le programme normal typé, basé sur $\mathcal{L}'$, suivant :

$$
\begin{array}{l}
\forall_\mu\, x, y\; solve(x, y) \leftarrow \\
\qquad succeed(x\, if\, x, y\, if\, empty) \\
\forall_\mu\, x\; succeed(x\, if\, empty, x\, if\, empty) \leftarrow \\
\qquad head_formula(x) \\
\forall_\mu\, l, r, s, u, v, w, x, y, z\; succeed(x\, if\, y, z) \leftarrow \\
\qquad select(y, l, positive(s), r, u) \wedge clause(w) \wedge \\
\qquad derive(resultant(x, l, s, r), w, v) \wedge succeed(v, z) \\
\forall_\mu\, l, r, s, u, x, y, z\; succeed(x\, if\, y, z) \leftarrow \\
\qquad select(y, l, negative(s), r, u) \wedge fail(empty\, if\, s) \wedge succeed(x\, if\, u, z) \\
\forall_\mu\, l, r, s, u, y\; fail(empty\, if\, y) \leftarrow \\
\qquad select(y, l, positive(s), r, u) \wedge \\
\qquad \forall_\mu\, w, z\, (fail(z) \leftarrow clause(w) \wedge derive(resultant(empty, l, s, r), w, z)) \\
\forall_\mu\, l, r, s, u, y\; fail(empty\, if\, y) \leftarrow \\
\qquad select(y, l, negative(s), r, u) \wedge succeed(empty\, if\, s, empty\, if\, empty) \\
\forall_\mu\, l, r, s, u, y\; fail(empty\, if\, y) \leftarrow \\
\qquad select(y, l, negative(s), r, u) \wedge fail(empty\, if\, s) \wedge fail(empty\, if\, u) \\
head_formula(empty) \leftarrow \\
\forall_\mu\, x\; head_formula(x) \leftarrow \\
\qquad conjunction_of_literals(x) \\
\forall_\mu\, x\; conjunction_of_literals(x) \leftarrow \\
\qquad literal(x) \\
\forall_\mu\, x, y\; conjunction_of_literals(x \,\&\, y) \leftarrow \\
\qquad literal(x) \wedge conjunction_of_literals(y) \\
\forall_\mu\, x\; literal(x) \leftarrow \\
\qquad atom(x) \\
\forall_\mu\, x\; literal(not\, x) \leftarrow \\
\qquad atom(x)
\end{array}
$$

Pour tout p symbole de prédicat de $\mathcal{L}$ d'arité n :

$$\forall_o x_1, \cdots, x_n \, atom(p'(x_1, \cdots, x_n)) \leftarrow$$

Lloyd ne donne pas de définition pour *select* et *derive* mais donne leurs sémantiques :

- *derive* est vrai si le premier argument est de la forme $resultant(E, F, G, H)$ avec $E \, if F \,\&\, G \,\&\, H$ représentant une résultante R et G un atome A, le second argument représente une clause C, et le troisième argument représente une résultante dérivée de R utilisant l'atome sélectionné A et une variante de C[2].

- *select* est vrai si son premier argument représente une conjonction de littéraux Q, le second argument représente la conjonction à "gauche" du littéral sélectionné dans Q, le troisième argument est de la forme $positive(E)$ si E est le littéral positif sélectionné ou de la forme $negative(E)$ si $\neg E$ est le littéral négatif sélectionné, le quatrième argument représente la conjonction à "droite" du littéral sélectionné dans Q, et le cinquième argument représente la conjonction de littéraux obtenue à partir de Q en supprimant le littéral sélectionné.

Si P est un programme normal basé sur le langage $\mathcal{L}$ alors on notera $\mathbf{G}_P$ le programme constitué de $\mathbf{G}$ et des clauses : $clause(A' \, if \, Q') \leftarrow$, pour chaque clause $A \leftarrow Q$ du programme P ($Q' = empty$ si Q est vide).

2.3.3 Théorèmes

Soient P un programme normal et Q un but normal. Soit $\mathbf{G}_P$ le programme tel qu'il a été défini plus haut.

Théorèmes

1. Si $\{x/(Q\theta)'\}$ est une réponse correcte pour $comp(\mathbf{G}_P) \cup \{\leftarrow succeed(Q' \, if \, Q', x \, if \, empty)\}$ alors θ est une réponse calculée pour $P \cup \{\leftarrow Q\}$.

2. Si $fail(empty \, if \, Q')$ est une conséquence logique de $comp(\mathbf{G}_P)$ alors $P \cup \{\leftarrow Q\}$ a un arbre SLDNF d'échec fini.

Corollaires

1. Si $\{x/(Q\theta)'\}$ est une réponse calculée pour $comp(\mathbf{G}_P) \cup \{\leftarrow succeed(Q' \, if \, Q', x \, if \, empty)\}$ alors θ est une réponse calculée pour $P \cup \{\leftarrow Q\}$.

2. Si la substitution identité est une réponse calculée pour $comp(\mathbf{G}_P) \cup \{\leftarrow fail(empty \, if \, Q')\}$ alors $P \cup \{\leftarrow Q\}$ a un arbre SLDNF d'échec fini.

3. Si $\{x/(Q\theta)'\}$ est une réponse correcte pour $comp(\mathbf{G}_P) \cup \{\leftarrow succeed(Q' \, if \, Q', x \, if \, empty)\}$ alors θ est une réponse correcte pour $comp(P) \cup \{\leftarrow Q\}$.

[2] Les définitions de résultantes et dérivations sont données dans [9].

4. Si $fail(empty\,if\,Q')$ est une conséquence logique de $comp(\mathbf{G}_P)$ alors $\leftarrow Q$ est une conséquence logique de $comp(P)$.

Le détail des preuves se trouve dans [8].

2.3.4 Les prédicats méta-logiques

Grâce à la représentation close et au méta-interprète $\mathbf{G}_P$, on peut donner la définition de méta-prédicats tel que *var*, *constant* ... Cette définition peut se faire à deux niveaux. Soit le programme P utilise ces "prédicats", soit c'est dans le méta-programme qu'ils apparaissent. Lloyd donne les définitions concernant le deuxième cas.

Par exemple, si $\mathcal{L}$ contient les constantes $a_1, \cdots, a_n$ alors *constant* est défini par :

$$\begin{array}{l} constant(a'_1) \leftarrow \\ \quad\vdots \\ constant(a'_n) \leftarrow \end{array}$$

Si $\mathcal{L}$ contient les symboles de fonctions $f_1, \cdots, f_m$ d'arités $k_1, \cdots, k_m$ alors *var* et *nonvar* sont définis par :

$$\begin{array}{l} \forall_o\, x\; nonvar(x) \leftarrow constant(x) \\ \forall_o\, x_1, \cdots, x_{k_1}\; nonvar(f'_1(x_1, \cdots, x_{k_1})) \leftarrow \\ \quad\vdots \\ \forall_o\, x_1, \cdots, x_{k_m}\; nonvar(f'_m(x_1, \cdots, x_{k_m})) \leftarrow \\ \forall_o\, x\; var(x) \leftarrow \neg nonvar(x) \end{array}$$

Avec ces définitions pour le méta-programme, nous pouvons nous intéresser à des buts tels que $\forall_o\, x\ \leftarrow solve(p'(x)) \wedge var(x)$.

2.3.5 Discussion

Le méta-interprète $\mathbf{G}_P$ est plus compliqué que le méta-interprète *vanilla*. *vanilla* fait apparaître un aspect déclaratif alors que $\mathbf{G}_P$ implante la SLDNF.

D'autre part, nous percevons une volonté opérationnelle sous-jacente à $\mathbf{G}_P$. En effet, $\mathbf{G}_P$ fait plus que rendre compte de la sémantique déclarative de P. Si *solve* n'avait qu'un seul argument, un interprète fournirait comme réponse à une question de la forme $solve(Q')$ avec Q' représentation de Q : OUI si Q est une réponse correcte pour P, ou NON sinon. Ici, grâce aux deux arguments de *solve*, un interprète donnera la réponse $solve(Q', R')$ de la même façon qu'il aurait donné la réponse R à la question Q (rappelons que Q' est clos même si Q ne l'est pas). En fait, nous verrons dans le chapitre 5 section 5.5 que nous pouvons conserver ce souci opérationnel en n'ayant qu'un seul argument et que nous obtenons même un méta-interprète plus polyvalent.

Nous pouvons voir dans le programme $\mathbf{G}_P$ (ou sa version simplifiée pour programmes définis donnée figure 2.2 page 16) que le prédicat *derive* a un rôle primordial. C'est lui qui assurera le lien entre les représentations de variables de l'ancienne résultante et leurs instances dans la nouvelle résultante. Il implante la notion de dérivation.

$$
\begin{aligned}
&solve(But, Rep) \leftarrow succes(But\,if\; But, Rep\,if\; empty)\\
\\
&succes(But\,if\; empty, But\,if\; empty) \leftarrow\\
&\qquad conjonction(But)\\
&succes(But\,if\; Reste, Rep) \leftarrow\\
&\qquad select(Reste, Gauche, Choix, Droite) \wedge\\
&\qquad clause(Cl) \wedge\\
&\qquad derive(But, Gauche, Choix, Droite, Cl, Nbut) \wedge\\
&\qquad succes(Nbut, Rep)
\end{aligned}
$$

Figure 2.2: Méta-interprète avec représentation close pour programmes définis

Le prédicat *clause* assure la seule interface entre **G** et le programme objet P, Comme la représentation est close, les atomes de la définition de *clause* sont tous clos. On constate alors que le typage est complètement inutile. Les réponses pour $solve(\cdots)$ seront obligatoirement bien formées ("typées") en se plaçant dans la logique classique (sans types).

2.4 Conclusion

Ce chapitre est consacré aux travaux de Hill et Lloyd issus de [8]. Cet article constitue le point de départ et la référence pour les chapitres suivants.

L'état de l'art exposé ici paraît maigre, mais ne reflète que la réalité quantitative des travaux sur la sémantique des méta-programmes en Prolog. Nous trouvons d'autres papiers sur la méta-programmation pour lesquels l'accent n'est pas mis sur la sémantique, ou bien, pour lesquels le langage logique (ou déclaratif) étudié n'est pas Prolog.

Dans l'article de Hill et Lloyd, la sémantique des méta-programmes est recherchée à travers l'interprétation en logique du premier ordre typée de ceux-ci. Les rapports entre la logique du premier ordre et Prolog sont bien connus; ceux entre la logique du premier ordre typée et Prolog le sont moins. En Prolog, l'unification typée n'existe pas. Aussi, les théorèmes de ce chapitre doivent être adaptés à Prolog. De plus, l'utilisation des conséquences logiques du programme ou de son complété sont fastidieuses à exhiber lorsque nous ne considérons plus des cas d'école.

Pour toutes ces raisons, nous décidons de reprendre les deux représentations du programme objet pour les étudier en parlant de plus petit point fixe et de racines d'arbres de preuve. Ce formalisme très simple a l'avantage d'être purement déclaratif et proche de Prolog. Nous n'étudions que les programmes définis, mais le chapitre 7 présente des résultats sur la sémantique "négative" des méta-programmes.

Qu'il s'agisse de la représentation implicite ou close, à chaque programme objet correspond un méta-interpète. Nous pouvons écrire un méta-interprète "universel" qui a la propriété d'interpréter tout programme objet, sans préciser, dans un premier temps, la représentation choisie. Le programme objet devient un argument de l'interprète *vanilla*. Ce méta-programme est présenté par la figure 2.3 page 17 et peut être baptisé *super_vanilla*. Le problème en suspens est l'existence d'une représentation du programme objet et d'une définition du prédicat *clause*. Le chapitre 7 écartera ces doutes en donnant la version complète de *super_vanilla*.

$$solve(P, empty) \leftarrow$$
$$solve(P, A \,\&\, B) \leftarrow solve(P, A) \wedge solve(P, B)$$
$$solve(P, A) \leftarrow clause(P, A\, if\; B) \wedge solve(P, B)$$

$clause(P, C)$ appartient à la sémantique du méta-interprète ssi C est la représentation d'une instance d'une clause du programme représenté par P.

Figure 2.3: *Méta-interprète indépendant de la représentation de* P

Chapitre 3

Notations utilisées et rappels sur l'opérateur T

Soit $\mathcal{L}$ un langage du premier ordre, on notera :

$VAR_{\mathcal{L}}$	: l'ensemble des variables de $\mathcal{L}$
$FONC_{\mathcal{L}}$	: l'ensemble des symboles de fonctions de $\mathcal{L}$
$PRED_{\mathcal{L}}$	: l'ensemble des symboles de prédicats de $\mathcal{L}$
$TERM_{\mathcal{L}}$	: l'ensemble des termes de $\mathcal{L}$
$ATOM_{\mathcal{L}}$	: l'ensemble des atomes de $\mathcal{L}$

Si P est un programme défini et $\mathcal{L}$ le langage de P, on notera T_P l'application :

$$T_P : \mathcal{P}(ATOM_{\mathcal{L}}) \longrightarrow \mathcal{P}(ATOM_{\mathcal{L}})$$

définie par :

$$T_P(I) = \{A \in ATOM_{\mathcal{L}} / \exists\, (B \leftarrow E) \in P \text{ et } \exists\, \sigma \text{ une substitution telles que } \sigma E \subseteq I \text{ et } \sigma B = A\}$$

Définition :

Un arbre de preuve est un arbre fini dont les nœuds sont étiquetés par des atomes et tel que pour chaque nœud il existe une instance de clause $A \leftarrow B_1 \wedge \cdots \wedge B_n$ telle que A soit l'étiquette du nœud et $B_1, \cdots, B_n$ les étiquettes de ses fils.

Les feuilles de l'arbre sont étiquetés par des instances de faits.

On note parfois $adp(Q)$ un arbre de preuve de racine étiquetée par Q.

Théorème :

Si RAP_P est l'ensemble des racines d'arbre de preuve de P alors

$$RAP_P = \bigcup_{n \in I\!N} T_P^n(\emptyset) = T_P \uparrow \omega = pppf(T_P)$$

Théorème :

I est un modèle de P `ssi` $T_P(I) \subseteq I$.

Définition :

I est un post-point fixe de T_P si $T_P(I) \subseteq I$.

Théorèmes :

- $pppf(T_P)$ est le plus petit post-point fixe de T_P.
- $pppf(T_P)$ est l'intersection de tous les post-points fixes de T_P.

Dans le chapitre 2, la sémantique des programmes est définie par leur interprétation en logique et par les conséquences logiques du complété pour les programmes normaux. Dans la suite, c'est le $pppf(T_P)$ qui définira la sémantique d'un programme défini P. Le premier théorème est à la base de toutes les démonstrations qui suivront.

Chapitre 4

Représentation implicite

4.1 Motivations

Reprenons ici la représentation typée, introduite par Lloyd et Hill dans [8], à laquelle nous retirons les types. Nous l'appelons représentation implicite, car les atomes sont représentés par un terme syntaxiquement "isomorphe".

De façon intuitive, nous nous apercevons que le typage n'est pas vraiment nécessaire si nous nous attachons à l'aspect termal plutôt qu'à l'aspect logique de la programmation en logique. Il est évident que, dans l'ensemble des racines d'arbre de preuve du méta-interprète, nous trouverons, si le programme objet contient des clauses avec variables, des atomes de la forme $solve(Q)$ où Q n'est pas représentation d'un atome du langage objet puisque les variables du programme objet sont représentées par des variables du méta-langage.

De façon plus formelle, sachant que nous nous intéressons aux réponses pour $\mathbf{V}_P \cup \{solve(Q') \leftarrow\}$ où Q' est la représentation d'un atome Q du langage objet, nous pouvons nous interroger sur l'influence de l'élimination du typage sur les résultats donnés dans [8].

Dans la suite nous nous intéresserons au $pppf$ de T_P (programme objet) et au $pppf$ de $T_{\mathbf{V}_P}$ réduits aux atomes de la forme $solve(Q')$ où Q' est représentation d'un atome Q. Nous montrerons que ces ensembles sont isomorphes. Le méta-interprète *vanilla* comme nous allons le voir est donc complet (nous parlerons de sa correction relative à un ensemble de buts dans la section 4.5 page 31).

4.2 Représentation implicite du programme P

Soit $\mathcal{L}$ un langage muni des connecteurs $\leftarrow$, $\wedge$ et $true$. Passage de $\mathcal{L}$ à $\mathcal{L}'$:

$$a \text{ constante de } \mathcal{L} \quad \longleftrightarrow \quad a' \text{ constante de } \mathcal{L}'$$

$$x \text{ variable de } \mathcal{L} \quad \longleftrightarrow \quad x' \text{ variable de } \mathcal{L}'$$

$$f \text{ fonction n-aire de } \mathcal{L} \quad \longleftrightarrow \quad f' \text{ fonction n-aire de } \mathcal{L}'$$

$$p \text{ prédicat n-aire de } \mathcal{L} \quad \longleftrightarrow \quad p' \text{ fonction n-aire de } \mathcal{L}'$$

Nous supposons que les applications $a \to a'$, $x \to x'$, $f \to f'$ et $p \to p'$ sont toutes bijectives.

Pour représenter les clauses exprimer dans le langage $\mathcal{L}$, nous ajoutons au langage $\mathcal{L}'$ une constante : *empty*, les symboles de fonctions : & et *if*. Le langage $\mathcal{L}'$ contient les deux symboles de prédicats *solve* et *clause*.

Le passage des termes, atomes et formules du langage $\mathcal{L}$ dans le langage $\mathcal{L}'$ est défini par induction.

Si $f(t_1, \cdots, t_n)$ est un terme de $\mathcal{L}$, alors $f(t_1, \cdots, t_n)$ est représenté par le terme $f'(t'_1, \cdots, t'_n)$ de $\mathcal{L}'$, où $t'_1, \cdots, t'_n$ sont les représentations respectives de $t_1, \cdots, t_n$.

Si $p(t_1, \cdots, t_n)$ est un atome de $\mathcal{L}$, alors $p(t_1, \cdots, t_n)$ est représenté par le terme $p'(t'_1, \cdots, t'_n)$ de $\mathcal{L}'$, où $t'_1, \cdots, t'_n$ sont les représentations respectives de $t_1, \cdots, t_n$.

true est représenté par *empty*.

Si F et G sont des formules de $\mathcal{L}$ représentées par F' et G' dans $\mathcal{L}'$ alors $F \wedge G$ et $F \leftarrow G$ sont représentées respectivement par les termes $F' \,\&\, G'$ et $F' \,if\, G'$.

Soit $\mathbf{V}$ le programme normal, basé sur $\mathcal{L}'$, suivant :

$$solve(empty) \leftarrow$$
$$solve(x \,\&\, y) \leftarrow solve(x) \wedge solve(y)$$
$$solve(x) \leftarrow clause(x \,if\, y) \wedge solve(y)$$

Si P est un programme défini basé sur le langage $\mathcal{L}$, nous notons $\mathbf{V}_P$ le programme constitué de $\mathbf{V}$ et des clauses : $clause(A' \,if\, Q') \leftarrow$, pour chaque clause $A \leftarrow Q$, du programme P ($Q' = empty$ si Q est vide).

Remarque : Nous notons X' pour dénoter la représentation d'un objet X du langage $\mathcal{L}$ dans le langage $\mathcal{L}'$.

4.3 Théorèmes

Théorème 4.1 :

Pour tout $Q \in ATOM_{\mathcal{L}}$: $Q \in pppf(T_P)$ `ssi` $solve(Q') \in pppf(T_{\mathbf{V}_P})$

Remarque 4.2 :

Si I' est un post-point fixe de $\mathbf{V}_P$, soit $I = \{\, Q \in ATOM_{\mathcal{L}} \,/\, solve(Q') \in I' \,\}$, alors I est un post-point fixe de P.

Les lemmes suivants sont utilisés pour la preuve du théorème 4.1 :
σ désigne une substitution de $TERM_{\mathcal{L}} \longrightarrow TERM_{\mathcal{L}}$ et σ' désigne une substitution de $TERM_{\mathcal{L}'} \longrightarrow TERM_{\mathcal{L}'}$.

Lemmes :

4.3 $\forall \sigma$, $\exists \sigma'$ telle que si $t \in TERM_{\mathcal{L}}$ alors $(\sigma t)' = \sigma' t'$.

4.4 $\forall \sigma$, $\exists \sigma'$ telle que si A est une formule de $\mathcal{L}$ alors $(\sigma A)' = \sigma' A'$.

4.5 $\forall \sigma'$, $\exists \sigma$ telle que si r et s sont des termes de $\mathcal{L}$ tels que $s' = \sigma' r'$, alors $s = \sigma r$.

4.6 $\forall \sigma'$, $\exists \sigma$ telle que Si Q et A sont des formules de $\mathcal{L}$ telles que $Q' = \sigma' A'$, alors $Q = \sigma A$.

4.7 Si A est une formule de $\mathcal{L}$ et Q est une instance de A alors Q' est une instance de A'.

4.8 Si Q' et A' sont les représentations des formules A et Q et si Q' est une instance de A', alors Q est une instance de A.

4.9 Si Q et A sont des formules de $\mathcal{L}$ alors :
Q est une instance de A `ssi` Q' est une instance de A'.

Elimination des instances "impropres"

Soient t_f la représentation d'un terme de $TERM_{\mathcal{L}}$, et t_p la représentation d'un atome de $ATOM_{\mathcal{L}}$. t_f et t_p étant fixés,
soit $\varphi : TERM_{\mathcal{L}'} \rightarrow TERM_{\mathcal{L}'}$ définie par :

- $\varphi(empty) = empty$
- $\varphi(t_1 \,\&\, t_2) = \varphi_p(t_1) \,\&\, \varphi(t_2)$
- $\varphi(t) = \varphi_p(t)$ dans tous les autres cas

avec $\varphi_p : TERM_{\mathcal{L}'} \rightarrow TERM_{\mathcal{L}'}$ définie par :

- $\varphi_p(x') = x'$ si x' est la représentation de $x \in VAR_{\mathcal{L}}$
- $\varphi_p(p'(t_1, \cdots, t_n)) = p'(\varphi_f(t_1), \cdots, \varphi_f(t_n))$ si p' est la représentation de $p \in PRED_{\mathcal{L}}$
- $\varphi_p(t) = t_p$ dans tous les autres cas

et $\varphi_f : TERM_{\mathcal{L}'} \rightarrow TERM_{\mathcal{L}'}$ définie par :

- $\varphi_f(x') = x'$ si x' est la représentation de $x \in VAR_{\mathcal{L}}$
- $\varphi_f(f'(t_1, \cdots, t_n)) = f'(\varphi_f(t_1), \cdots, \varphi_f(t_n))$ si f' est la représentation de $f \in FONC_{\mathcal{L}}$
- $\varphi_f(t) = t_f$ dans tous les autres cas

Remarques :

1. Si $t \in TERM_{\mathcal{L}'}$ alors $\varphi(t) \in TERM_{\mathcal{L}'}$.
2. $\forall t \in TERM_{\mathcal{L}'}$, $\varphi(t)$ est la représentation d'un atome ou d'une conjonction d'atomes de $\mathcal{L}$.
3. Si t est la représentation d'un atome ou d'une conjonction d'atomes de $\mathcal{L}$, alors $\varphi(t) = t$.
4. Si t est la représentation d'un atome ou d'une conjonction d'atomes de $\mathcal{L}$, et si s est une instance de t alors $\varphi(s)$ est une instance de t.

Preuves :

1. Évident du fait de la définition de φ.
2. Immédiat, preuve par induction.
3. Immédiat, preuve par induction.
4. Preuve par induction sur les représentations des termes de $\mathcal{L}$, puis immédiat pour les représentations d'atome et conjonction d'atomes.

φ est donc une application qui laisse invariantes les représentations de formules de $\mathcal{L}$, et qui transforme les autres termes de $\mathcal{L}'$ pour qu'ils soient représentation d'une formule de $\mathcal{L}$.

Lemme 4.10 :

S'il existe un arbre de preuve de racine $solve(Q')$ avec $Q' = empty$ ou $Q' = p'(\cdots)$ ou $Q' = p'_1(\cdots)\& \cdots \& p'_m(\cdots)$, avec p', $p'_1, \cdots, p'_m$ représentations de symboles de prédicats de $\mathcal{L}$, alors il existe un arbre de preuve de racine $solve(\varphi(Q'))$ ayant même squelette et tel que tous les termes apparaissant dans les atomes étiquetant les noeuds de l'arbre soient des représentations d'atome ou de conjonction d'atomes de $\mathcal{L}$.

4.4 Preuves

4.4.1 Preuves des lemmes

Preuve du lemme 4.3 :

Soit σ une substitution sur $\mathcal{L}$, nous définissons $\sigma' : VAR_{\mathcal{L}'} \longrightarrow TERM_{\mathcal{L}'}$ par : $\sigma' x' = (\sigma x)'$. σ' s'étend à $TERM_{\mathcal{L}'} \longrightarrow TERM_{\mathcal{L}'}$ de la façon habituelle.

Si $t \in VAR_{\mathcal{L}}$ alors $t' \in VAR_{\mathcal{L}'}$ et par définition $(\sigma t)' = \sigma' t'$.

Supposons $t_1, \cdots, t_n \in TERM_{\mathcal{L}}$ tels que $(\sigma t_i)' = \sigma' t'_i$.
Si $t = f(t_1, \cdots, t_n)$ alors

$$
\begin{aligned}
(\sigma t)' &= (\sigma f(t_1, \cdots, t_n))' \\
(\sigma t)' &= (f(\sigma t_1, \cdots, \sigma t_n))' \\
(\sigma t)' &= f'((\sigma t_1)', \cdots, (\sigma t_n)') \\
(\sigma t)' &= f'(\sigma' t'_1, \cdots, \sigma' t'_n) \\
(\sigma t)' &= \sigma' f'(t'_1, \cdots, t'_n) \\
(\sigma t)' &= \sigma' t'
\end{aligned}
$$

Conclusion : $\forall\, t \in TERM_{\mathcal{L}}, \forall \sigma, (\sigma t)' = \sigma' t'$.

Preuve du lemme 4.4 :

Nous reprenons l'application $\sigma \longrightarrow \sigma'$ définie dans la preuve du lemme 4.3.

Si $A = p(t_1, \cdots, t_n)$ alors

$$\begin{aligned}
(\sigma A)' &= (\sigma p(t_1, \cdots, t_n))' \\
(\sigma A)' &= (p(\sigma t_1, \cdots, \sigma t_n))' \\
(\sigma A)' &= p'((\sigma t_1)', \cdots, (\sigma t_n)') \\
(\sigma A)' &= p'(\sigma' t_1', \cdots, \sigma' t_n') \\
(\sigma A)' &= \sigma' p'(t_1', \cdots, t_n') \\
(\sigma A)' &= \sigma' A'
\end{aligned}$$

Si $A = A_1 \wedge A_2$ alors

$$\begin{aligned}
(\sigma A)' &= (\sigma(A_1 \wedge A_2))' \\
(\sigma A)' &= (\sigma A_1 \wedge \sigma A_2)' \\
(\sigma A)' &= (\sigma A_1)' \,\&\, (\sigma A_2)' \\
(\sigma A)' &= \sigma' A_1' \,\&\, \sigma' A_2' \\
(\sigma A)' &= \sigma'(A_1' \,\&\, A_2') \\
(\sigma A)' &= \sigma' A'
\end{aligned}$$

Si $A = A_1 \leftarrow A_2$ alors

$$\begin{aligned}
(\sigma A)' &= (\sigma(A_1 \leftarrow A_2))' \\
(\sigma A)' &= (\sigma A_1 \leftarrow \sigma A_2)' \\
(\sigma A)' &= (\sigma A_1)' \, if \, (\sigma A_2)' \\
(\sigma A)' &= \sigma' A_1' \, if \, \sigma' A_2' \\
(\sigma A)' &= \sigma'(A_1' \, if \, A_2') \\
(\sigma A)' &= \sigma' A'
\end{aligned}$$

Conclusion : $\forall A$ formule de $\mathcal{L}$, $\forall \sigma$, $(\sigma A)' = \sigma' A'$.

Preuve du lemme 4.5 :

Soit r et $s \in TERM_{\mathcal{L}}$, tels que s' est une instance de r', donc $s' = \beta r'$.
Une simple preuve par induction montre qu'il existe σ' telle que $s' = \sigma' r'$ et $\forall x' \in VAR_{\mathcal{L}'}$ $\sigma' x'$ est la représentation d'un terme de $\mathcal{L}$. Seules ces substitutions seront utilisées dans la suite.

Soit σ définie par $\sigma x = t$ si $t' = \sigma' x'$.

Si $r' \in VAR_{\mathcal{L}'}$ et $s' = \sigma' r'$ alors $s = \sigma r$ par définition de σ.

Soit $r_1, \cdots, r_n$ et $s_1, \cdots, s_n \in TERM_{\mathcal{L}'}$ tels que $s_i' = \sigma' r_i' \Longrightarrow s_i = \sigma r_i$.
Si $r = f(r_1, \cdots, r_n)$ et $s' = f'(s_1', \cdots, s_n') = \sigma' r'$ alors

$$\begin{aligned}
\sigma r &= \sigma f(r_1, \cdots, r_n) \\
\sigma r &= f(\sigma r_1, \cdots, \sigma r_n) \\
\sigma r &= f(s_1, \cdots, s_n) \\
\sigma r &= s
\end{aligned}$$

Conclusion : Si r et s sont deux termes de $TERM_{\mathcal{L}}$ tels que $s' = \sigma' r'$,

alors $s = \sigma r$.

Preuve du lemme 4.6 :

Nous reprenons l'application $\sigma' \longrightarrow \sigma$ définie dans la preuve du lemme 4.5.

Si $A = p(r_1, \cdots, r_n)$ et $Q' = \sigma' A' = p'(s'_1, \cdots, s'_n)$ alors

$$\begin{aligned} \sigma A &= \sigma p(r_1, \cdots, r_n) \\ \sigma A &= p(\sigma r_1, \cdots, \sigma r_n) \\ \sigma A &= p(s_1, \cdots, s_n) \\ \sigma A &= Q \end{aligned}$$

Si $A = A_1 \wedge A_2$ et $Q' = Q'_1 \,\&\, Q'_2 = \sigma' A'$
avec A'_1 et A'_2 tels que $\sigma' A'_i = Q'_i \Longrightarrow \sigma A_i = Q_i$ alors

$$\begin{aligned} \sigma A &= \sigma(A_1 \wedge A_2) \\ \sigma A &= \sigma A_1 \wedge \sigma A_2 \\ \sigma A &= Q_1 \wedge Q_2 \\ \sigma A &= Q \end{aligned}$$

Si $A = A_1 \leftarrow A_2$ et $Q' = Q'_1 \, if \, Q'_2 = \sigma' A'$
avec A'_1 et A'_2 tels que $\sigma' A'_i = Q'_i \Longrightarrow \sigma A_i = Q_i$ alors

$$\begin{aligned} \sigma A &= \sigma(A_1 \leftarrow A_2) \\ \sigma A &= \sigma A_1 \leftarrow \sigma A_2 \\ \sigma A &= Q_1 \leftarrow Q_2 \\ \sigma A &= Q \end{aligned}$$

Conclusion : Si A et Q sont deux formules $\mathcal{L}$ telles que $Q' = \sigma' A'$, alors $Q = \sigma A$.

Preuve du lemme 4.7 :

Conséquence immédiate du lemme 4.4.

Preuve du lemme 4.8 :

Conséquence immédiate du lemme 4.6.

Preuve du lemme 4.9 :

Conséquence des lemmes 4.7 et 4.8.

Preuve du lemme 4 .10:

- Soit un arbre de preuve de hauteur 0 et de racine $solve(Q')$. Cet arbre est :

$$solve(empty)$$

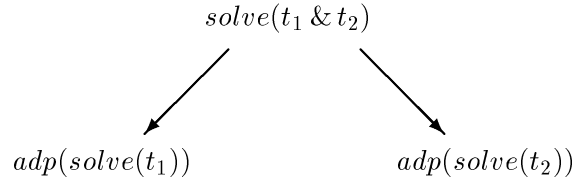
or $\varphi(empty) = empty$ donc

$$solve(empty)$$

est un arbre de preuve de racine $solve(\varphi(Q'))$ et de même squelette.

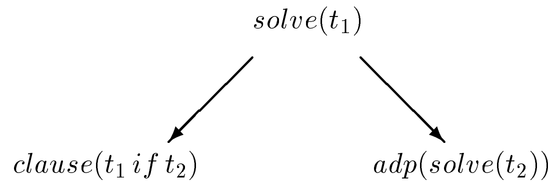
- Supposons que la propriété soit vraie pour tout arbre de preuve de profondeur inférieure à n.
  Soit un arbre de preuve de hauteur $n+1$, de racine $solve(Q')$ avec Q' de la forme voulue.
  Nous distinguerons deux cas :

  – l'arbre est de la forme :



  t_1 par hypothèse sur les racines étudiées est de la forme $p'(\cdots)$ avec p' représentation de $p \in PRED_{\mathcal{L}}$, donc $\varphi(t_1) = \varphi_p(t_1)$.
  Il existe par hypothèse un arbre de preuve de racine $solve(\varphi(t_1))$ et un autre de racine $solve(\varphi(t_2))$, donc il existe un arbre de preuve de racine $solve(\varphi(t_1)\,\&\,\varphi(t_2))$ et de même squelette.
  Or $\varphi(t_1) = \varphi_p(t_1)$ donc $\varphi(t_1)\,\&\,\varphi(t_2) = \varphi_p(t_1)\,\&\,\varphi(t_2) = \varphi(t_1\,\&\,t_2)$.
  Donc il existe un arbre de preuve de racine $solve(\varphi(Q'))$, et de même squelette.

  – l'arbre est de la forme :



  Par hypothèse il existe un arbre de preuve de racine $solve(\varphi(t_2))$ car $t_1\,if\,t_2$ est une instance d'une représentation d'une clause de P donc t_2 est $empty$, $p'(\cdots)$ ou $p'_1(\cdots)\,\&\cdots\&\,p_m(\cdots)$. Si $t_1\,if\,t_2$ est instance de $A'\,if\,E'$ représentation d'une clause $A \leftarrow E$ de P alors $\varphi(t_1)\,if\,\varphi(t_2)$ est instance de $A'\,if\,E'$.
  Donc il existe un arbre de preuve de racine $solve(\varphi(t_1))$ et de même squelette.

Conclusion : S'il existe un arbre de preuve de racine $solve(Q')$ alors il existe un arbre de preuve de racine $solve(\varphi(Q'))$, de même squelette, dans lequel tous les atomes de la forme $solve(B')$ sont tels que B' est la représentation d'une conjonction d'atome (éventuellement vide) de $\mathcal{L}$.

4.4.2 Preuve du théorème 4.1

$\Longrightarrow$

Montrons que si $Q \in pppf(T_P)$ alors $solve(Q') \in pppf(T_{\mathbf{V}_P})$.

Rappels : $pppf(T_P) = \bigcup_{n \in I\!N} T_P \uparrow n$.

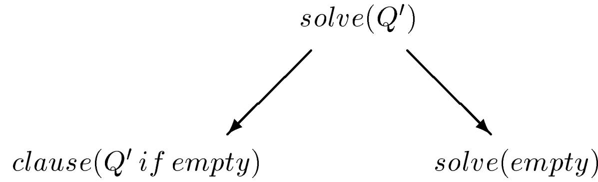
Nous montrons donc que $\forall n$, $[\, Q \in T_P \uparrow n \Longrightarrow solve(Q') \in pppf(T_{\mathbf{V}_P})\,]$.

- **cas** $n = 0$
  $T_P \uparrow 0 = \emptyset$ donc $Q \in T_P \uparrow 0 \Longrightarrow solve(Q') \in pppf(T_{\mathbf{V}_P})$.

- **récurrence**
  Supposons que $\forall i < n$, $Q \in T_P \uparrow i \Longrightarrow solve(Q') \in pppf(T_{\mathbf{V}_P})$.

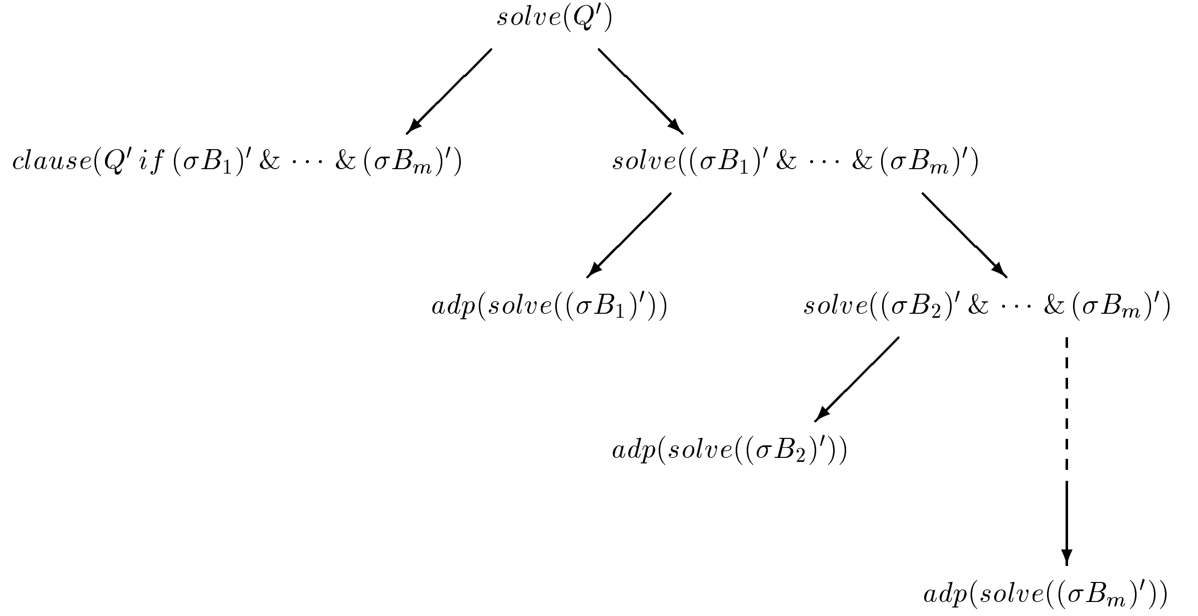
  Si $Q \in T_P \uparrow n$ alors $\exists\,(A \leftarrow E) \in P$ et $\exists\,\sigma$ une substitution,
  telles que $\sigma A = Q$ et $\sigma E \subseteq T_P \uparrow n - 1$.

  Si $E = true$
  alors $(A \leftarrow) \in P$ donc $(clause(A'\,if\,empty) \leftarrow) \in \mathbf{V}_P$.
  Si Q est instance de A alors Q' est instance de A' (d'après le lemme 4.9).
  Donc :

  est un arbre de preuve de $solve(Q')$.
  Or $RAP_{\mathbf{V}_P} = pppf(T_{\mathbf{V}_P})$ donc $solve(Q') \in pppf(T_{\mathbf{V}_P})$.

  Si $E = B_1 \wedge \cdots \wedge B_m$
  $\sigma B_i \in T_P \uparrow n - 1$ donc $solve((\sigma B_i)') \in pppf(T_{\mathbf{V}_P})$.
  Si $(A \leftarrow E) \in P$ alors $(clause(A'\,if\,B_1{}'\,\&\cdots\&\,B_m{}') \leftarrow) \in \mathbf{V}_P$,
  $Q'\,if\,(\sigma B_1)'\,\&\cdots\&\,(\sigma B_m)'$ est instance de $A'\,if\,B_1'\,\&\cdots\&\,B_m'$ d'après le lemme 4.9 $(Q' = (\sigma A)')$.
  $A \leftarrow E$ est une clause donc les B_i sont en nombre fini.
  Soit $adp(solve((\sigma B_i)'))$ l'arbre de preuve de $solve((\sigma B_i)')$.



est un arbre de preuve de $solve(Q')$.
Par conséquent, $solve(Q') \in pppf(T_{\mathbf{V}_P})$.

Conclusion :

$\forall n, [\, Q \in T_P \uparrow n \Longrightarrow solve(Q') \in pppf(T_{\mathbf{V}_P})\,]$.

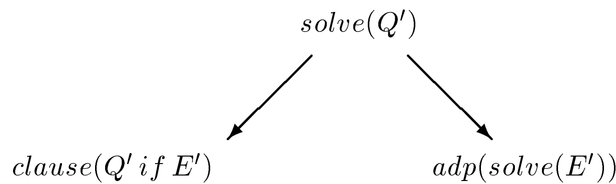
Donc $Q \in pppf(T_P) \Longrightarrow solve(Q') \in pppf(T_{\mathbf{V}_P})$.

$\Longleftarrow$

Montrons que si $solve(Q') \in pppf(T_{\mathbf{V}_P})$ alors $Q \in pppf(T_P)$.

Nous montrons (comme pour $\Longrightarrow$) que :
$\forall n, [\, solve(Q') \in T_{\mathbf{V}_P} \uparrow n \Longrightarrow Q \in pppf(T_P)\,]$.

- **cas** $n = 0$
  $T_{\mathbf{V}_P} \uparrow 0 = \emptyset$ donc $solve(Q') \in T_{\mathbf{V}_P} \uparrow 0 \Longrightarrow Q \in pppf(T_P)$.
- **récurrence**
  Supposons que $\forall i < n$, $solve(Q') \in T_{\mathbf{V}_P} \uparrow i \Longrightarrow Q \in pppf(T_P)$.
  Si $solve(Q') \in T_{\mathbf{V}_P} \uparrow n$ alors il existe un arbre de preuve de racine $solve(Q')$.
  Cet arbre est de la forme :



(Les deux premières clauses pour *solve* ne nous intéressent pas car Q' est la représentation d'un atome de $\mathcal{L}$)

Nous distinguerons deux cas :

Si $E' = empty$

alors :

$$solve(Q')$$

$$clause(Q' \, if \, empty) \qquad solve(empty)$$

est un arbre de preuve de $solve(Q')$.
$clause(Q' \, if \, empty) \leftarrow$ est une instance de $(clause(A' \, if \, empty) \leftarrow) \in \mathbf{V}_P$ donc $(A \leftarrow) \in P$.
Si Q' est une instance de A' alors Q est une instance de A d'après le lemme 4.9, et donc $Q \in pppf(T_P)$.

Si $E' = B'_1 \,\&\, \cdots \,\&\, B'_m$

alors :

$$solve(Q')$$

$$clause(Q' \, if \, B'_1 \,\&\, \cdots \,\&\, B'_m) \qquad solve(B'_1 \,\&\, \cdots \,\&\, B'_m)$$

$$adp(solve(B'_1)) \qquad solve(B'_2 \,\&\, \cdots \,\&\, B'_m)$$

$$adp(solve(B'_2))$$

$$adp(solve(B'_m))$$

est un arbre de preuve de $solve(Q')$.
Pour pouvoir appliquer l'hypothèse d'induction, il faut montrer qu'il existe un arbre de preuve de racine $solve(Q')$ identique à celui ci-dessus (de même hauteur), tel que B'_i est la représentation d'un $B_i \in ATOM_{\mathcal{L}}$.

D'après le lemme 4.10, il existe un arbre de preuve de racine $solve(\varphi(Q'))$ ayant même squelette que celui ci-dessus, avec tous les B'_i représentation de $B_i \in ATOM_{\mathcal{L}}$, or Q' est la représentation de $Q \in ATOM_{\mathcal{L}}$, donc $\varphi(Q') = Q'$.

Nous savons donc qu'il existe un arbre de preuve de racine $solve(Q')$, identique à celui représenté, et tel que tous les B'_i sont la représentation d'un $B_i \in ATOM_{\mathcal{L}}$.

$\forall\, i,\ adp(solve(B_i'))$ a une hauteur inférieure à n,
donc $solve(B_i') \in T_{\mathbf{V}_P} \uparrow n-1$ et par hypothèse $B_i \in pppf(T_P)$.
Si $clause(Q'\ if\ B_1'\ \&\cdots\&\ B_m') \leftarrow$ est une instance de
$(clause(A'\ if\ E') \leftarrow) \in \mathbf{V}_P$ alors $(A \leftarrow E) \in P$
et, d'après le lemme 4.9, $Q \leftarrow B_1 \wedge \cdots \wedge B_m$ est une instance de $A \leftarrow E$,
donc $Q \in pppf(T_P)$.

Conclusion :

$\forall\, n,\ [\, solve(Q') \in T_{\mathbf{V}_P} \uparrow n \Longrightarrow Q \in pppf(T_P)\,]$.

Donc $solve(Q') \in pppf(T_{\mathbf{V}_P}) \Longrightarrow Q \in pppf(T_P)$.

$\Longleftrightarrow$

Nous pouvons conclure :

Pour tout $Q \in ATOM_{\mathcal{L}}$,
$Q \in pppf(T_P)$ `ssi` $solve(Q') \in pppf(T_{\mathbf{V}_P})$.

4.5 Comparaison avec Hill et Lloyd, et Conclusion

D'après les théorèmes 8.4 et 8.6, ainsi que le lemme 8.2 de [9] et de la définition de réponse calculée (ou de résolution restreinte aux unificateurs les plus généraux), nous avons montré que :

Théorème 4.11 :
$\{x_1/t_1, \cdots, x_n/t_n\}$ est une réponse calculée pour $P \cup \{Q \leftarrow\}$ `ssi`
$\{x_1'/t_1', \cdots, x_n'/t_n'\}$ est une réponse calculée pour $\mathbf{V}_P \cup \{solve(Q') \leftarrow\}$.

Ce résultat rejoint celui de Lloyd dans [8], mais il a été obtenu ici sans typage. En fait, le théorème est rigoureusement identique dans son énoncé au théorème 2.3.3 (1) de [8]. Cependant il est plus général puisque, non seulement, le langage $\mathcal{L}'$ de Lloyd est inclus dans celui que nous avons défini, mais aussi la sémantique de son méta-interprète "est incluse" dans la nôtre. Nous savons qu'il existe des programmes objets tels que $solve(Q')$ est racine d'arbre de preuve, et Q' n'est pas la représentation d'une formule de $\mathcal{L}$. En d'autres termes, le méta-interprète de Lloyd est correct, le nôtre ne l'est pas, mais le théorème reste valide.

Nous avons montré (sans typage) que le méta-interprète *vanilla* est correct et complet pour des buts de la forme $solve(Q') \leftarrow$ où Q' est la représentation d'un atome Q du langage $\mathcal{L}$.

Soit $SUBST_{\mathcal{L}}$ l'ensemble des substitutions $VAR_{\mathcal{L}} \to TERM_{\mathcal{L}}$.
Soit $\psi : SUBST_{\mathcal{L}} \to SUBST_{\mathcal{L}'}$ définie par : $\forall\, \sigma,\ \psi(\sigma)(x') = (\sigma(x))'$

Remarque évidente : si σ est un unificateur le plus général de A et B formules de $\mathcal{L}$ alors $\psi(\sigma)$ est un unificateur le plus général de A' et B'.

D'après le lemme 4.9 et les remarques précédant le lemme 4.10, si Q est l'instance commune la plus générale de A et B alors Q' est l'instance commune la plus

générale de A' et B'. Nous constatons rapidement que Q' est instance de A' et B' par $\psi(\sigma)$ et que cette substitution est un unificateur le plus général de A' et B'.

Théorème 4.12 :
Si $P \cup \{\leftarrow Q\}$ réussit avec la réponse σ alors $\mathbf{V}_P \cup \{\leftarrow solve(Q')\}$ réussit avec la réponse $\psi(\sigma)$.

preuve :
Le théorème 4.12 est conséquence immédiate du théorème 4.11 et de la remarque précédente.

La représentation implicite permet d'obtenir des résultats tout à fait satisfaisants pour les méta-programmes style *vanilla*. Mais elle est limitée.

Dans le cas d'un débogueur, l'instance d'une erreur pour un programme P n'est pas toujours une erreur pour ce programme. Aussi, il est impossible d'utiliser la représentation implicite pour représenter le programme objet et ses erreurs.

Dans le chapitre 5 nous étudions une autre représentation du programme objet qui permettra de déclarer une sémantique contenant des représentations de formules du langage objet non fermées par substitution (dans le langage objet, la sémantique d'un programme logique est, bien entendu, toujours fermée par substitution).

Chapitre 5

Représentation close

5.1 Motivations

Dans la section 2.3, nous avons étudié la représentation close typée et le méta-interprète de Lloyd. Nous avons expliqué dans la section 2.3.5 que le typage était ici complètement inutile. Donc, contrairement à la représentation implicite, le méta-interprète sans typage reste complet et correct.

Lloyd vise un aspect opérationnel puisque, pour lui, $solve(Q', R')$ est conséquence logique de $comp(\mathbf{G}_P)$ `ssi` R est conséquence logique de $comp(P)$, et R est une instance de Q. Ou, en d'autres termes, $solve(Q', R')$ est tel que si Q' est un but alors R' est la représentation d'une réponse correcte pour $P \cup \{\leftarrow Q\}$.

Nous n'attachons pas d'importance ici aux réponses pour un but donné, mais nous désirons traduire la sémantique suivante :

$$solve(Q') \in pppf(T_{\mathbf{G}_P}) \texttt{ ssi } Q \in pppf(T_P).$$

Aussi, l'intérêt est déclaratif et non opérationnel, puisque si $Q \notin pppf(T_P)$ a pour instance $R \in pppf(T_P)$ alors $solve(Q')$ échoue (Q' est clos en représentation close). Ou, en d'autres termes, si $R \in pppf(T_P)$ et R est une instance de Q alors jamais $solve(Q')$ ne réussit avec réponse σ' telle que $\sigma' Q' = R'$. Mais nous discuterons ce point dans la section 5.5.

Nous n'étudierons donc pas le méta-interprète de la figure 2.2 page 16, mais dans un esprit d'homogénéité nous reprenons *vanilla*, avec cette fois une représentation close du programme objet.

Nous montrons que *vanilla* est, dans ce cas, correct et complet, ce qui n'est pas aussi évident que sa complétude et sa correction relative en représentation implicite.

5.2 Représentation close du programme P

Soit $\mathcal{L}$ un langage muni des connecteurs $\leftarrow$, $\wedge$ et *true*. Passage de $\mathcal{L}$ à $\mathcal{L}'$:

$$a \text{ constante de } \mathcal{L} \quad \longleftrightarrow \quad a' \text{ constante de } \mathcal{L}'$$

$$x \text{ variable de } \mathcal{L} \quad \longleftrightarrow \quad x' \text{ constante de } \mathcal{L}'$$

$$f \text{ fonction n-aire de } \mathcal{L} \quad \longleftrightarrow \quad f' \text{ fonction n-aire de } \mathcal{L}'$$

$$p \text{ prédicat n-aire de } \mathcal{L} \quad \longleftrightarrow \quad p' \text{ fonction n-aire de } \mathcal{L}'$$

Nous supposons que les applications $a \to a'$, $x \to x'$, $f \to f'$ et $p \to p'$ sont toutes bijectives.

Nous ajoutons au langage $\mathcal{L}'$ une constante : *empty*, les symboles de fonctions : & et if. Le langage $\mathcal{L}'$ contient les deux symboles de prédicats *solve* et *clause*.

Le passage des termes, atomes et formules du langage $\mathcal{L}$ dans le langage $\mathcal{L}'$, est défini par induction.

Si $f(t_1, \cdots, t_n)$ est un terme de $\mathcal{L}$, alors $f(t_1, \cdots, t_n)$ est représenté par le terme $f'(t'_1, \cdots, t'_n)$ de $\mathcal{L}'$, où $t'_1, \cdots, t'_n$ sont les représentations respectives de $t_1, \cdots, t_n$.

Si $p(t_1, \cdots, t_n)$ est un atome de $\mathcal{L}$, alors $p(t_1, \cdots, t_n)$ est représenté par le terme $p'(t'_1, \cdots, t'_n)$ de $\mathcal{L}'$, où $t'_1, \cdots, t'_n$ sont les représentations respectives de $t_1, \cdots, t_n$.

true est représenté par *empty*.

Si F et G sont des formules de $\mathcal{L}$ représentées par F' et G' dans $\mathcal{L}'$ alors $F \wedge G$ et $F \leftarrow G$ sont représentées respectivement par les termes $F' \,\&\, G'$ et $F' \, if \, G'$.

Soit **G** le programme normal, basé sur $\mathcal{L}'$, suivant :

$$solve(empty) \leftarrow$$
$$solve(x \,\&\, y) \leftarrow solve(x) \wedge solve(y)$$
$$solve(x) \leftarrow clause(x \, if \, y) \wedge solve(y)$$

Si P est un programme défini basé sur le langage $\mathcal{L}$ alors nous notons $\mathbf{G}_P$ le programme constitué de **G** et des clauses : $clause(A' \, if \, Q') \leftarrow$, pour chaque instance d'une clause $A \leftarrow Q$ du programme P ($Q' = empty$ si Q est vide).

Remarque : Nous notons X' pour dénoter la représentation d'un objet X du langage $\mathcal{L}$ dans le langage $\mathcal{L}'$ (même convention de notation que dans le chapitre 4).

5.3 Théorèmes

Théorèmes :

5.1 Pour tout $Q \in ATOM_{\mathcal{L}}$: $Q \in pppf(T_P)$ ssi $solve(Q') \in pppf(T_{\mathbf{G}_P})$

5.2 $\mathbf{G}_P$ est un méta-interprète correct et complet.

5.4 Preuves

5.4.1 Preuve du théorème 5.1

$\Longrightarrow$

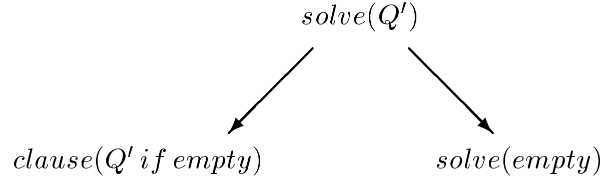
Montrons que si $Q \in pppf(T_P)$ alors $solve(Q') \in pppf(T_{\mathbf{G}_P})$.
Montrons par récurrence sur n que $\forall n$, $[\, Q \in T_P \uparrow n \Longrightarrow solve(Q') \in pppf(T_{\mathbf{G}_P})\,]$.

- **cas** $n = 0$
  $T_P \uparrow 0 = \emptyset$ donc $Q \in T_P \uparrow 0 \Longrightarrow solve(Q') \in pppf(T_{\mathbf{G}_P})$.

- **récurrence**
  Supposons que $\forall i < n$, $Q \in T_P \uparrow i \Longrightarrow solve(Q') \in pppf(T_{\mathbf{G}_P})$.

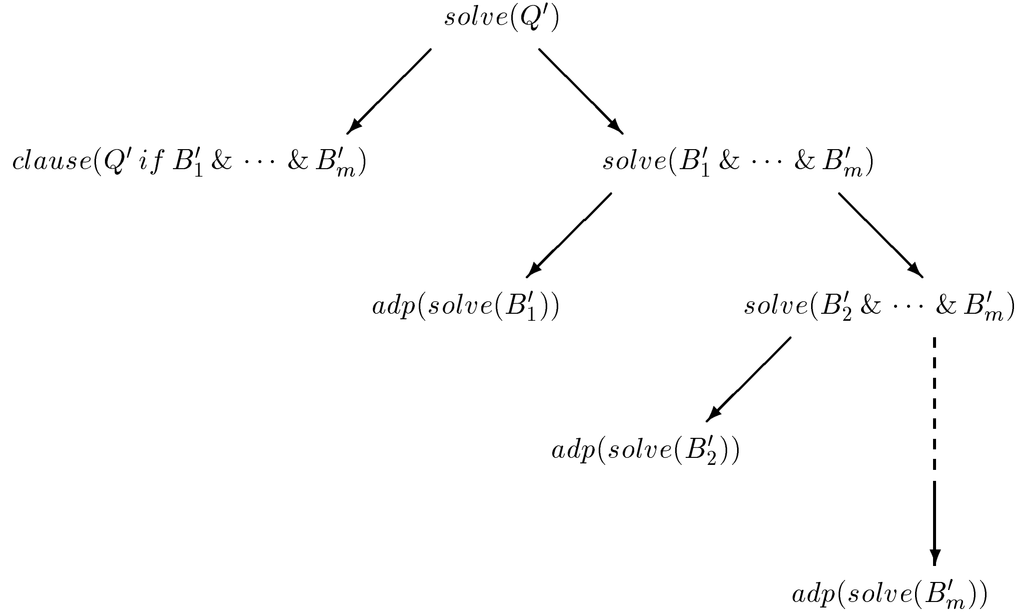
  Si $Q \in T_P \uparrow n$ alors il existe $Q \leftarrow E$ instance d'une clause de P telle que $E = true$ ou $E \subseteq T_P \uparrow n - 1$.

  Si $E = true$
  $Q \leftarrow$ est instance d'une clause de P donc $(clause(Q'\, if\, empty) \leftarrow) \in \mathbf{G}_P$.



  est un arbre de preuve de $solve(Q')$, et par conséquent $solve(Q') \in pppf(T_{\mathbf{G}_P})$.

  Si $E = B_1 \wedge \cdots \wedge B_m$
  $solve(B'_i) \in pppf(T_{\mathbf{G}_P})$ par hypothèse donc il existe un arbre de preuve de racine $solve(B'_i)$, noté $adp(solve(B'_i))$.
  $Q \leftarrow E$ est instance d'une clause de P donc $clause(Q'\, if\, E') \leftarrow \in \mathbf{G}_P$.



est un arbre de preuve de $solve(Q')$ et par conséquent $solve(Q') \in pppf(T_{\mathbf{G}_P})$.

Conclusion :

$\forall n, [\, Q \in T_P \uparrow n \Longrightarrow solve(Q') \in pppf(T_{\mathbf{G}_P})\,]$.

Donc $Q \in pppf(T_P) \Longrightarrow solve(Q') \in pppf(T_{\mathbf{G}_P})$.

$\Longleftarrow$

Montrons à l'aide d'une spécification[1] inductive que si $solve(Q') \in pppf(T_{\mathbf{G}_P})$ alors $Q \in pppf(T_P)$.

Rappels sur la correction partielle relativement à une spécification.
Un programme P est dit partiellement correct pour une spécification $\mathcal{S}$, ou $\mathcal{S}$ est dite valide pour P si et seulement si pour tout atome de la dénotation de P de la forme $p(t_1, \cdots, t_n)$, la formule $\mathcal{S}^p[t_1, \cdots, t_n]$ est valide.

Une spécification $\mathcal{S}$ est dite inductive pour le programme logique P si et seulement si elle vérifie la condition :
pour toute clause de P de la forme $p_0(t_0) \leftarrow p_1(t_1) \wedge \cdots \wedge p_n(t_n)$
la formule $\mathcal{S}^{p_1}[t_1] \wedge \cdots \wedge \mathcal{S}^{p_n}[t_n] \Rightarrow \mathcal{S}^{p_0}[t_0]$ est valide.

Théorème :

Une spécification $\mathcal{S}$ est valide pour P si et seulement si elle est conséquence logique d'une spécification inductive.

Soit la spécification $\mathcal{S} = \{\mathcal{S}^{solve}, \mathcal{S}^{clause}\}$ définie par :

$$\mathcal{S}^{solve} = radp(solve1)$$

[1] Voir [4] pour un complément sur les spécifications.

$$\mathcal{S}^{clause} = inst(clause1)$$

Avec $radp(x)$ vrai si $x = Q'$ est représentation d'une conjonction d'atomes telle que $Q \subseteq pppf(T_P)$, ou x est représentation d'un atome $Q \in pppf(T_P)$, ou x est *empty*; et $inst(x)$ vraie si x est la représentation d'une instance d'une clause de P.

Montrons que $\mathcal{S}$ est inductive pour $\mathbf{G}_P$:

- **1° clause :** $solve(empty) \leftarrow$
  $vrai \Rightarrow radp(empty)$ est valide par définition de $radp$.
- **2° clause :** $solve(x \,\&\, y) \leftarrow solve(x) \wedge solve(y)$
  $radp(x) \wedge radp(y) \Rightarrow radp(x \,\&\, y)$ est valide par définition de $radp$.
- **3° clause :** $solve(x) \leftarrow clause(x\, if\, y) \wedge solve(y)$
  $radp(y) \wedge inst(x\, if\, y) \Rightarrow radp(x)$
  Si $radp(y)$ est valide alors y est représentation de $E \subseteq T_P \uparrow j$ pour un certain j.
  Si $inst(x\, if\, y)$ alors $x\, if\, y$ est la représentation de $Q \leftarrow E$ qui est une instance d'une clause de P.
  Par définition de T_P, $Q \in T_P \uparrow j + 1$ donc $Q \in pppf(T_P)$.
- **autres clauses :** $clause(Q'\, if\, A')$
  $inst(Q'\, if\, A')$ est valide par définition du prédicat *clause* de $\mathbf{G}_P$

La spécification est inductive, donc elle est valide.
Si $solve(Q') \in pppf(\mathbf{G}_P)$ alors Q' est *empty* ou Q' est représentation d'une conjonction d'atomes Q telle que $Q \subseteq pppf(T_P)$ ou Q' est représentation d'un atome $Q \in pppf(T_P)$.

Conclusion :

$$solve(Q') \in pppf(T_{\mathbf{G}_P}) \Longrightarrow Q \in pppf(T_P).$$

$\Longleftarrow$

Nous pouvons conclure :

Pour tout $Q \in ATOM_{\mathcal{L}}$,
$Q \in pppf(T_P)$ `ssi` $solve(Q') \in pppf(T_{\mathbf{G}_P})$.

5.4.2 Preuve du théorème 5.2

Nous savons par le théorème 5.1 que $\mathbf{G}_P$ est correct. Montrons par induction sur les arbres de preuve qu'il est complet.

Nous montrons que $solve(Q') \in pppf(T_{\mathbf{G}_P})$ `ssi` Q' est *empty* ou Q' est représentation d'un atome ou Q' est représentation d'une conjonction d'atomes.

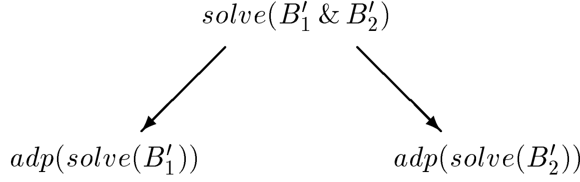
- **cas** $n = 0$
  $T_{\mathbf{G}_P} \uparrow 0 = \emptyset$ donc $\forall\, solve(Q') \in T_{\mathbf{G}_P} \uparrow 0$ $solve(Q')$ vérifie l'hypothèse.

- **récurrence**
  Supposons l'hypothèse vraie pour tout $solve(Q') \in T_{\mathbf{G}_P} \uparrow i$, $i < n$. Montrons qu'elle est vraie pour tout $solve(Q') \in T_{\mathbf{G}_P} \uparrow n$.
  Si $solve(Q') \in T_{\mathbf{G}_P} \uparrow n$ alors nous distinguerons trois possibilités correspondant aux trois clauses définissant $solve$.

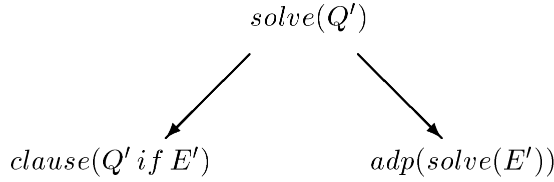
  1. l'arbre de preuve de $solve(Q')$ est :

     $solve(empty)$

     donc $solve(Q')$ vérifie l'hypothèse.
  2. l'arbre de preuve de $solve(Q')$ est :

     avec $solve(B'_1)$ et $solve(B'_2) \in T_{\mathbf{G}_P} \uparrow n-1$ donc vérifiant l'hypothèse.
     Donc $B'_1 \,\&\, B'_2$ est représentation d'une conjonction d'atomes.
  3. l'arbre de preuve de $solve(Q')$ est :

     $Q'\,if\,E'$ est représentation de $Q \leftarrow E$ instance d'une clause de P par définition du predicat $clause$ (rappelons que $Q'\,if\,E'$ est un terme clos).
     Donc Q' est représentation d'un atome Q.

Conclusion :

$solve(Q') \in pppf(T_{\mathbf{G}_P})$ `ssi` Q' est $empty$ ou Q' est représentation d'un atome ou d'une conjonction d'atomes.

Remarque : Nous avons supposé que $\wedge$ est associatif, c'est à dire que $(B'_1 \,\&\, B'_2) \,\&\, B'_3$ est représentation de $(B_1 \wedge B_2) \wedge B_3 = B_1 \wedge (B_2 \wedge B_3)$.

Le méta-interprète $\mathbf{G}_P$ est correct et complet.

5.5 Conclusion

Nous constatons que la correction de $\mathbf{G}_P$ permet d'avoir des preuves plus courtes (théorème 5.1, partie $\Longleftarrow$, par rapport au théorème 4.1). Nous avons pu réduire également la taille de la partie $\Longrightarrow$ par la définition de *clause* :

$clause(C')$ `ssi` C' représentation d'une instance d'une clause de P

par rapport à la représentation implicite où la définition de *clause* est :

$clause(C')$ `ssi` C' représentation d'une clause de P.

Nous aurions pu simplifier de la même manière la preuve en représentation implicite, en prenant la même définition pour *clause*, mais pourquoi avoir un programme infini si cela n'est pas nécessaire ...

Nous n'avons pas eu besoin des lemmes du chapitre 4 ici. Par ailleurs, ils sont tous faux en représentation close. Si Q est instance de A et $Q \neq A$ alors Q' n'est pas instance de A' (Q' est clos).

Intéressons-nous à l'aspect opérationnel de $\mathbf{G}_P$ et répondons au problème soulevé dans la section 2.3.5. Nous annoncions que le méta-interprète de Lloyd avec le prédicat *solve* à deux arguments permettait d'obtenir en deuxième argument les réponses à un but en premier argument.

Par exemple si P est le programme :

$$p(x,a) \leftarrow$$
$$p(b,c) \leftarrow$$

le méta-interprète de Lloyd donne comme réponse au but $solve(p'(x',y'),X)$:

$X = p'(x',a')$
$X = p'(b',c')$

alors que pour $\mathbf{G}_P$ $solve(p'(x',y'))$ conduit à un échec.
En revanche, pour le but $solve(p'(X,Y))$, nous obtenons les réponses :

$X = x'$ $\quad Y = a'$
$X = b'$ $\quad Y = c'$

En remplaçant les représentations de variables par des variables dans le but, nous retrouvons l'effet opérationnel voulu.

Mais, $\mathbf{G}_P$ peut faire plus encore puisque pour le but $solve(p'(x',Y))$ s'affichera l'unique réponse :

$Y = a'$

C'est-à-dire que nous pouvons nous intéresser aux réponses pour un but, en contraignant ces réponses à contenir des variables. Ceci sera réalisé moins simplement avec le méta-programme de Lloyd.

D'un point de vue opérationnel, $\mathbf{G}_P$ traduit agréablement la sémantique déclarative de P. Dommage qu'il s'agisse d'un programme infini ...

Chapitre 6

Sémantique close, représentation implicite et représentation close

6.1 Introduction

Nous proposons, dans ce chapitre, d'étudier la variante du méta-interprète *vanilla* qui rend compte de la sémantique close du programme objet P. Elle sera notée $\mathbf{H}_P$.

Nous comparons ensuite $\mathbf{H}_P$ avec la sémantique close des méta-programmes $\mathbf{V}_P$ et $\mathbf{G}_P$.

Nous montrons également que la représentation des variables de $VAR_{\mathcal{L}}$ est sans importance et ne modifie pas la sémantique de $\mathbf{H}_P$.

Enfin, nous comparons représentation implicite et représentation close à travers les arbres de preuve de $\mathbf{V}_P$ et $\mathbf{G}_P$.

6.2 Le méta-programme $\mathbf{H}_P$

Supposons, provisoirement, que nous reprenons la représentation close pour le programme objet.

Soit $\mathbf{H}_P$ le programme constitué de $\mathbf{G}$ et des $clause(A' \, if \, Q') \leftarrow$ pour chaque instance close d'une clause $A \leftarrow Q$ du programme P ($Q' = empty$ si Q est vide).

6.3 Théorèmes

Soit $SEM_P^C = \{ \, Q \in pppf(T_P)/Q$ est clos $\}$

Théorème 6.1 :
$solve(Q') \in pppf(T_{\mathbf{H}_P})$ `ssi` $Q \in SEM_P^C$.

<u>**Remarques**</u> :

6.2 Il n'est plus nécessaire de décider d'une représention des variables pour $\mathbf{H}_P$.

6.3 $\mathbf{H}_P$ est aussi le méta-interprète utilisant la représentation implicite, tel que $solve(Q') \in pppf(T_{\mathbf{H}_P})$ `ssi` $Q \in SEM_P^C$.

6.4 Si Q' est la représentation d'un atome Q alors : $solve(Q') \in pppf(T_{\mathbf{H}_P})$ ssi $solve(Q') \in SEM^C_{\mathbf{V}_P}$

6.5 Il existe P tel que $pppf(T_{\mathbf{H}_P}) \neq SEM^C_{\mathbf{V}_P}$.

6.6 Il existe P tel que $pppf(T_{\mathbf{H}_P}) \neq SEM^C_{\mathbf{G}_P}$.

6.7 Il existe P tel que $SEM^C_{\mathbf{G}_P} \neq SEM^C_{\mathbf{V}_P}$.

6.8 Pour tout P, $SEM^C_{\mathbf{G}_P} = pppf(T_{\mathbf{G}_P})$.

Notons $\mathcal{L}^1$ le langage de la représentation implicite et $\mathcal{L}^2$ le langage de la représentation close. X^1 désigne la représentation implicite de X et X^2 désigne la représentation close de X.

Nous appelons arbre de preuve bien formé dans le langage $\mathcal{L}^1$, tout arbre de preuve tel que les atomes étiquetant ses nœuds sont bien formés. Nous notons $ATOMBF_{\mathcal{L}^1}$ l'ensemble des atomes bien formés. D'après le lemme 4.10 du chapitre 4, pour tout $Q \in ATOM_{\mathcal{L}}$, si $solve(Q^1) \in pppf(T_{\mathbf{V}_P})$ alors il existe un arbre de preuve bien formé de racine $solve(Q^1)$ et ayant même squelette.

Soit $\psi : TERM_{\mathcal{L}^1} \longrightarrow TERM_{\mathcal{L}^2}$ l'application définie par :

$\forall x \in VAR_{\mathcal{L}}$, $\psi(x^1) = x^2$

$\forall f \in FONC_{\mathcal{L}}$, f d'arité n et $\forall t_1 \cdots t_n \in TERM_{\mathcal{L}}$,

$\psi(f^1(t_1^1, \cdots, t_n^1)) = f^2(\psi(t_1^1), \cdots, \psi(t_n^1))$

$\forall p \in PRED_{\mathcal{L}}$, p d'arité n et $\forall t_1 \cdots t_n \in TERM_{\mathcal{L}}$,

$\psi(p^1(t_1^1, \cdots, t_n^1)) = p^2(\psi(t_1^1), \cdots, \psi(t_n^1))$

$\psi(empti^1) = empty^2$

$\forall t_1, t_2 \in TERM_{\mathcal{L}}$, $\psi(t_1^1 \,\&^1\, t_2^1) = \psi(t_1^1) \,\&^2\, \psi(t_2^1)$

$\forall t_1, t_2 \in TERM_{\mathcal{L}}$, $\psi(t_1^1 \, if^1 \, t_2^1) = \psi(t_1^1) \, if^2 \, \psi(t_2^1)$

ψ définit une bijection entre $VAR_{\mathcal{L}^1}$ et $VAR_{\mathcal{L}^2}$

ψ s'étend à $\psi : ATOMBF_{\mathcal{L}^1} \longrightarrow ATOM_{\mathcal{L}^2}$ par :

$\psi(solve^1(Q^1)) = solve^2(\psi(Q^1))$

$\psi(clause^1(Q^1)) = clause^2(\psi(Q^1))$

Remarques :

- ψ est bijective, notons ψ^{-1} l'application réciproque.
- Si Q^1 est la représentation implicite de Q et si Q^2 est la représentation close de Q alors $\psi(Q^1) = Q^2$.

$adp(X)$ désigne un arbre de preuve de X. Notons $\psi(adp(X))$, l'arbre obtenu en appliquant ψ à chaque atome étiquetant un nœud de l'arbre $adp(X)$.

Théorème 6.9 :
Soit $Q \in ATOM_{\mathcal{L}}$, si $adp(solve^1(Q^1))$ est un arbre de preuve bien formé de racine $solve^1(Q^1)$ pour $\mathbf{V}_P$ alors $\psi(adp(solve^1(Q^1)))$ est un arbre de preuve de racine $solve^2(Q^2)$ pour $\mathbf{G}_P$.

Théorème 6.10 :

Soit $Q \in ATOM_{\mathcal{L}}$, si $adp(solve^2(Q^2))$ est un arbre de preuve de racine $solve^2(Q^2)$ pour $\mathbf{G}_P$ alors $\psi^{-1}(adp(solve^2(Q^2)))$ est un arbre de preuve bien formé de racine $solve^1(Q^1)$ pour $\mathbf{V}_P$.

Corollaire :

6.11 ψ est une bijection entre les arbres de preuve bien formés de $\mathbf{V}_P$ et les arbres de preuve de $\mathbf{G}_P$.

6.12 Ces arbres ont le même squelette et leurs nœuds sont étiquetés par des atomes "syntaxiquement isomorphes" (par ψ).

6.4 Preuves

6.4.1 Preuve du théorème 6.1

La preuve est identique à la preuve du théorème 5.1 du chapitre 5, en remplaçant "instance" par "instance close", "atome" par "atome clos" et "$\mathbf{G}_P$" par "$\mathbf{H}_P$".

6.4.2 Justification des remarques

remarque 6.2 :

Dans la preuve du théorème 6.1, nous voyons qu'aucun arbre de preuve ne fait apparaître des représentations de variables. De manière informelle, nous pouvons dire que le prédicat *clause* constitue l'interface entre le programme objet et le méta-programme. Il est, en quelque sorte, le constructeur de termes représentations de formules du langage objet. Le prédicat *solve*, quant à lui, ne fait qu'utiliser ces objets construits. En aucun cas, si le prédicat *clause* ne contient de représentation de variables, nous ne pourrons voir dans des arbres de preuve, du méta-programme, des nœuds étiquetés par des atomes contenant des représentations de variables. Il n'est donc pas nécessaire de définir la représentation des variables.

remarque 6.3 :

Cette remarque découle immédiatement de ce qui vient d'être énoncé. Pour le programme $\mathbf{H}_P$ la représentation des variables objets est indépendante de sa sémantique.

remarque 6.4 :

Nous savons d'une part que $solve(Q') \in pppf(T_{\mathbf{V}_P})$ `ssi` $Q \in pppf(T_P)$ et que $solve(Q') \in pppf(T_{\mathbf{H}_P})$ `ssi` $Q \in SEM_P^C$. Nous savons d'autre part que Q est clos `ssi` Q' est clos. Donc, pout tout Q clos, $solve(Q') \in pppf(T_{\mathbf{V}_P})$, et pour tout Q non clos, $solve(Q')$ n'est pas clos. Donc pour tout $Q \in ATOM_{\mathcal{L}}$, $solve(Q') \in SEM^C_{\mathbf{V}_P}$ `ssi` $solve(Q') \in pppf(T_{\mathbf{H}_P})$.

remarque 6.5 :

Il suffit de constater que $pppf(T_{\mathbf{V}_P})$ peut contenir des atomes de la forme $solve(X)$, où X n'est pas représentation d'un atome de $\mathcal{L}$. Donc dans $SEM^C_{\mathbf{V}_P}$, nous trouverons des $solve(X)$ avec X non représentation d'un atome.

Pour tout atome $solve(Q') \in pppf(T_{\mathbf{H}_P})$, Q' est la représentation d'un atome racine d'arbre de preuve pour P.

Donc, en général, $pppf(T_{\mathbf{H}_P}) \neq SEM^C_{\mathbf{V}_P}$.

remarque 6.6 :

Nous avons vu que $solve(Q') \in pppf(T_{\mathbf{G}_P})$ `ssi` $Q \in pppf(T_P)$ or pour tout atome $solve(Q') \in pppf(T_{\mathbf{G}_P})$, $solve(Q')$ est clos. Nous pouvons donc avoir $solve(Q') \in pppf(T_{\mathbf{G}_P})$ avec Q' clos et Q non clos.

Donc, en général, $pppf(T_{\mathbf{H}_P}) \neq SEM^C_{\mathbf{G}_P}$.

remarque 6.7 :

Comme pour la remarque 6.5, nous savons que $SEM^C_{\mathbf{V}_P}$ peut contenir des instances impropres.

D'une autre façon, une justification identique à celle de la remarque 6.6 est aussi valable.

Donc, en général, $SEM^C_{\mathbf{G}_P} \neq SEM^C_{\mathbf{V}_P}$.

remarque 6.8 :

La justification découle de ce qui a été dit dans la remarque 6.2. Si $solve(Q') \in pppf(T_{\mathbf{G}_P})$ alors $solve(Q')$ est clos. Nous constatons aussi rapidement que si $clause(C') \in pppf(T_{\mathbf{G}_P})$ alors $clause(C')$ est clos. Nous pouvons aussi remarquer que tous les faits de $\mathbf{G}_P$ sont clos.

Donc $SEM^C_{\mathbf{G}_P} = pppf(T_{\mathbf{G}_P})$.

6.4.3 Preuve du théorème 6.9

Il suffit de montrer que pour toute instance $A^1 \leftarrow B^1_1 \wedge \cdots \wedge B^1_n$ bien formée d'une règle de $\mathbf{V}_P$, $\psi(A^1) \leftarrow \psi(B^1_1) \wedge \cdots \wedge \psi(B^1_n)$ est une instance d'une règle de $\mathbf{G}_P$.

- Soit $clause^1(Q^1) \leftarrow$ l'instance bien formée d'une règle de $\mathbf{V}_P$. D'après le lemme 7 du chapitre 4 et de la définition de $clause^1$, Q est instance d'une règle de P. Par définition, $clause^2(Q^2) \in \mathbf{G}_P$. Par ailleur, $Q^2 = \psi(Q^1)$. Donc, $\psi(clause^1(Q^1))$ est instance d'une règle de $\mathbf{G}_P$.

- $\psi(solve^1(empty^1)) \leftarrow = solve^2(\psi(empty^1)) \leftarrow = solve^2(empty^2) \leftarrow$
  Donc $\psi(solve^1(empty^1)) \leftarrow \in \mathbf{G}_P$.

  $$\begin{aligned}&\psi(solve^1(x \,\&^1\, y)) \leftarrow \psi(solve^1(x)) \wedge \psi(solve^1(y)) = \\ &\qquad solve^2(\psi(x \,\&^1\, y)) \leftarrow solve^2(\psi(x)) \wedge solve^2(\psi(y)) = \\ &\qquad solve^2(\psi(x) \,\&^2\, \psi(y)) \leftarrow solve^2(\psi(x)) \wedge solve^2(\psi(y))\end{aligned}$$

  $\psi(x)$ et $\psi(y) \in VAR_{\mathcal{L}^2}$

  Donc $\psi(solve^1(x \,\&^1\, y)) \leftarrow \psi(solve^1(x)) \wedge \psi(solve^1(y))$ est instance d'une règle

de $\mathbf{G}_P$.

$$\psi(solve^1(x\, if^1\, y)) \leftarrow \psi(clause^1(x)) \wedge \psi(solve^1(y)) =$$
$$solve^2(\psi(x\, if^1\, y)) \leftarrow clause^2(\psi(x)) \wedge solve^2(\psi(y)) =$$
$$solve^2(\psi(x)\, if^2\, \psi(y)) \leftarrow clause^2(\psi(x)) \wedge solve^2(\psi(y))$$

$\psi(x)$ et $\psi(y) \in VAR_{\mathcal{L}^2}$
Donc $\psi(solve^1(x\, if^1\, y)) \leftarrow \psi(clause^1(x)) \wedge \psi(solve^1(y))$ est instance d'une règle de $\mathbf{G}_P$.

Conclusion :
Si $adp(solve^1(Q^1))$ est un arbre de preuve bien formé pour $\mathbf{V}_P$ alors $\psi(adp(solve^1(Q^1)))$ est un arbre de preuve pour $\mathbf{G}_P$.

6.4.4 Preuve du théorème 6.10

Raisonnement identique à celui utilisé pour la preuve du théorème 6.9.

6.4.5 Preuve des corollaires

Les corollaires 6.11 et 6.12 sont conséquences immédiates du théorème 6.9 et de la définition de ψ.

6.5 Conclusion

On remarque qu'il existe une bijection entre les arbres de preuve bien formés de $\mathbf{V}_P$ et ceux de $\mathbf{G}_P$. La sémantique de $\mathbf{V}_P$ restreinte aux atomes bien formés est identique à celle de $\mathbf{G}_P$. On aurait pu constater les mêmes similitudes entre les arbres de preuve clos bien formés de $\mathbf{V}_P$ et les arbres de preuve de $\mathbf{H}_P$.

L'unique différence entre ces trois programmes, mis à part la représentation du programme objet, réside dans la définition de *clause*.

Il se dégage de *vanilla* que le prédicat *solve* détermine les squelettes des arbres de preuve alors que le prédicat *clause* détermine les instances des atomes qui étiquettent les nœuds de ces arbres.

Dans $\mathbf{G}_P$, nous représentons l'ensemble des instances d'une clause $A \leftarrow E$ de P par tous les faits $clause(C') \leftarrow$ où C' représente une instance de $A \leftarrow E$. Pour $\mathbf{V}_P$, on décide simplement de les représenter par $clause(A'\, if\, E') \leftarrow$ dont toutes les instances incluent toutes les représentations d'instances de $A \leftarrow E$. Comme le prédicat *clause* détermine la nature des atomes étiquetant les nœuds des arbres de preuve, il en résulte la complétude de $\mathbf{V}_P$.

La structure de *vanilla* s'impose comme base de nombreux méta-programmes, non seulement parce qu'elle est simple, mais aussi, parce que les rôles de *solve* et *clause* permettent aisément d'y greffer les sémantiques voulues.

Le véritable méta-interprète de base est *vanilla* avec représentation close du programme objet ($\mathbf{G}_P$).

En modifiant la définition de *clause*, nous exprimerons les sémantiques voulues. Par exemple, si nous voulons n'avoir dans la sémantique du méta-programme que les racines d'arbre de preuve tel que tous les nœuds étiquetant l'arbre de preuve contiennent la représentation d'un atome non clos, il suffit de prendre *vanilla*, une représentation close du programme objet et la définition suivante pour *clause* :
$clause(A'\, if\, E') \leftarrow$ `ssi` $A \leftarrow E$ est instance d'une clause de P et A n'est pas clos.

Dans le chapitre suivant, nous montrerons que *vanilla* permet d'écrire un méta-interprète "universel". Dans [12, chapitre 19], nous trouvons une palette des méta-programmes issus de *vanilla*.

Chapitre 7

Représentation quelconque du programme objet

7.1 Introduction

Le chapitre 4 a fait apparaître deux difficultés relativement liées. La première est dûe à la distinction entre symboles de prédicat et symboles de fonction, la deuxième vient des instances "impropres" de représentation d'atomes. Le langage de la logique du premier ordre n'est pas adaptée à la méta-programmation. Nous ne souhaitons pourtant pas changer radicalement de langage, l'objectif opérationnel restant Prolog. Nous définissons un langage qui élimine les difficultés et qui se trouve plus proche de Prolog (déclaratif). Notre langage est un sur-ensemble de Prolog (variable autorisée en tête de clause) alors que les clauses de Horn n'en sont qu'une partie. Comme pour Prolog, l'aspect termal de la programmation est l'unique considération.

7.2 Définition du langage

Plaçons-nous dans un cadre plus général que pour les chapitres 4 et 5.

On suppose un ensemble infini dénombrable de variables, noté VAR, et un ensemble infini dénombrable de symboles, noté $FONC$. On définit l'ensemble $TERM$ par :

$$\forall x \in VAR, x \in TERM$$
$$\forall f \in FONC, f \in TERM$$
$$\forall f \in FONC, \forall t_1, \cdots, t_n \in TERM, f(t_1, \cdots, t_n) \in TERM$$

On définit de la façon habituelle les notions d'instance et de substitution. $TERM_c$ désigne l'ensemble des termes clos.

Soit $\leftarrow$ et $\wedge$ deux constructeurs, on définit une règle par :

$$\forall t \in TERM, t \leftarrow \text{ est une règle}$$
$$\forall t_1, \cdots, t_n \in TERM, t \leftarrow t_1 \wedge \cdots \wedge t_n \text{ est une règle.}$$

Un programme désigne un ensemble de règles. Si P est un programme, on définit $T_P : TERM \rightarrow TERM$ par :

$$T_P(I) = \{Q \in TERM / \exists (Q \leftarrow E) \text{ instance d'une règle de } P \text{ telle que } E \subseteq I\}$$

Si P est un programme, on définit $T_P^c : TERM_c \rightarrow TERM_c$ par :

$$T_P^c(I) = \{Q \in TERM_c / \exists (Q \leftarrow E) \text{ instance close d'une règle de } P \text{ telle que } E \subseteq I\}$$

$(T_P^c(I) = T_P(I) \cap TERM_c)$.

On note $REGLE$ l'ensemble des règles et $PROG$ l'ensemble des programmes.

La sémantique de P est définie par $pppf(T_P)$.

7.3 Le méta-interprète *Super_vanilla*

Nous étudions dans ce chapitre le méta-interprète donné figure 2.3 page 17.

Soit **R** le programme :

$$resoud(P, empty) \leftarrow$$
$$resoud(P, A \,\&\, B) \leftarrow resoud(P, A) \wedge resoud(P, B)$$
$$resoud(P, A) \leftarrow regle(P, A\, if\ B) \wedge resoud(P, B)$$

$$regle(P, C) \leftarrow instance(C, D) \wedge appartenance(D, P)$$

$instance(C, D)$ est supposé appartenir à la sémantique de R `ssi` C est la représentation d'une instance de la règle représentée par D. $appartenance(D, P)$ est supposé appartenir à la sémantique de R `ssi` D est la représentation d'une règle du programme représenté par P.

Nous sous-entendons, dans la suite, par représentation des programmes, la représentation choisie pour représenter les programmes en premier argument de *resoud* et *regle*, et par représentation des termes, la représentation choisie pour représenter les termes des règles en deuxième argument de *resoud* et *regle*. Il est évidement plus simple de décider d'une même représentation pour les programmes et les termes; c'est, bien entendu, notre choix dans la suite. Nous dirons donc représentation pour désigner la représentation des programmes et des termes. Nous noterons X' la représentation d'un objet X (terme, règle ou programme).

Nous donnons plus loin une définition de *instance* et *appartenance* pour une représentation des programmes et une représentation des termes fixées.

Il n'existe pas de représentations simples (implicite ou close) qui soient des représentations correctes pour les programmes et les termes.

Deux représentations implicites sont envisageables. Les programmes sont représentés par la liste des représentations de leurs règles, les termes sont représentés par eux-mêmes (ou une variante).

Considérons le programme P, contenant l'unique règle $p(x) \leftarrow$ avec $p \in FONC$ et $x \in VAR$. Il est représenté par P' : $[p(x)\, if\ empty]$.
Soit $a \in FONC$, $regle([p(x)\, if\ empty], p(a) ifempty)$ doit appartenir à $pppf(T_R)$ ainsi que toutes ses instances.
Donc, si $b \in FONC$, $regle([p(b)\, if\ empty], p(a)\, if\ empty) \in pppf(T_P)$.
Or, $p(a)\, if\ empty$ n'est instance d'aucune règle du programme qui est représenté par $[p(b)\, if\ empty]$.

La représentation implicite n'est donc pas satisfaisante. La représentation close ne l'est pas non plus, pour la simple raison qu'elle n'est pas une représentation. Toute représentation doit être injective. Soit $p \in FONC$, $x \in VAR$, $c \in FONC$, on choisit de représenter x par c. Donc $p(x) \leftarrow$ est représenté par $p(c)\, if\ empty$, et $p(c) \leftarrow$ est représenté par $p(c)\, if\ empty$. La généralité de notre langage ne permet pas d'utiliser la représentation close telle que nous l'avions définie. Il est néanmoins possible de donner une représentation assez proche :

Soit φ une bijection de VAR dans $FONC$.
Soit $t \in TERM$,
si $t \in FONC$ alors t est représenté par $fonc(t)$,
si $t \in VAR$ alors t est représenté par $var(\varphi(t))$,
si $t = f(t_1, \cdots, t_n)$ et $t_1, \cdots t_n$ sont représentés par $t'_1, \cdots, t'_n$ alors
t est représenté par $f(t'_1, \cdots, t'_n)$.

Nous étudions dans la section suivante une représentation qui nous permet d'écrire simplement et de manière finie les définitions de *instance* et *appartenance*.

7.4 Définition de *instance* et *appartenance*

7.4.1 Représentation

Symboles utilisés pour représenter les termes objets :
$zero$ avec une arité 0 (entier 0)
s avec une arité 1 (fonction successeur des entiers)
var avec une arité 1 (son argument est une variable)
app avec une arité 2 (application du premier argument aux termes de la liste en deuxième argument)
[] avec une arité 0 (liste vide)
| avec une arité 2 (constructeur de liste avec les notations habituelles en Prolog)

Soit φ une bijection de VAR dans $\mathbb{N}$.
Soit ψ une bijection de $FONC$ dans $\mathbb{N}$.
Si $s \in FONC$ et $t \in TERM$, on s'autorise à noter $s^0(t)$ le terme t et $s^{i+1}(t)$ le terme $s(s^i(t))$.

$x \in VAR$ est représenté par $var(s^{\varphi(x)}(zero))$.

Si $t_1, \cdots, t_n \in TERM$ sont représentés par $t'_1, \cdots, t'_n$ alors $f(t_1, \cdots, t_n) \in TERM$ est représenté par $app(s^{\psi(f)}(zero), [t'_1, \cdots, t'_n])$.

Si A est un terme représenté par A' et B une conjonction de termes représentée par B' (B est éventuellement un terme ou $true$) alors
$true$ est représenté par $empty$
$A \wedge B$ est représenté par $A' \,\&\, B'$
$A \leftarrow$ est représenté par $A' \, if \; emty$
$A \leftarrow B$ est représenté par $A' \, if \; B'$

Un programme est représenté par la liste de ses règles.

7.4.2 Les méta-interprètes R_v et R_c

Nous allons maintenant donner le méta-interprète R_v utilisant cette représentation.

Une substitution est représentée par une liste ordonnée de termes. Nous pouvons indicer cette liste par des entiers formels. Le premier élément de la liste a l'indice $zero$; si un élément a l'indice e, l'élément suivant dans la liste a l'indice $s(e)$. Le terme de la liste se trouvant à l'indice $s^n(zero)$ désigne la représentation du terme substitué à la variable représentée par $var(s^n(zero))$. Les substitutions ainsi décrites portent sur un nombre fini de variables, ce qui est suffisant puisque notre problème est de savoir qu'un terme est instance d'un autre. Toutes les variables absentes de la liste sont supposées être substituées par elles-mêmes.

Soit R_v le programme :

$resoud(P, empty) \leftarrow$
$resoud(P, A \,\&\, B) \leftarrow resoud(P, A) \wedge resoud(P, B)$
$resoud(P, A) \leftarrow regle(P, A \, if \; B) \wedge resoud(P, B)$

$regle(P, C) \leftarrow instance(C, D) \wedge appartenance(D, P)$

$instance(C, D) \leftarrow subst(S) \wedge instregle(C, D, S)$

$appartenance(C, [C|P]) \leftarrow estregle(C) \wedge estprog(P)$
$appartenance(C, [D|P]) \leftarrow estregle(D) \wedge appartenance(C, P)$

$instregle(A \, if \; E, B \, if \; F, S) \leftarrow inst(A, B, S) \wedge instconj(E, F, S)$
$instconj(empty, empty, S) \leftarrow$
$instconj(A, B, S) \leftarrow inst(A, B, S)$
$instconj(A \,\&\, E, B \,\&\, F, S) \leftarrow inst(A, B, S) \wedge instconj(E, F, S)$
$inst(app(E, X), app(E, Y), S) \leftarrow instliste(X, Y, S) \wedge entier(E)$
$inst(T, var(V), S) \leftarrow danssubst(T, V, S)$
$instliste([], [], S) \leftarrow$
$instliste([T|X], [V|Y], S) \leftarrow inst(T, V, S) \wedge instliste(X, Y, S)$

$subst([]) \leftarrow$
$subst([T|S]) \leftarrow terme(T) \wedge subst(S)$
$danssubst(T, zero, [T|S]) \leftarrow$
$danssubst(T, s(E), [X|S]) \leftarrow danssubst(T, E, S)$

$terme(var(E)) \leftarrow entier(E)$
$terme(app(E,X)) \leftarrow entier(E) \wedge listetermes(X)$
$listetermes([]) \leftarrow$
$listetermes([T|X]) \leftarrow terme(T) \wedge listetermes(X)$
$entier(zero) \leftarrow$
$entier(s(E)) \leftarrow entier(E)$

$estprog([]) \leftarrow$
$estprog([C|P]) \leftarrow estregle(C) \wedge estprog(P)$
$estregle(A\, if\, E) \leftarrow terme(A) \wedge estcorps(E)$
$estcorps(empty) \leftarrow$
$estcorps(A) \leftarrow terme(A)$
$estcorps(A\,\&\,E) \leftarrow terme(A) \wedge estcorps(E)$

Une preuve classique montre que $entier(E) \in pppf(T_{R_v})$ ssi E est un entier formel. Une spécification inductive montre immédiatement que $terme(T) \in pppf(T_{R_v})$ ssi T est la représentation d'un terme. $subst(S) \in pppf(T_{R_v})$ ssi S est une liste de représentation de termes, donc si S est une substitution telle que nous l'avons définie. $danssubst(T,V,S) \in pppf(T_{R_v})$ ssi à l'indice V de la liste S se trouve le terme T. $instance(C,D) \in pppf(T_{R_v})$ ssi C représente une règle instance de la règle représentée par D. $appartenance(C,P) \in pppf(T_{R_v})$ ssi C est la représentation d'une règle du programme représenté par P.

Nous ne prouvons pas la complétude et la correction de toutes ces règles. Les preuves sont classiques et simples, mais longues.

Nous en concluons que la définition de $regle$ a la propriété attendue :

$regle(P,C) \in pppf(T_{R_v})$ ssi C représente l'instance d'une règle du programme représenté par P.

R_c désigne le programme issu de R_v et qui permet d'obtenir la sémantique close des programmes objets. R_c est obtenu en ajoutant à R_v les règles :

$substclose([]) \leftarrow$
$substclose([T|S]) \leftarrow termeclos(T) \wedge substclose(S)$
$termeclos(app(E,X)) \leftarrow entier(E) \wedge listeclos(X)$
$listeclos([]) \leftarrow$
$listeclos([T|X]) \leftarrow termeclos(T) \wedge listeclos(X)$

et en modifiant la définition de $instance$ par :

$instance(C,D) \leftarrow substclose(S) \wedge instregle(C,D,S)$

$substclose(S)$ permet de décrire des substitutions closes, donc $instance(C,D) \in pppf(T_{R_c})$ ssi C représente une instance close de la règle représentée par D.

7.5 Théorèmes

Théorèmes

7.1 Soit deux représentations fixées pour le programme objet et les termes,
$\forall Q \in TERM, \forall P \in PROG,$
$resoud(P', Q') \in pppf(T_R)$ `ssi` $Q \in pppf(T_P)$.

7.2 Il existe une relation unaire exprimable par programme telle que sa relation complémentaire n'est pas exprimable par programme.

7.6 Preuves

7.6.1 Preuve du théorème 7.1

Rappelons la propriété de $regle$:
$regle(P', C') \in pppf(T_R) \Longleftrightarrow C$ est instance d'une règle de P.

$\Longleftarrow$
Montrons que $Q \in pppf(T_P) \Longrightarrow resoud(P', Q') \in pppf(T_R)$.

- $T_P \uparrow 0 = \emptyset$ donc $Q \in T_P \uparrow 0 \Rightarrow resoud(P', Q') \in pppf(T_R)$.
- Supposons $\forall i < n$, $Q \in T_P \uparrow i \Rightarrow resoud(P', Q') \in pppf(T_R)$.
  Si $Q \in T_P \uparrow n$ alors $\exists (Q \leftarrow E)$ instance d'une règle de P telle que $E \subseteq T_P \uparrow n-1$ ou E est vide.
  Par définition, $regle(P', Q'\, if\, E') \in pppf(T_R)$.
  $resoud(P', Q') \leftarrow regle(P', Q'\, if\, E') \wedge resoud(P', E')$ est instance d'une règle de R.
  - Si E est vide
    $resoud(P', empty) \in pppf(T_R)$ donc $resoud(P', Q') \in pppf(T_R)$.
  - Si $E = t_1 \wedge \cdots \wedge t_m$
    $resoud(P', t'_i) \in pppf(T_R)$,
    $\forall t'_1, t'_2\, resoud(P', t'_1 \,\&\, t'_2) \leftarrow resoud(P', t'_1) \wedge resoud(P', t'_2)$ est instance d'une règle de R.
    Donc, $resoud(P', E') \in pppf(T_R)$ et $resoud(P', Q') \in pppf(T_R)$.

$\Longrightarrow$
Montrons que $resoud(P', Q') \in pppf(T_R) \Longrightarrow Q \in pppf(T_P)$.

- $T_R \uparrow 0 = \emptyset$ donc $resoud(P', Q') \in T_R \uparrow 0 \Rightarrow Q \in pppf(T_P)$.
- Supposons $\forall i < n$, $resoud(P', Q') \in T_R \uparrow i \Rightarrow Q \in pppf(T_P)$.
  Si $resoud(P', Q') \in T_R \uparrow n$ alors
  $\exists E'$, tel que $resoud(P', Q') \leftarrow regle(P', Q'\, if\, E') \wedge resoud(P', E')$ est instance de la troisième règle de R.
  D'après la définition de $regle$, $regle(P', C') \in pppf(T_R)$ `ssi` C est instance d'une règle de P.
  Donc, $Q \leftarrow E$ est instance d'une règle de P. Par hypothèse, $E \subseteq pppf(T_P)$ donc $Q \in pppf(T_P)$.

7.6.2 Preuve du théorème 7.2

Soit Q le programme constitué de R_c et de la clause :

$$q(x) \leftarrow resoud(x,y) \wedge transforme(x,y)$$

avec $transforme(x,y) \in pppf(T_Q^c)$ `ssi` $y = (\overline{q}(x))'$.

x est la représentation d'un programme, donc x est clos et x utilise des symboles issus d'un ensemble fini de symboles (app, var, s, $zero$ ...). La représentation étant fixée, il est aisé d'exprimer par programme la relation $transforme$.

Soit $A = \{t/q(t) \in pppf(T_Q^c)\}$, i-e $A = \{t/resoud(t,(\overline{q}(t))') \in pppf(T_{R_c}^c)\}$ ou encore $A = \{P'/\overline{q}(P') \in pppf(T_P^c)\}$.

Soit $\overline{A}$ le complémentaire de A par rapport à $TERM_c$.

$$\overline{A} = \{t/resoud(t,(\overline{q}(t))') \notin pppf(T_{R_c}^c)\}$$

Supposons que la relation $\overline{q}$, relation complémentaire de q, soit exprimable par le programme P.

Donc $\overline{A} = \{t/\overline{q}(t) \in pppf(T_P^c)\}$, i-e $\overline{A} = \{t/resoud(P',(\overline{q}(t))') \in pppf(T_{R_c}^c)\}$

Montrons par contradiction que P n'existe pas.

$$P' \in \overline{A} \Longleftrightarrow resoud(P',(\overline{q}(P'))') \in pppf(T_{R_c}^c)$$
$$P' \in \overline{A} \Longleftrightarrow q(P') \in pppf(T_Q^c)$$
$$P' \in \overline{A} \Longleftrightarrow P' \in A$$

Donc la relation $\overline{q}$ n'est pas exprimable par programme.

7.7 Le problème de la négation

7.7.1 Sémantique négative d'un programme

Nous ne parlons pas ici de programme avec "négation", mais nous cherchons à donner une sémantique négative (en plus de la sémantique déjà définie) à nos programmes. Pour simplifier, nous nous limitons à la sémantique close. Le théorème 7.2 nous montre qu'il est illusoire de penser que la sémantique négative d'un programme peut être définie comme le complémentaire de sa sémantique positive. Nous reprenons la sémantique définie dans [2].

Si P est un programme, $SEM^+(P)$ désigne sa sémantique positive, $SEM^-(P)$ sa sémantique négative. Elles sont définies par :

$$SEM^+(P) = pppf(T_P^c)$$

$$SEM^-(P) = \overline{pgpf(T_P^c)}$$

Nous définissons $ECHEC(P)$ par :

$$ECHEC(P) = \overline{T_P^c \downarrow \omega}$$

Intuitivement, $ECHEC(P)$ correspond aux échecs finis de P (même si cette notion n'est pas définie ici).

7.7.2 Théorèmes

Théorèmes :

7.3 $resoud(P', A') \in SEM^+(R)$ `ssi` $A' \in SEM^+(P)$

7.4 $resoud(P', A') \in SEM^-(R)$ `ssi` $A' \in SEM^-(P)$

7.5 $resoud(P', A') \in ECHEC(R)$ `ssi` $A' \in ECHEC(P)$

Nous ajoutons le nouveau connecteur $\neg$ au langage (sans changer la syntaxe des programmes).

Nous définissons le négatif d'un terme ($\neg t$) et d'un ensemble de termes.

Si $E \subseteq TERM$ alors $\neg E = \{\neg t / t \in E\}$.

Nous donnons une nouvelle formulation de la sémantique des programmes. Nous définissons $SEM(P)$, sémantique du programme P par :

$SEM(P) = SEM^+(P) \cup \neg SEM^-(P)$.

C'est-à-dire :

$$SEM(P) = pppf(T_P^c) \cup \neg \overline{pgpf(T_P^c)}$$

Corollaires :

7.6 $resoud(P', A') \in SEM(R)$ `ssi` $A \in SEM(P)$

7.7 $\neg resoud(P', A') \in SEM(R)$ `ssi` $\neg A \in SEM(P)$

7.8 Le méta-interprète R est correct et complet par rapport à la sémantique SEM.

7.7.3 Preuves

Preuve du théorème 7.3

Le théorème 7.3 est équivalent au théorème 7.1.

Preuve du théorème 7.4

$\Longrightarrow$

Il suffit de montrer que pour tout ordinal n :

$resoud(P', A') \notin T_R^c \downarrow n \Longrightarrow A \notin pgpf(T_P^c)$

- $T_R^c \downarrow 0 = TERM_c$ donc $resoud(P', A') \notin T_R^c \downarrow 0 \Rightarrow A \notin pgpf(T_P^c)$.
- Supposons $\forall i < n+1$, $resoud(P', A') \notin T_R^c \downarrow i \Rightarrow A \notin pgpf(T_P^c)$
  Si $resoud(P', A') \notin T_R^c \downarrow n+1$ alors
  $\forall resoud(P', A') \leftarrow C$ instance d'une règle de R, C est forcément de la forme $regle(A' \, if \, E') \wedge resoud(E')$ et
  soit $regle(A' \, if \, E') \notin T_R^c \downarrow n$
  soit $resoud(E') \notin T_R^c \downarrow n$
  Par conséquent, soit $A \leftarrow E$ n'est pas instance d'une règle de P (d'après la définition de $regle$), soit $E \not\subseteq pgpf(T_P^c)$
  donc $A \notin pgpf(T_P^c)$.

- α ordinal limite,
  Si $resoud(P', A') \notin T_R^c \downarrow \alpha$ alors $\exists\, i < \alpha, resoud(P', A') \notin T_R^c \downarrow i$
  donc $A \notin pgpf(T_P^c)$.

$\Longleftarrow$

Il suffit de montrer que pour tout ordinal n :
$$A \notin T_P^c \downarrow n \Longrightarrow resoud(P', A') \notin pgpf(T_R^c)$$

- $T_P^c \downarrow 0 = TERM_c$ donc $A \notin T_P^c \downarrow 0 \Rightarrow resoud(P', A') \notin pgpf(T_R^c)$.
- Supposons $\forall\, i < n+1,\ A \notin T_P^c \downarrow i \Rightarrow resoud(P', A') \notin pgpf(T_R^c)$
  Si $A \notin T_P^c \downarrow n+1$ alors
  $\forall\, A \leftarrow B_1 \wedge \cdots \wedge B_m$ instance d'une règle de P, $\exists\, i$, $B_i \notin T_P^c \downarrow n$
  Par conséquent, $\forall\, A'\, if\, E'$ représentation de l'instance d'une règle de P, $resoud(P', E') \notin pgpf(T_R^c)$
  donc $resoud(P', A') \notin pgpf(T_R^c)$.
- α ordinal limite,
  Si $A \notin T_P^c \downarrow \alpha$ alors $\exists\, i < \alpha, A \notin T_P^c \downarrow i$
  donc $resoud(P', A') \notin pgpf(T_R^c)$.

Preuve du théorème 7.5
Même raisonnement que pour la preuve du théorème 7.4, sans considérer le cas ordinal limite.

Preuve du corollaire 7.6
Conséquence logique du théorème 7.3.

Preuve du corollaire 7.7
Conséquence logique du théorème 7.4.

Preuve du corollaire 7.8
Conséquence logique des corollaires 7.6 et 7.7.

7.8 Conclusion

Le langage de la logique du premier ordre et le langage Prolog présentent principalement deux différences. La première est la distinction entre symboles de prédicats et symboles de fonctions qui n'existe pas en Prolog. La seconde est la restriction d'un langage de la logique du premier ordre à un ensemble donné de symboles alors que Prolog considère tous les symboles constructibles et que deux programmes Prolog seront toujours basés sur le même langage.

Nous avons résorbé ces deux différences en utilisant un langage unique dans lequel ne sont pas distingués les symboles de prédicats et les symboles de fonctions. Ce cadre nous permet d'avoir un langage identique à Prolog (déclaratif). Beaucoup de programmes Prolog ne peuvent avoir de sémantique en logique du premier ordre puisqu'ils ne sont pas expressions de clauses, alors que tous sont expressions de règles et ont un sens pour nous.

Ce nouveau langage nous permet de nous placer dans un cadre idéal pour l'étude de la méta-programmation. C'est dans ce langage que nous écrivons le méta-interprète "universel" *super_vanilla*.

Il est très intéressant de constater que R_v est un méta-interprète universel fini, dont nous avons prouvé la complétude et la correction. Rappelons que le méta-programme $\mathbf{V}_P$ (chapitre 4) n'est généralement pas correct, que le méta-programme $\mathbf{G}_P$ (chapitre 5) est généralement infini, et que tous deux sont seulement des interprètes du programme P.

Nous ne nous sommes pas intéressés dans les chapitres précédents à la négation. Ce problème est complexe, et nous avons préféré nous consacrer à la sémantique négative des programmes sans négation pour, éventuellement plus tard, nous pencher sur les programmes et méta-programmes avec négation.

Le théorème 7.2, qui, dans un autre formalisme, est équivalent au théorème 6.8.1 p. 176 de [7], nous permet d'écarter une définition de la sémantique de la négation qui ne serait pas programmable (ce qui est bien connu[1]).

Les corollaires 7.6 et 7.7 justifient l'ajout de la règle :

$$resoud(P, not\, A) \leftarrow \neg resoud(P, A)$$

au méta-interprète *super_vanilla*, mais ne prouvent pas que cela satisfasse notre attente.

[1]Problème de décision de la terminaison d'un programme.

Chapitre 8

Conclusion

Nous avons réalisé une étude approfondie du méta-interprète *vanilla* à travers la sémantique exprimée par le plus petit point fixe de l'application conséquence immédiate associée au programme.

Ce formalisme a l'avantage d'être inductif, donc de faciliter les démonstrations. De plus, comme le montre le chapitre 7, il est plus adapté à la méta-programmation que la logique.

Lloyd, dans [8], propose de travailler en logique du premier ordre typée, mais son langage est trop éloigné de Prolog pour y voir des applications directes. Il semble que le langage[1] présenté dans le chapitre 7 est idéal pour l'étude de la méta-programmation en Prolog.

Nous avons souvent précisé que le langage défini est identique à Prolog "déclaratif". Nous entendons, ici, Prolog sans coupure, ni prédicats prédéfinis. La coupure est une injure à notre formalisme, dans la mesure où elle impose à travers une vision SLD de préciser une règle de choix et d'ordonner les clauses du programme. C'est difficilement exprimable par l'application conséquence immédiate qui manipule des ensembles non ordonnés. En revanche, il est possible de l'adapter aux prédicats prédéfinis, et nous pouvons ajouter à *vanilla* les règles :

$$solve(A) \leftarrow predefini(A) \wedge A$$

et

$$predefini(X) \leftarrow$$

pour tout X de la forme $f(X_1, \cdots, X_n)$ où f est prédéfini d'arité n et $X_1, \cdots, X_n$ sont n variables.

L'utilisation de la méta-variable dans la première règle se justifie par le fait que le succès ou l'échec d'un prédicat prédéfini ne dépend pas de l'aspect déclaratif du programme objet, mais de l'interprète (celui sur la machine), ou de la définition de l'application dérivée de conséquence immédiate.

Nous n'avons étudié que les programmes définis mais le dernier chapitre est un point de départ à l'introduction de la négation dans les programmes. La règle pro-

[1] Langage unique pour tous les programmes, pas de distinction entre symboles de prédicats et symboles de fonctions.

posée pour *vanilla* étant bien entendu :

$$solve(not\, A) \leftarrow \neg solve(A)$$

Nous avons déjà défini une sémantique pour la négation, qui a l'avantage de ne pas s'écarter de notre formalisme. Nous pouvons introduire la notion d'∞-arbre de preuve comme dans [2].

Notre langage est bien plus proche de Prolog que les clauses de la logique qui n'en sont qu'un sous-ensemble. Nous pouvons constater qu'il donne une sémantique aux figures 1.1 et 1.2 page 6.

Il serait intéressant de reprendre les chapitres 4 et 5 dans ce formalisme. Nous constaterions alors que $\mathbf{V}_P$ est correct, et que la définition de *clause*, pour ce méta-interprète, est très proche de la définition de *clause* en Prolog.

Les chapitres 4 et 5 ont montré que la représentation close était mieux adaptée au débogage que la représentation implicite. Pourtant, les débogueurs programmés utilisent, pour des raisons d'efficacité, la représentation implicite. Mais ces débogueurs font usage du méta-prédicat *var* (ou du méta-prédicat *write* qui a un effet comparable). Cependant, l'utilisation de *var* ne nous permet plus de parler d'arbre de preuve. Il sous-entend la notion de dérivation. Encore une fois, il est possible d'adapter l'application conséquence immédiate à l'usage de *var*. Ce n'est pas un bon choix car la définition de l'application devient moins agréable. L'idéal serait d'opter pour la représentation close dans les preuves des méta-programmes, puis d'avoir un mécanisme de traduction de ces méta-programmes en des versions Prolog qui utilisent la représentation implicite et le méta-prédicat *var*.

Dans le même ordre d'idées, nous pouvons définir un nouveau langage de programmation proche de Prolog facilitant l'écriture de méta-programmes. Le langage Gödel de Lloyd est peut-être ce langage, mais nous n'avons pas eu assez de temps pour l'étudier.

Bibliographie

[1] P. Aczel, *An Introduction to Inductive Definitions.*
In : *Handbook of Mathematical Logic.*
Edited by J. Barwise, 1977 (pp 739–782).

[2] M. Bergère, *Approche déclarative du diagnostic d'erreur pour la programmation en logique avec négation.*
Thèse Université d'Orléans, 1991.

[3] H. Christiansen, *Models and resolution principles for logical meta-programming languages.*
Rapport de recherche N° 1594.
INRIA, 1992.

[4] P. Deransart et G. Ferrand, *Introduction à la méthodologie de programmation en logique.*
Novia, janvier 1988.

[5] P. Deransart and J. Maluszyński, *Logic Programming and Attribute Grammars.*
The Journal of Logic Programming, 1985.

[6] G. Ferrand, *Error diagnosis in Logic Programming, an adaptation of E.Y. Shapiro's method.*
Rapport de recherche INRIA RR 375.
The Journal of Logic Programming, 1987.

[7] M. Fitting, *Computability Theory, Semantics, and Logic Programming.*
Oxford Logic Guides : 13.
Oxford University Press, 1985.

[8] P.M. Hill and J.W. Lloyd, *Analysis of meta-programs.*
In : H. Abramson and M.H. Rogers, *Meta-Programming in Logic Programming.*
M I T Press, 1989 (pp 23–51).

[9] J.W. Lloyd, *Foundations of Logic Programming.*
Springer-Verlag, second edition, 1987.

[10] U. Nilsson and J. Maluszyński, *Logic, Programming and Prolog.*
Chapter 8 (pp 151–166), *Amalgamating Object and Meta-language.*
John Wiley & Sons, 1989.

[11] E. Shapiro, *Algorithmic Program Debugging.*
Thesis, Yale University.
M I T Press, 1982.

[12] L. Sterling and E. Shapiro, *The Art of Prolog.*
M I T Press, 1986.