THÈSE de DOCTORAT

présentée à

l'École Nationale des Ponts et Chaussées

Spécialité

Informatique

par

Mathieu JAUME

Sujet de la thèse

Contributions à la sémantique de la programmation logique

Soutenue le 11 Janvier 1999 devant le jury composé de

Irène	GUESSARIAN	Présidente
Véronique	VIGUIÉ DONZEAU-GOUGE	Rapporteurs
Olivier	RIDOUX	
Gilles	BERNOT	Examinateurs
Gilles	DOWEK	
René	LALEMENT	Directeur

Remerciements

Mon goût pour les études est assez récent et a pour origine la grande qualité des enseignants que j'ai pu rencontrés durant ma scolarité. Irène GUESSARIAN figure parmi eux et je suis très heureux qu'elle ait accepté de présider le jury.

Véronique VIGUIÉ DONZEAU-GOUGE et Olivier RIDOUX ont accepté de rapporter sur ce travail. Je leur adresse tous mes remerciements pour la lecture attentive de la version préliminaire de ce manuscrit ainsi que pour les remarques qu'ils m'ont faites.

La fréquentation de René LALEMENT est enrichissante à plus d'un titre et cette thèse doit beaucoup à l'originalité de ses points de vue et à son optimisme. Il m'est impossible de «résumer» ici ses qualités en quelques mots mais je tiens à le remercier pour le plaisir que j'ai eu à travailler sous sa direction.

Je tiens à remercier tout particulièrement Gilles BERNOT pour avoir accepté de participer au jury mais aussi pour m'avoir accueilli au sein du LAMI de l'Université d'Évry. J'en profite aussi pour remercier Catherine DUBOIS pour les discussions que nous avons eues et l'ensemble du LAMI pour leur accueil chaleureux.

Je remercie Gilles DOWEK pour avoir bien voulu consacrer un peu de son temps à mon travail en acceptant de participer au jury.

Bien qu'un peu éloigné de ses préoccupations actuelles, ce travail n'aurait pu exister sans l'amitié et la confiance de Alain DAVID. Il fut mon premier contact avec le monde de la recherche et mon travail doit beaucoup à ses encouragements constants et ses conseils judicieux tout au long de ma scolarité.

J'ai eu la chance de poursuivre mes études aux côtés de Jannick DELECHAMP. La rigueur de sa démarche et l'originalité de ses reflexions m'ont accompagné durant ces dernières années. Si nos centres d'intérêt se sont depuis un peu éloignés, son amitié est restée constante et ses encouragements précieux. Pour tous les moments que nous avons partagés, je lui adresse de chaleureux remerciements.

Daniel HIRSCHKOFF et, plus récemment, Hervé GRALL ont partagé mon quotidien au CERMICS et je les remercie pour la «bonne humeur» qu'ils ont apportée dans le bureau.

Enfin, je tiens à remercier Jean-Michel DOUIN et Jean-François DAZY avec qui j'ai eu la chance d'enseigner sur des thèmes proches de la programmation logique et qui m'ont toujours apporté leur soutien.

Bien sûr, j'oublie ici de nombreuses personnes qui ont fait preuve de beaucoup de patience et de délicatesse. Qu'ils en soient remerciés.

Résumé La notion de preuves en programmation logique est examinée à deux niveaux différents. D'un point de vue externe, la «théorie classique» de la programmation logique est complètement formalisée dans le calcul des constructions inductives. Après avoir envisagé le problème de la définition de fonctions partielles dans un système dans lequel seules les fonctions totales sont représentables, l'unification est obtenue en réutilisant une preuve formelle existante portant sur un sur-ensemble des termes. Les propriétés fondamentales de la SLD-résolution sont alors formalisées. Le niveau de détail imposé par la mécanisation des preuves considérées a mis en relief la complexité cachée de certaines preuves : le mécanisme de renommage est traité de manière explicite, transformant ainsi certaines certitudes théoriques en réalités. D'un point de vue interne, les preuves SLD, finies ou infinies, sont comparées à celles que l'on peut obtenir, par induction ou par co-induction, à partir des clauses d'un programme logique vues comme des règles d'inférence. Dans le cas fini la correspondance est complète («ce que calcule un programme est prouvable») tandis que dans le cas infini, certains objets non calculables sont toutefois prouvables. Les propriétés classiques des définitions co-inductives et la comparaison de certaines dérivations infinies à des termes de preuve d'un type co-inductif, se révèlent utiles tant pour expliquer les résultats d'incomplétude d'approches existantes que pour définir une sémantique valide et complète pour une classe de dérivations infinies (précisément celles qui ne construisent pas de termes infinis).

Abstract This work can be split in two parts. First, we present a full formalisation of the semantics of definite programs, in the calculus of inductive constructions. For this, we describe a formalisation of the proof of first order terms unification obtained from a similar proof dealing with quasi-terms, thus showing in a general setting how partial functions can be considered in a system with total functions. Then, SLD-resolution is explicitely defined: the renaming process required in SLD-derivations is made explicit, thus introducing complications, usually overlooked, during the proofs of classical results. Last, switching and lifting lemmas and soundness and completeness theorems are formalised. For this, we present two lemmas, usually omitted, which are needed. The second part focuses on the assignment of meaning to infinite derivations in logic programming. By considering proofs as objects in a co-inductive set, standard properties of co-inductive definitions are used both to explain why approaches developped by considering infinite elements in the universe of the discourse are not complete and to define a sound and complete semantics, based on the "logic program as co-inductive definition" paradigm, for a subclass of infinite derivations, called infinite derivations over a finite domain (i.e. derivations which do not compute infinite terms).

Table des matières

Introduction

Spécifier les propriétés sémantiques d'un langage permet, d'une part à celui qui programme dans ce langage de disposer d'une sémantique claire, précise et non ambiguë, et, d'autre part, à celui qui conçoit un interpréteur (ou un compilateur) pour ce langage d'en vérifier la correction. Les chapitres qui suivent portent sur deux questions de sémantique de la de programmation logique: la première est celle de la formalisation et du développement de la «théorie classique» de la programmation logique dans un méta-système logique (le calcul des constructions inductives), la seconde est celle de la sémantique des dérivations infinies.

La programmation logique

Issue de la recherche en démonstration automatique en logique du premier ordre, initiée par J. Herbrand [46], et basée sur le principe de résolution de A.J. Robinson [86, 87], la programmation logique a été introduite dans la pratique par A. Colmerauer (langage Prolog, 1972) [18] tandis que ses fondements théoriques ont été établis par R.A. Kowalski et M.H. van Emden [96]. Ce modèle de programmation abandonne le «principe» de programmation impérative, qui oblige l'informaticien à indiquer pas à pas à l'ordinateur ce qu'il doit faire, au profit d'une programmation plus déclarative qui consiste à représenter dans un formalisme adéquat les données du problème à partir desquelles un résultat pourra être déduit. Ce paradigme de programmation est né de la découverte d'un sous-ensemble de la logique du premier ordre (aussi appelé le fragment Hornien de la logique des prédicats) associé à une interprétation procédurale correcte et complète, basée sur la SLD-résolution (*Selection Linear Definite*). La naissance du langage Prolog a été motivée par l'étude de problèmes d'analyse et de compréhension de la «langue naturelle» nécessitant l'utilisation de résultats en logique du premier ordre et en démonstration automatique. L'expression des connaissances sous forme de clauses et l'emploi d'une règle d'inférence adéquate a conduit à un véritable langage de programmation. Une des idées sous-jacentes de la programmation logique est de considérer un programme comme un ensemble de relations dont l'exécution consiste à «démontrer» une nouvelle relation à partir de celles qui constituent le programme. On peut donc voir le calcul effectué par un «programme logique», à partir d'une requête, comme l'extraction d'un résultat à partir d'une preuve. Le programme utilisé exprime les propriétés caractérisant la recherche d'une preuve, et la requête soumise spécifie quelle preuve doit être construite. On peut ainsi schématiser le paradigme de la programmation logique par les deux célèbres identifications :

$$\begin{array}{rcl} \text{programme} & \equiv & \text{théorie du premier ordre} \\ \text{exécution} & \equiv & \text{recherche de preuves} \end{array}$$

Les fondements théoriques de la programmation logique sont synthétisés dans [5, 7, 63, 67, 96] et l'utilisation du langage de programmation PROLOG est illustrée dans [16, 94]. Par la suite, les bases théoriques de la programmation logique ont été consolidées et de nombreuses extensions ont été proposées afin d'augmenter la puissance d'expression de ce modèle de programmation : négation [6, 84], contraintes [47], concurrence [90], ordre supérieur [74] ...

Un «programme logique» est un ensemble fini de clauses de Horn. Ces formules sont notées $A \leftarrow B_1, \cdots, B_n$ où A et B_i $(1 \leqslant i \leqslant n)$ désignent des atomes. Plusieurs lectures d'un programme logique sont possibles, donnant lieu à différentes sémantiques. Traditionnellement, on distingue la sémantique déclarative d'un programme de sa sémantique opérationnelle (aussi appelée sémantique procédurale). D'un point de vue déclaratif, la clause $A \leftarrow B_1, \cdots, B_n$ peut s'interpréter par «*si B_1 et $\cdots$ et B_n sont vrais, alors A est vrai*». Dans ce cas, une clause spécifie une relation «logique» entre formules atomiques exprimée par la formule quantifiée universellement $\forall \vec{x}(A \vee \neg B_1 \vee \cdots \vee \neg B_n)$ (ou de manière logiquement équivalente par $\forall \vec{x}(B_1 \wedge \cdots \wedge B_n \Rightarrow A)$). Cette formule indique comment interpréter logiquement une clause en utilisant des connecteurs logiques ($\neg$, $\vee$, $\wedge$, $\Rightarrow$) et des quantificateurs ($\forall$). Une clause négative (aussi appelée but) est une clause de la forme $\leftarrow B_1, \cdots, B_q$ dont la signification logique est donnée par la formule $\forall \vec{x}(\neg B_1 \vee \cdots \vee \neg B_q)$ (ou de manière logiquement équivalente par $\neg \exists \vec{x}(B_1 \wedge \cdots \wedge B_q)$). L'exécution d'un programme P à partir d'un but $\leftarrow B_1, \cdots, B_q$ consiste à réfuter l'ensemble $P \cup \{\forall \vec{x}(\neg B_1 \vee \cdots \vee \neg B_q)\}$, ce qui revient, d'un point de vue déclaratif, à prouver à partir de P, la formule existentielle $\exists \vec{x}(B_1 \wedge \cdots \wedge B_q)$. Cette preuve est obtenue de manière constructive : un «résultat» est extrait de la preuve de $\exists \vec{x}(B_1 \wedge \cdots \wedge B_q)$ et correspond à une affectation des variables de $\vec{x}$. Il s'agit d'une substitution solution θ telle que $P \models \forall \vec{z} \theta(B_1 \wedge \cdots \wedge B_q)$ où $\models$ dénote la relation de conséquence sémantique. Cette lecture logique des programmes (qui utilise la notion de «vérité logique» *à la* A. Tarski) ne fournit aucune information quant à la manière dont le programme, en tant que processus de preuve, s'exécute. D'un point de vue opérationnel, la clause $A \leftarrow B_1, \cdots, B_n$ peut se comprendre comme une procédure A dont l'appel engendrerait l'exécution des B_i. Si l'on adopte ce point de vue, le passage des paramètres entre procédure appelante et procédure appelée se fait par le mécanisme d'unification [1]. Cette interprétation opérationnelle des programmes est définie par la SLD-résolution, règle d'inférence présentée et formalisée *en détail* dans le chapitre 3. Cette lecture repose sur la notion de dérivabilité syntaxique (*à la* G. Frege) ; les substitutions construites lors d'une dérivation sont appelées des réponses. Un aspect fondamental de la programmation logique est la correspondance complète entre ces deux lectures (sémantiques). Mais cette correspondance se révèle aussi utile d'un point de vue plus pratique : elle permet de comprendre le résultat de l'exécution d'un programme logique à partir d'une requête sans en connaître nécessairement le mécanisme d'exécution. Enfin, signalons qu'il existe bien d'autres lectures de la programmation logique : réécriture, résolution d'équations, preuve de théorème, réseau de processus communicants ...

Formalisation de la SLD-résolution dans un méta-système logique : pourquoi formaliser ce qui semble déjà formel ?

Une preuve formelle est une preuve dont on peut établir la correction sans interpréter les symboles qu'elle contient : il suffit de s'assurer que chaque étape de la preuve obéit à certaines règles de manipulation des symboles. Les ordinateurs permettent une manipulation d'objets symboliques, et il est donc possible de représenter des preuves (formelles) et d'en vérifier la validité (en quoi cet objet est-il bien une preuve ?) à l'aide d'un formalisme adéquat. Le logiciel utilisé pour le codage et la vérification des preuves est le «Coq *Proof Assistant*» (version 6.2.2) [19], développé à l'Inria. Coq est un assistant à la preuve basé sur un méta-système logique (*logical framework*) : le calcul des constructions inductives [21, 97]. Il s'agit d'une extension d'un λ-calcul typé d'ordre supérieur [8] permettant de représenter, via l'isomorphisme de Curry-Howard, un «raisonnement mathématique» (preuve en logique intuitionniste d'ordre supérieur) pouvant être mis en œuvre sur ordinateur. On peut ainsi vérifier et garantir «mécaniquement» à l'aide de ce formalisme la validité d'une preuve. Le chapitre 1 est essentiellement consacré à la présentation de cet assistant à la preuve. De nombreux développements ont déjà été réalisés à l'aide du système Coq : axiomatisations de théories mathématiques (groupes, domaines, catégories, théorie des ensembles de Zermelo-Fraenkel ...), propriétes formelles de langages fonctionnels, λ-calcul avec substitutions explicites, théorie des automates, unification de (quasi-)termes du premier ordre ...

Toutes ces «contributions» se révèlent d'une grande utilité : tout comme un «composant logiciel», une preuve formelle est réutilisable. Le développe-

1. l'unification fournissant une affectation à certaines variables des procédures appelantes et appelées, ce mécanisme de passage de paramètres est multi-directionnel, abolissant ainsi la distinction entre paramètres d'entrée et paramètres de sortie

ment présenté dans le chapitre 3 correspond à une formalisation des résultats classiques de la programmation logique : lemmes de commutation et de généralisation, théorèmes de validité et de complétude de la SLD-résolution. La première étape de cette formalisation consiste naturellement à spécifier un ensemble de termes et à prouver la décidabilité de la propriété d'unification sur cet ensemble, propriété constamment utilisée en programmation logique. Cette propriété a été formalisée par J. Rouyer [89] et fait partie des «contributions» du logiciel COQ ; la preuve déjà établie sera donc réutilisée dans le chapitre 2. Toutefois, l'ensemble sur lequel elle est définie ne correspond pas exactement à l'ensemble des termes que nous utilisons et la propriété d'unification sera obtenue en «transposant» aux termes le théorème établi pour les quasi-termes. Cette «technique» de preuve est présentée dans le chapitre 2 et dans [52, 55].

Les résultats fondamentaux de la SLD-résolution sont ensuite formalisés dans le chapitre 3. Puisque ces résultats ne sont pas «nouveaux» et sont maintenant démontrés dans de nombreux ouvrages consacrés à la programmation logique, on peut s'interroger sur l'utilité de leur formalisation. En effet, la règle de coupure du calcul des séquents et le fragment Hornien de la logique du premier ordre sont à la base du paradigme de la programmation logique et les propriétés classiques de ce modèle de programmation s'expriment et s'établissent dans un cadre assez formel. Aussi, il peut paraître surprenant, voire inutile, de vouloir formaliser ces résultats dans un méta-système logique. Pour cela, rappelons que la formalisation d'une preuve consiste, d'une part, à définir à l'aide d'un langage de spécification les objets manipulés et à expliciter complètement, et *a priori*, les hypothèses de l'énoncé à démontrer et, d'autre part, à décomposer les étapes de la preuve en «exhibant» les mécanismes de construction des conclusions à partir des hypothèses. C'est en se livrant à cet exercice que l'on constate que certaines preuves classiques, dans leur présentation courante, omettent des *détails* qui peuvent s'avérer cruciaux. Ces «*détails*», volontairement ignorés dans la littérature, jouent pourtant un rôle important et leur omission est à l'origine de «petites erreurs» courantes : certains résultats corrects sont établis incorrectement et certains résultats ne sont corrects que sous certaines hypothèses sans lesquelles il existe des «cas pathologiques» pour lesquels ces résultats ne s'appliquent pas. Aussi, ce développement nous a conduit à préciser certaines hypothèses et à expliciter tous les *détails* et mécanismes des preuves formalisées. Les principales difficultés rencontrées proviennent évidemment des différents renommages nécessaires dans une dérivation mais aussi du besoin d'expliciter *a priori* les hypothèses d'un énoncé à démontrer. L'explicitation des hypothèses de renommage sur les clauses utilisées nous a conduit à construire (i.e. à calculer) les substitutions de renommage utilisées en spécifiant un «véritable calcul» sur les variables de renommage : tout comme le concepteur d'un interprète PROLOG doit garantir certaines propriétés sur les noms des variables automatiquement engendrées lors de l'exécution d'un programme, les preuves formelles des propriétés sémantiques de la programmation logique doivent prendre en compte, de manière explicite, les propriétés des divers renommages effectués. Ces propriétés, spécifiées dans les définitions formelles du mécanisme d'exécution des programmes définis, constituent une étape essentielle lorsque l'on souhaite donner une interprétation procédurale à un langage purement déclaratif (le fragment Hornien de la logique des prédicats). La prise en compte de ces *détails* complique considérablement les preuves des résultats classiques et nous verrons, par exemple, que la preuve du théorème de complétude de la SLD-résolution est plus complexe qu'il n'y paraît et nécessite, d'un point de vue formel, plusieurs lemmes «techniques» (renommage d'une dérivation, combinaisons de plusieurs dérivations, ...). Il est important de noter que cette explicitation des hypothèses doit se faire *a priori*. Curieusement, il est courant, lorsque l'on prouve, «à la main», les résultats classiques de la programmation logique «sur le papier», de supposer, au fur et à mesure que l'on progresse dans une preuve, des propriétés sur le renommage des clauses. Bien entendu, il existe un théorème d'indépendance vis à vis du choix des variables de renommage, lorsque ce choix satisfait de «bonnes» propriétés : il est toujours possible de se placer dans une situation où ces «suppositions» sont vérifiées. Cependant, pour que ces «bonnes» propriétés soient satisfaites, il est nécessaire d'appliquer ce résultat d'indépendance en indiquant explicitement (i.e. en instanciant les objets qu'il met en jeu) en quoi il permet de supposer telle propriété sur les variables de renommage. Pourtant, ce théorème d'indépendance est rarement énoncé et son utilisation est souvent implicite (comment distinguer clairement ce qui est supposable de ce qui ne l'est pas ?) même s'il constitue, d'un point de vue formel, un résultat primordial de la programmation logique, au même titre que les théorèmes de validité et de complétude, qu'il permet de prouver rigoureusement. Pour obtenir une preuve formelle, il faut donc spécifier au départ toutes les hypothèses nécessaires à la preuve. Cette explicitation est parfois délicate puisque ces hypothèses doivent obligatoirement porter sur des objets présents dans l'énoncé à montrer et ne peuvent pas porter sur des objets construits dans la preuve. Par exemple, contrairement à un usage courant, on ne pourra pas supposer qu'une clause soit renommée *en dehors* des variables d'un objet (requête, domaine d'une substitution, ...) construit dans la preuve mais ne figurant pas dans l'énoncé initial : il faudra identifier cet ensemble de variables à partir des objets présents dans l'énoncé initial ou bien imposer certaines propriétés *a priori* à partir desquelles une telle supposition pourra être déduite. Le paragraphe 3.1 présente les quelques «*points de détail*» généralement omis et qui nécessitent quelques précautions lors d'une formalisation. La formalisation de la sémantique des programmes définis, présentée dans le chapitre 3 et dans [51, 53, 54, 56], explicite tous ces «*points de détail*» et pourra donc paraître fastidieuse.

Sémantique des dérivations infinies

Les résultats classiques de la SLD-résolution, formalisés dans la première partie, concernent les calculs finis (i.e. définissent une sémantique pour les SLD-réfutations). En effet, il existe une tradition en informatique qui veut que tout «bon» programme soit un programme dont l'exécution termine. La programmation logique n'échappe pas à cette tradition et les propriétés de terminaison des programmes logiques ont fait l'objet de nombreux travaux. Pour n'en citer qu'un, M. Bezem propose dans [11] une caractérisation d'une classe de programmes définis qui «terminent» (indépendamment de la stratégie de recherche et de la règle de séléction utilisée) à partir d'une large classe de buts. Ce développement repose sur la notion d'applications de niveaux[2] (*level mappings*) introduite par A. Cavendon et établit la terminaison des programmes récurrents[3] à partir de buts dont les instances ont un niveau inférieur à un certain niveau. Répondant aux mêmes préoccupations, R.N. Bol étudie dans [12] la possibilité d'introduire dans la SLD-résolution un mécanisme permettant la détection de certaines «boucles», conduisant à des dérivations infinies, afin de ne pas poursuivre la recherche dans ces cas (tout en garantissant que toutes les solutions finies restent constructibles).

Aujourd'hui, l'étude de la sémantique des «calculs» infinis apparaît dans de nombreux domaines de l'informatique : extension de la théorie des automates au domaine des mots infinis, représentations finies des réponses infinies aux requêtes dans les bases de données déductives [14], dérivations infinies dans des systèmes de réécriture [26], λ-calcul infini [59], dérivations infinies en programmation logique ... En effet, certains objets sont, par nature, infinis et les programmes qui permettent de les construire, même s'ils ne terminent pas, effectuent bien un calcul «utile en un certain sens». Cependant, dans le domaine de la programmation logique, toutes les dérivations infinies ne correspondent pas nécessairement à la construction d'un tel objet : certaines dérivations «fuient». Toutefois, même si ces dérivations ne «calculent» pas (i.e. ne construisent pas de termes infinis), elles «prouvent» (preuve infinie sur un objet fini) : par exemple, on peut voir la dérivation infinie obtenue à partir du programme $p(x) \leftarrow p(x)$ et de la requête $p(z)$ comme une preuve (infinie) de $\forall x\, p(x)$. Dans la littérature consacrée à la programmation logique, il n'existe pas une «sémantique classique», mais plusieurs approches non équivalentes. Dans la plupart d'entre elles, la dénotation d'un programme défini P est obtenue en considérant le plus grand point fixe d'un opérateur associé à P. Dans le chapitre 4, nous verrons que lorsque l'univers du discours contient des éléments infinis, l'utilisation d'un plus grand point fixe conduit à définir une

2. il s'agit d'applications de l'ensemble des atomes fermés $At_{\Sigma,\Pi}[\varnothing]$ dans $\mathbb{N}$

3. les programmes constitués de clauses pour lesquelles chaque instance fermée de la tête de clause a un niveau au moins supérieur au niveau des instances correspondantes des atomes du corps de la clause

sémantique valide mais non complète (i.e. certains atomes dans la dénotation de P ne sont pas «constructibles» par une SLD-dérivation à partir de P). Nous verrons que les propriétés classiques des définitions co-inductives, rappelées dans le chapitre 1, se révèleront utiles tant pour définir une nouvelle approche pour la sémantique des dérivations infinies que pour expliquer les résultats d'incomplétude obtenus dans d'autres approches existantes. Une sémantique valide et complète, présentée dans [57], sera définie pour les dérivations infinies qui ne «construisent» pas de termes infinis. Cette classe de dérivations infinies sera interprétée selon l'isomorphisme de Curry-Howard en identifiant ces dérivations infinies à des termes d'un type co-inductif.

Chapitre 1

Préliminaires

Les définitions (co-)inductives permettent de définir des ensembles à partir d'un ensemble de règles indiquant comment construire les éléments de ces ensembles. Ces définitions fournissent un outil privilégié pour décrire les objets manipulés par l'informatique et munissent les ensembles ainsi définis de schémas de (co-)induction permettant de raisonner sur ces ensembles. La première partie de ce chapitre est entièrement consacrée à la présentation des propriétés classiques des définitions (co-)inductives qui se révèleront très utiles lors de l'étude de la sémantique des programmes définis (la formalisation de certains de ces résultats dans le système Coq est présentée dans l'annexe A). Ensuite, le calcul des constructions (co-)inductives, vu au travers de l'assistant à la preuve Coq [19], est présenté. Il ne s'agit pas d'une présentation formelle (et encore moins d'une formalisation de ce calcul [9]) et nous n'abordons pas en profondeur les fondements théoriques de l'assistant à la preuve Coq : nous nous plaçons résolument du côté de l'utilisateur. Ce chapitre donne donc seulement une esquisse des concepts mis en jeu dans le calcul des constructions inductives à partir d'exemples exprimés dans le langage de spécification du système Coq.

1.1 Définitions (co-)inductives

1.1.1 Définitions inductives

Un des exemples les plus classiques d'ensemble défini inductivement est l'ensemble des théorèmes que l'on peut «dériver» à partir d'un système formel : ce point de vue permet de considérer une définition inductive comme la donnée d'un ensemble de règles, l'ensemble ainsi défini correspond au plus petit ensemble contenant tous les axiomes de ce système et clos par application des règles. Il est aussi possible de caractériser un ensemble défini inductivement à l'aide d'un opérateur monotone.

Définition 1.1 (Ensemble défini inductivement [4])

- Définition inductive à partir d'un ensemble de règles.
  Une règle est une paire (E, e) où E désigne un ensemble de prémisses et e désigne la conclusion. On note $e \leftarrow E$ cette règle. Etant donné un ensemble de règles Φ, un ensemble A est dit Φ-clos si pour toute règle $e \leftarrow E$ de Φ, $E \subseteq A$ implique $e \in A$. L'ensemble défini inductivement par un ensemble de règles Φ est l'intersection de tous les ensembles Φ-clos :
  $$\mathsf{Ind}(\Phi) = \bigcap\{A,\ A \textit{ est } \Phi\textit{-clos}\}$$
- Définition inductive à partir d'un opérateur monotone.
  Un opérateur φ de $2^{\mathcal{B}}$ dans $2^{\mathcal{B}}$ est monotone si $E_1 \subseteq E_2 \subseteq \mathcal{B} \Rightarrow \varphi(E_1) \subseteq \varphi(E_2)$. Etant donné un opérateur φ, un ensemble A est φ-clos si $\varphi(A) \subseteq A$. L'ensemble défini inductivement par un opérateur φ est l'intersection de tous les ensembles φ-clos :
  $$\mathsf{Ind}(\varphi) = \bigcap_{\varphi(A)\subseteq A\subseteq\mathcal{B}} A$$

A chaque ensemble de règles Φ est associé un «principe d'induction» : si P est une propriété définie sur $\mathsf{Ind}(\Phi)$, telle que pour chaque règle $e \leftarrow E$ de Φ, $(\forall e' \in E\ P(e')) \Rightarrow P(e)$, alors pour tout élément a de $\mathsf{Ind}(\Phi)$, $P(a)$ est vérifié. D'autre part, on montre que $\mathsf{Ind}(\Phi)$ est le plus petit ensemble Φ-clos :

Lemme 1.1 *Tout ensemble de règles Φ admet un ensemble Φ-clos et l'intersection d'ensembles Φ-clos quelconques est encore un ensemble Φ-clos.*

Preuve. Etant donné un ensemble de règles Φ, l'ensemble :

$$\mathcal{B} = \bigcup_{e\leftarrow E\in\Phi} \{\{e\} \cup E\}$$

est Φ-clos. Considérons à présent l'intersection $\bigcap_i A_i$ où les A_i sont des ensembles Φ-clos. Soit $e \leftarrow E \in \Phi$, si $E \subseteq \bigcap_i A_i$, alors $\forall i \geq 0\ E \subseteq A_i$ et puisque tous les A_i sont, par hypothèse, Φ-clos, on a $\forall i \geq 0\ e \in A_i$ et donc $e \in \bigcap_i A_i$. $\bigcap_i A_i$ est donc bien un ensemble Φ-clos. ◀

Les deux approches considérées dans la définition 1.1 sont équivalentes : tout ensemble défini inductivement à partir d'un ensemble de règles peut être défini à partir d'un opérateur monotone et *vice versa*. En effet, à tout ensemble de règles Φ peut être associé un opérateur monotone T_Φ de $2^{\mathcal{B}}$ dans $2^{\mathcal{B}}$ défini par :

$$\mathcal{B} = \bigcup_{e\leftarrow E\in\Phi} \{\{e\} \cup E\} \quad T_\Phi(A) = \{e \in \mathcal{B},\ \exists\ e \leftarrow E \in \Phi \quad E \subseteq A\} \tag{1.1}$$

Lemme 1.2 *T_Φ est un opérateur monotone.*

PREUVE. Soit A_1 et A_2 deux sous-ensembles de $\mathcal{B}$ tels que $A_1 \subseteq A_2$. Si $e \in T_\Phi(A_1)$ alors il existe une règle $e \leftarrow E \in \Phi$ telle que $E \subseteq A_1$ et on peut conclure puisqu'on a alors $E \subseteq A_2$. ◀

Réciproquement, à tout opérateur monotone φ peut être associé l'ensemble de règles Φ_φ défini par :

$$\Phi_\varphi = \{e \leftarrow E,\ E \subseteq \mathcal{B} \text{ et } e \in \varphi(E)\}$$

Le lemme qui suit permet d'établir les égalités :

$$\mathsf{Ind}(\varphi) = \mathsf{Ind}(\Phi_\varphi) \qquad \mathsf{Ind}(\Phi) = \mathsf{Ind}(T_\Phi) \tag{1.2}$$

Lemme 1.3

- *Si φ est un opérateur monotone, alors $A \subseteq \mathcal{B}$ est Φ_φ-clos si et seulement si A est φ-clos.*
- *Un ensemble A est Φ-clos si et seulement si A est T_Φ-clos.*

PREUVE.
• ($\Rightarrow$). Si $e \in \varphi(A)$, alors, par définition, il existe une règle $e \leftarrow E \in \Phi_\varphi$ telle que $E \subseteq A$. Puisque A est Φ_φ-clos, on a $E \subseteq A \Rightarrow e \in A$ ce qui permet de conclure. ($\Leftarrow$). Soit $e \leftarrow E$ une règle de Φ_φ et montrons que $E \subseteq A \Rightarrow e \in A$. Par définition, $e \in \varphi(E)$ et par monotonie $\varphi(E) \subseteq \varphi(A)$ ce qui permet de conclure puisque, par hypothèse, $\varphi(A) \subseteq A$.
• ($\Rightarrow$). Si $e \in T_\Phi(A)$, alors il existe une règle $e \leftarrow E \in \Phi$ telle que $E \subseteq A$ et puisque A est Φ-clos, on a $e \in A$ ce qui permet de conclure. ($\Leftarrow$). Soit $e \leftarrow E$ une règle de Φ et montrons que $E \subseteq A \Rightarrow e \in A$. Par définition, si $E \subseteq A$, alors $e \in T_\Phi(A)$ et puisque, par hypothèse, $T_\Phi(A) \subseteq A$, on a $e \in A$ ce qui permet de conclure. ◀

Enfin, $\mathsf{Ind}(\varphi)$ peut être caractérisé en terme de plus petit point fixe de φ :

Théorème 1.1 (Tarski [95]) *Si φ est monotone, alors $\mathsf{Ind}(\varphi) = \mathsf{lfp}(\varphi)$.*

PREUVE. Soit $A \subseteq \mathcal{B}$, puisque $\mathsf{Ind}(\varphi)$ est défini comme l'intersection de tous les ensembles φ-clos, on a $\varphi(A) \subseteq A \Rightarrow \mathsf{Ind}(\varphi) \subseteq A$ et, par monotonie, $\varphi(A) \subseteq A \Rightarrow \varphi(\mathsf{Ind}(\varphi)) \subseteq \varphi(A)$. Par transitivité, il vient alors $\varphi(A) \subseteq A \Rightarrow \varphi(\mathsf{Ind}(\varphi)) \subseteq A$ ce qui permet d'établir :

$$\varphi(\mathsf{Ind}(\varphi)) \subseteq \mathsf{Ind}(\varphi)$$

Réciproquement, puisque φ est monotone, on a $\varphi(\varphi(\mathsf{Ind}(\varphi))) \subseteq \varphi(\mathsf{Ind}(\varphi))$. $\varphi(\mathsf{Ind}(\varphi))$ est donc φ-clos ce qui permet d'établir :

$$\mathsf{Ind}(\varphi) \subseteq \varphi(\mathsf{Ind}(\varphi))$$

$\mathsf{Ind}(\varphi)$ est donc un point fixe de φ. Montrons que c'est le plus petit : soit A' un point fixe de φ. Par définition, on a $\varphi(A') \subseteq A'$ et donc A' est φ-clos. Il vient alors $\mathsf{Ind}(\varphi) \subseteq A'$ ce qui permet de conclure. ◀

Il existe un autre moyen permettant de caractériser les ensembles définis inductivement à partir d'un opérateur. Cette approche utilise la notion de puissances ordinales : étant donnés un opérateur φ de $2^{\mathcal{B}}$ dans $2^{\mathcal{B}}$ et un ordinal α, on définit $\varphi^{\uparrow\alpha}$ par :

$$\varphi^{\uparrow\alpha} = \begin{cases} \varnothing & \text{si } \alpha = 0 \\ \varphi(\varphi^{\uparrow\beta}) & \text{si } \alpha = \beta + 1 \\ \bigcup\limits_{\beta<\alpha} \varphi^{\uparrow\beta} & \text{si } \alpha \text{ est un ordinal limite} \end{cases}$$

Nous allons voir que si φ est $\uparrow$-continu, c'est à dire si, pour toute suite croissante $(E_n)_{n\geq 0}$ de sous-ensembles de $\mathcal{B}$:

$$\varphi(\bigcup_{n\geq 0} E_n) = \bigcup_{n\geq 0} \varphi(E_n)$$

alors $\mathsf{Ind}(\varphi) = \varphi^{\uparrow\omega}$. Pour ce faire, on montre tout d'abord le lemme suivant.

Lemme 1.4 *Soit φ un opérateur.*

1. *Si φ est monotone, alors pour tout ordinal α, $\varphi^{\uparrow\alpha} \subseteq \varphi^{\uparrow\alpha+1}$.*
2. *Si φ est $\uparrow$-continu, alors φ est monotone.*

Preuve. (1). Induction sur α. Pour $\alpha = 0$, on a clairement $\varphi^{\uparrow 0} = \varnothing \subseteq \varphi^{\uparrow 1}$. Si α est un ordinal successeur $\alpha = \beta + 1$, alors, par hypothèse d'induction, $\varphi^{\uparrow\beta} \subseteq \varphi^{\uparrow\beta+1}$. Puisque φ est monotone, il vient :

$$\varphi(\varphi^{\uparrow\beta}) = \varphi^{\uparrow\beta+1} = \varphi^{\uparrow\alpha} \subseteq \varphi(\varphi^{\uparrow\beta+1}) = \varphi(\varphi^{\uparrow\alpha}) = \varphi^{\uparrow\alpha+1}$$

Si α est un ordinal limite, alors, par hypothèse d'induction :

$$\forall \beta < \alpha \qquad \varphi^{\uparrow\beta} \subseteq \varphi^{\uparrow\beta+1}$$

Si $x \in \varphi^{\uparrow\alpha}$, alors, par définition, pour un ordinal β, $x \in \varphi^{\uparrow\beta}$. Par hypothèse d'induction, il vient $x \in \varphi^{\uparrow\beta+1}$, c'est à dire $x \in \varphi(\varphi^{\uparrow\beta})$. Cependant, $\varphi^{\uparrow\beta} \subseteq \bigcup_{\beta<\alpha} \varphi^{\uparrow\beta} = \varphi^{\uparrow\alpha}$, et par monotonie de φ, on peut conclure puisque :

$$\varphi(\varphi^{\uparrow\beta}) \subseteq \varphi(\bigcup_{\beta<\alpha} \varphi^{\uparrow\beta}) = \varphi(\varphi^{\uparrow\alpha}) = \varphi^{\uparrow\alpha+1}$$

(2). Soit φ un opérateur $\uparrow$-continu, et A_1 et A_2 deux parties de $\mathcal{B}$ telles que $A_1 \subseteq A_2$. Il est clair que $A_1 \subseteq A_2 \subseteq A_2 \subseteq \cdots \subseteq A_2 \subseteq \cdots$ est une séquence croissante et puisque φ est $\uparrow$-continu, on a $\varphi(A_1 \cup A_2) = \varphi(A_1) \cup \varphi(A_2)$. Enfin, puisque $A_1 \subseteq A_2$, on a $A_2 = A_1 \cup A_2$ et donc $\varphi(A_2) = \varphi(A_1) \cup \varphi(A_2)$ ce qui permet de conclure à $\varphi(A_1) \subseteq \varphi(A_2)$. ◀

Ce lemme permet de montrer le théorème :

Théorème 1.2 *Si φ est $\uparrow$-continu, alors $\mathsf{Ind}(\varphi) = \varphi^{\uparrow\omega} = \mathsf{lfp}(\varphi)$.*

PREUVE. Montrons tout d'abord que $\varphi^{\uparrow\omega}$ est un point fixe de φ. D'après le lemme 1.4.1, $\varphi^{\uparrow\omega} \subseteq \varphi^{\uparrow\omega+1}$. Aussi, montrons que $\varphi^{\uparrow\omega+1} \subseteq \varphi^{\uparrow\omega}$. Soit $x \in \varphi^{\uparrow\omega+1}$, toujours d'après le lemme 1.4.1, $(\varphi^{\uparrow n})_{n\geq 0}$ est une séquence croissante et puisque φ est $\uparrow$-continu et $\varphi^{\uparrow 0} = \varnothing$, il vient :

$$\varphi(\bigcup_{n\geq 0} \varphi^{\uparrow n}) = \bigcup_{n\geq 0} \varphi^{\uparrow n+1} = \bigcup_{n\geq 1} \varphi^{\uparrow n} = \bigcup_{n\geq 0} \varphi^{\uparrow n} = \varphi^{\uparrow\omega}$$

$\varphi^{\uparrow\omega}$ est donc un point fixe de φ. D'après le théorème 1.1, $\mathsf{Ind}(\varphi)$ est le plus petit point fixe de φ, il suffit donc de prouver que $\varphi^{\uparrow\omega} \subseteq \mathsf{Ind}(\varphi)$ ce qui revient à montrer :

$$\forall n \in \mathbb{N} \quad \varphi^{\uparrow n} \subseteq \mathsf{Ind}(\varphi)$$

Nous procédons par induction sur n. Pour $n = 0$, la propriété est triviale. Supposons maintenant que $\varphi^{\uparrow n} \subseteq \mathsf{Ind}(\varphi)$. Puisque φ est continu, d'après le lemme 1.4.2, φ est monotone et alors $\varphi(\varphi^{\uparrow n}) = \varphi^{\uparrow n+1} \subseteq \varphi(\mathsf{Ind}(\varphi)) = \mathsf{Ind}(\varphi)$. $\varphi^{\uparrow\omega}$ est donc bien le plus petit point fixe de φ. ◀

Sans l'hypothèse de continuité sur φ, ce théorème n'est plus vérifié pour un opérateur monotone quelconque. Toutefois, si l'opérateur monotone est finitaire, c'est à dire s'il vérifie pour toute suite croissante $(E_n)_{n\geq 0}$ de sous-ensembles de $\mathcal{B}$:

$$\varphi(\bigcup_{n\geq 0} E_n) \subseteq \bigcup_{n\geq 0} \varphi(E_n)$$

alors $\mathsf{Ind}(\varphi) = \varphi^{\uparrow\omega}$. En effet, on montre que :

Lemme 1.5 *Tout opérateur monotone et finitaire est $\uparrow$-continu.*

PREUVE. Soit φ un opérateur monotone et finitaire, il suffit de prouver que pour toute suite croissante $(E_n)_{n\geq 0}$ de sous-ensembles de $\mathcal{B}$, on a :

$$\bigcup_{n\geq 0} \varphi(E_n) \subseteq \varphi(\bigcup_{n\geq 0} E_n)$$

Soit $x \in \bigcup_{n\geq 0} \varphi(E_n)$, il existe un naturel k tel que $x \in \varphi(E_k)$. De plus, puisque $(E_n)_{n\geq 0}$ est une suite croissante, on a $E_k \subseteq \bigcup_{n\geq 0} E_n$. Par monotonie, il vient :

$$\varphi(E_k) \subseteq \varphi(\bigcup_{n\geq 0} E_n)$$

et alors $x \in \varphi(\bigcup_{n\geq 0} E_n)$ ce qui permet de conclure. ◀

Si l'opérateur φ correspond à l'opérateur T_Φ associé à un ensemble de règles Φ, alors φ est finitaire si l'ensemble des prémisses de chaque règle de Φ est fini, et dans ce cas on dit que Φ est finitaire. En effet, on montre que :

Lemme 1.6 *Si Φ est finitaire, alors T_Φ est finitaire.*

PREUVE. Soit $(E_n)_{n\geq 0}$ une suite croissante de sous-ensembles de $\mathcal{B}$ et e un élément de $T_\Phi(\bigcup_{n\geq 0} E_n)$. Par définition, il existe une règle $e \leftarrow F \in \Phi$ telle que $F \subseteq \bigcup_{n\geq 0} E_n$. Puisque $(E_n)_{n\geq 0}$ est une suite croissante, c'est qu'il existe un entier k tel que $F \subseteq E_k$ et donc $e \in T_\Phi(E_k) \subseteq \bigcup_{n\geq 0} T_\Phi(E_n)$ ce qui permet de conclure. ◀

1.1.2 Définitions co-inductives

Les définitions co-inductives permettent de définir des ensembles dont les éléments sont obtenus en appliquant un nombre de fois éventuellement infini les règles de construction. Comme pour les ensembles définis inductivement, il est possible de définir un ensemble de manière co-inductive à partir d'un ensemble de règles Φ ou d'un opérateur φ.

Définition 1.2 (Ensemble défini co-inductivement [4])

- Définition co-inductive à partir d'un ensemble de règles.
  Etant donné un ensemble de règles Φ, un ensemble A est dit Φ-dense si pour tout $e \in A$, il existe un ensemble $E \subseteq A$ tel que $e \leftarrow E \in \Phi$. L'ensemble défini co-inductivement par un ensemble de règles Φ est l'union de tous les ensembles Φ-denses :

$$\mathsf{CoInd}(\Phi) = \bigcup\{A,\ A \textit{ est } \Phi\textit{-dense}\}$$

- Définition co-inductive à partir d'un opérateur monotone.
  Etant donné un opérateur φ, un ensemble A est φ-dense si $A \subseteq \varphi(A)$. L'ensemble défini co-inductivement par un opérateur φ est l'union de tous les ensembles φ-denses :

$$\mathsf{CoInd}(\varphi) = \bigcup_{A\subseteq\varphi(A)\subseteq\mathcal{B}} A$$

$\mathsf{CoInd}(\Phi)$ est le plus grand ensemble Φ-dense. Les deux approches considérées dans cette définition sont équivalentes : tout ensemble défini de manière co-inductive à partir d'un ensemble de règles peut être défini à partir d'un opérateur monotone et *vice versa*. En effet, le lemme suivant permet d'établir les égalités :

$$\mathsf{CoInd}(\varphi) = \mathsf{CoInd}(\Phi_\varphi) \qquad \mathsf{CoInd}(\Phi) = \mathsf{CoInd}(T_\Phi)$$

Lemme 1.7

- *Si φ est un opérateur monotone, alors $A \subseteq \mathcal{B}$ est Φ_φ-dense si et seulement si A est φ-dense.*
- *Un ensemble A est Φ-dense si et seulement si A est T_Φ-dense.*

Preuve.
• ($\Rightarrow$). Soit A un ensemble Φ_φ-dense et $a \in A$. Par définition, il existe une règle $a \leftarrow E \in \Phi_\varphi$ telle que $E \subseteq A$. Par monotonie, il vient $\varphi(E) \subseteq \varphi(A)$. D'autre part, par définition de Φ_φ, on a $a \in \varphi(E)$ et il vient $a \in \varphi(A)$ ce qui permet de montrer $A \subseteq \varphi(A)$. ($\Leftarrow$). Soit $A \subseteq \varphi(A)$ et $a \in A$. Par hypothèse, $a \in \varphi(A)$ et donc, par définition, il existe une règle $a \leftarrow E \in \Phi_\varphi$ telle que $E \subseteq A$. A est donc bien Φ_φ-dense.
• ($\Rightarrow$). Soit A un ensemble Φ-dense et $a \in A$. Par hypothèse, il existe un ensemble $E \subseteq A$ tel que $a \leftarrow E \in \Phi$ et donc $a \in T_\Phi(A)$. A est donc bien T_Φ-dense. ($\Leftarrow$). Soit $a \in A$. Par hypothèse, $a \in T_\Phi(A)$ et il existe donc une règle $a \leftarrow E \in \Phi$ telle que $E \subseteq A$. A est donc bien Φ-dense. ◀

D'autre part, $\mathsf{CoInd}(\varphi)$ peut être caractérisé en terme de plus grand point fixe de φ :

Théorème 1.3 (Tarski) *Si φ est monotone, alors* $\mathsf{CoInd}(\varphi) = \mathsf{gfp}(\varphi)$.

Preuve. Soit $A \subseteq \mathcal{B}$, puisque $\mathsf{CoInd}(\varphi)$ est défini comme l'union de tous les ensembles φ-denses, on a $A \subseteq \varphi(A) \Rightarrow A \subseteq \mathsf{Coind}(\varphi)$ et, puisque φ est un opérateur monotone, il vient $A \subseteq \varphi(A) \Rightarrow \varphi(A) \subseteq \varphi(\mathsf{Coind}(\varphi))$. Par transitivité $A \subseteq \varphi(A) \Rightarrow A \subseteq \varphi(\mathsf{Coind}(\varphi))$ ce qui permet de conclure à :

$$\mathsf{Coind}(\varphi) \subseteq \varphi(\mathsf{Coind}(\varphi))$$

Réciproquement, puisque φ est monotone, $\varphi(\mathsf{Coind}(\varphi)) \subseteq \varphi(\varphi(\mathsf{Coind}(\varphi)))$. $\varphi(\text{Coind}(\varphi))$ est donc φ-dense ce qui permet d'établir :

$$\varphi(\mathsf{Coind}(\varphi)) \subseteq \mathsf{Coind}(\varphi)$$

$\mathsf{Coind}(\varphi)$ est donc bien un point fixe de φ. Montrons que c'est le plus grand : soit A' un point fixe de φ, par définition, on a $A' \subseteq \varphi(A')$. Aussi A' est φ-dense et il vient $A' \subseteq \mathsf{Coind}(\varphi)$ ce qui permet de conclure. ◀

Il est aussi possible de définir un ensemble de manière co-inductive en terme de puissances ordinales d'un opérateur : étant donnés un opérateur φ de $2^{\mathcal{B}}$ dans $2^{\mathcal{B}}$ et un ordinal α, on définit $\varphi^{\downarrow\alpha}$ par :

$$\varphi^{\downarrow\alpha} = \begin{cases} \mathcal{B} & \text{si } \alpha = 0 \\ \varphi(\varphi^{\downarrow\beta}) & \text{si } \alpha = \beta + 1 \\ \bigcap\limits_{\beta<\alpha} \varphi^{\downarrow\beta} & \text{si } \alpha \text{ est un ordinal limite} \end{cases}$$

Nous allons voir que si φ est $\downarrow$-continu, c'est à dire si, pour toute suite décroissante $(E_n)_{n\geq 0}$ de sous-ensembles de $\mathcal{B}$:

$$\varphi(\bigcap_{n\geq 0} E_n) = \bigcap_{n\geq 0} \varphi(E_n)$$

alors $\mathsf{CoInd}(\varphi) = \varphi^{\downarrow\omega}$. Pour ce faire, on montre tout d'abord le lemme suivant.

Lemme 1.8 *Soit φ un opérateur.*

1. *Si φ est monotone, alors pour tout naturel n, $\varphi^{\downarrow n+1} \subseteq \varphi^{\downarrow n}$.*
2. *Si φ est $\downarrow$-continu, alors φ est monotone.*

Preuve. (1). Induction sur n. Pour $n = 0$, la propriété est triviale. Si $n = k + 1$, alors par hypothèse d'induction $\varphi^{\downarrow k+1} \subseteq \varphi^{\downarrow k}$ et il vient par monotonie :

$$\varphi^{\downarrow n+1} = \varphi(\varphi^{\downarrow n}) = \varphi(\varphi^{\downarrow k+1}) \subseteq \varphi(\varphi^{\downarrow k}) = \varphi^{\downarrow k+1} = \varphi^{\downarrow n}$$

(2). Soit φ un opérateur $\downarrow$-continu, et A_1 et A_2 deux parties de $\mathcal{B}$ telles que $A_1 \subseteq A_2$. Il est clair que $A_2 \supseteq A_1 \supseteq A_1 \supseteq \cdots \supseteq A_1 \supseteq \cdots$ est une séquence décroissante et puisque φ est $\downarrow$-continu, on a $\varphi(A_1 \cap A_2) = \varphi(A_1) \cap \varphi(A_2)$. Enfin, puisque $A_1 \subseteq A_2$, on a $A_1 = A_1 \cap A_2$ et donc $\varphi(A_1) = \varphi(A_1) \cap \varphi(A_2)$ ce qui permet de conclure à $\varphi(A_1) \subseteq \varphi(A_2)$. ◀

La première assertion de ce lemme est plus faible que celle correspondant au lemme 1.4. Ce lemme permet de montrer le théorème :

Théorème 1.4 *Si φ est $\downarrow$-continu, alors $\mathsf{CoInd}(\varphi) = \varphi^{\downarrow\omega} = \mathsf{gfp}(\varphi)$.*

Preuve. Montrons d'abord que $\varphi^{\downarrow\omega}$ est un point fixe de φ. Puisque φ est $\downarrow$-continu, d'après le lemme 1.8, φ est monotone et $(\varphi^{\downarrow n})_{n\geq 0}$ est une suite décroissante. Par conséquent :

$$\varphi^{\downarrow\omega+1} = \varphi(\bigcap_{n\geq 0} \varphi^{\downarrow n}) = \bigcap_{n\geq 0} \varphi(\varphi^{\downarrow n}) = \bigcap_{n\geq 0} \varphi^{\downarrow n+1}$$

En appliquant encore le lemme 1.8, il vient :

$$\bigcap_{n\geq 0} \varphi^{\downarrow n+1} = \bigcap_{n\geq 0} \varphi^{\downarrow n} = \varphi^{\downarrow\omega}$$

$\varphi^{\downarrow\omega}$ est donc un point fixe de φ. Montrons qu'il s'agit du plus grand point fixe. D'après le théorème de Tarski, $\varphi^{\downarrow\omega} \subseteq \mathsf{Coind}(\varphi)$. Il suffit donc de prouver que $\mathsf{CoInd}(\varphi) \subseteq \varphi^{\downarrow\omega}$ ce qui revient à montrer :

$$\forall n \in \mathbb{N} \quad \mathsf{CoInd}(\varphi) \subseteq \varphi^{\downarrow n}$$

Nous procédons par induction sur n. Pour $n = 0$, la propriété est triviale. Supposons maintenant que $\mathsf{CoInd}(\varphi) \subseteq \varphi^{\downarrow n}$. Puisque φ est $\downarrow$-continu, φ est monotone et on a alors :

$$\mathsf{CoInd}(\varphi) = \varphi(\mathsf{CoInd}(\varphi)) \subseteq \varphi(\varphi^{\downarrow n}) = \varphi^{\downarrow n+1}$$

ce qui permet de conclure. ◀

1.2 Coq et le calcul des constructions inductives

1.2.1 Isomorphisme de Curry-Howard

Si le raisonnement n'est pas réductible au calcul, en revanche, la vérification de la validité du raisonnement l'est. En effet, la correction d'un raisonnement ne dépend que de sa forme et la proposition «*p est une preuve de la proposition P*» est vérifiable par le calcul. Muni d'un formalisme adéquat, on peut donc obtenir des programmes de vérification automatique de preuves.

Sémantique BHK

La sémantique de BHK (L.E.J. Brouwer, A. Heyting et A.N. Kolmogorov) propose d'interpréter les preuves de propositions par des objets fonctionnels. A chaque proposition est associé l'ensemble, éventuellement vide, de ses preuves. Cette sémantique s'exprime par :

- Une preuve de la proposition False est un élément de l'ensemble vide.
- Une preuve de $A \wedge B$ est un élément de $A \times B$ formé d'une preuve de A et d'une preuve de B.
- Une preuve de $A \vee B$ est une paire dont la seconde composante est soit une preuve de A soit une preuve de B et dont la première composante est un «indicateur» permettant de savoir si l'on dispose d'une preuve de A ou de B.
- Une preuve de $A \Rightarrow B$ est une fonction f qui à chaque preuve a de A associe une preuve $f(a)$ de B.
- Une preuve de la proposition existentielle $\exists x \in A\, B(x)$ est un objet composé d'un élément $x \in A$ et d'une preuve de $B(x)$.
- Une preuve de la proposition universelle $\forall x \in A\, B(x)$ est une fonction f qui associe à tout objet $x \in A$, une preuve de $B(x)$.

Un formalisme pour représenter des preuves

Le calcul des constructions [21] est un formalisme permettant de représenter et de manipuler de manière explicite des preuves via l'isomorphisme de Curry-Howard. Ce formalisme repose sur la théorie des types, introduite par P. Martin-Löf [72], et sur le λ-calcul typé [8]. Le calcul des constructions ne dispose que d'une seule classe syntaxique de termes permettant de représenter aussi bien les types que les objets typés :

$$
\begin{array}{llcl}
t ::= & x & | & \text{variables} \\
& s & | & \text{sortes} \\
& c & | & \text{constantes introduites dans l'environnement} \\
& (t_1\ t_2) & | & \text{application du terme } t_1 \text{ au terme } t_2 \\
& [x{:}\,t_1]t_2 & | & \text{fonction qui à tout } x \text{ de type } t_1 \text{ associe } t_2 \\
& (x{:}\,t_1)t_2 & . & \text{produit}
\end{array}
$$

L'ensemble de sortes disponibles est :

$$\mathcal{S} = \{\mathsf{Prop}, \mathsf{Set}, \mathsf{Type}(i),\ i \in \mathbb{N}\}$$

Ces sortes sont organisées hiérarchiquement comme suit :

$$
\begin{array}{rcl}
\mathsf{Prop} & \text{est de type} & \mathsf{Type}(0) \\
\mathsf{Set} & \text{est de type} & \mathsf{Type}(0) \\
\mathsf{Type}(i) & \text{est de type} & \mathsf{Type}(i+1)
\end{array}
$$

Les termes du calcul des constructions sont utilisés pour représenter des types de données, des propriétés et des preuves. C'est le typage des termes qui permet de les classifier : Set est utilisé pour définir formellement des collections d'objets et Prop est utilisé pour exprimer des propriétés formelles sur ces objets. En utilisant l'interprétation donnée par l'isomorphisme de Curry-Howard, présenté plus bas, le jugement de typage $\Gamma \vdash t{:}\,T$, exprimant qu'un terme t a pour type T dans un contexte Γ, admet alors deux lectures différentes selon que T soit de type Prop ou Set : dans le premier cas, t désigne une preuve de la proposition T (on parle de type habité ou de proposition prouvable), dans le deuxième cas, t est un objet de la collection T (on parle d'objet «calculatoire»). La principale règle de réduction du calcul des constructions est la β-réduction qui permet l'élimination conjointe d'une abstraction et d'une application : le terme $(\lambda x{:}\,T.t \quad u)$ (noté dans le calcul des constructions $([x{:}\,T]t \quad u)$) se β-réduit en un terme noté $t[x \leftarrow u]$ et correspondant au terme t dans lequel on substitue toutes les occurrences de x par le terme u.

$$(\lambda x{:}\,T.t \quad u) \rightarrow_\beta t[x \leftarrow u]$$

Initialement utilisée pour exprimer un calcul fonctionnel, la notation du λ-calcul permet, dans sa version typée, de représenter les preuves exprimées dans le formalisme de déduction naturelle, dû à G. Gentzen (1934) [85]. Cette correspondance repose sur l'isomorphisme de Curry-Howard [35, 37] qui établit un lien fort, d'une part, entre preuves et λ-termes, et d'autre part, entre propositions et types :

$$
\begin{array}{rcccl}
\text{preuve} & \equiv & \lambda\text{-terme} & \equiv & \text{programme} \\
\text{proposition} & \equiv & \text{type} & \equiv & \text{spécification}
\end{array}
$$

Preuves et algorithmes admettent donc une représentation uniforme, ce qui s'exprime par le slogan «*programmer = prouver*» qui permet d'«unifier» les activités de programmation et de démonstration. Ecrire un programme, c'est prouver, de manière constructive, une proposition ; déterminer le type d'une expression, c'est trouver de quelle proposition cette expression est une preuve. Par exemple, la fonction identité sur T définie par $[x\colon T]x$ est un λ-terme de type $T \to T$ et correspond à une preuve de la proposition $T \Rightarrow T$. Les symboles $\to$ (langage fonctionnel) et $\Rightarrow$ (langage logique) sont «unifiés».

λ-calcul	λ-terme $[x : T]x$	(de) type $T \to T$
logique	preuve (de	la) proposition $T \Rightarrow T$

Aussi, si l'on reconsidère la sémantique BHK décrite plus haut, on obtient :

- Une preuve de $A \Rightarrow B$ est un terme $f\colon A \to B$ qui à chaque preuve a de A associe une preuve $f(a)$ de B (cas non dépendant).
- Une preuve de la proposition existentielle $\exists x \in A\, B(x)$ est un élément de $\sum_{x\in A} B(x)$: le type somme $\sum_{x\in A} B$ généralise $A \times B$ lorsque le type du deuxième élément dépend de la valeur du premier.
- Une preuve de la proposition universelle $\forall x \in A\, B(x)$ est un terme $f\colon \prod_{x\in A} B(x)$: le type produit $\prod_{x\in A} B$ généralise $A \to B$ lorsque le type de la valeur retournée dépend de la valeur de l'argument.

Vérification d'une preuve et typage d'un terme : un exemple

C'est la décidabilité du typage des termes du calcul des constructions qui permet la vérification (automatique) de la validité d'une preuve (la preuve n'étant pas obtenue de manière automatique). Illustrons cela sur un exemple. Le formalisme de déduction naturelle, exprimé à l'aide de séquents, manipule des jugements de la forme $\Gamma \vdash A$ que l'on peut interpréter par «*A est une conséquence des hypothèses du contexte* Γ». Pour construire des preuves dans ce formalisme, on utilise des règles d'inférence permettant d'introduire ou de supprimer des connecteurs (ou des quantificateurs). L'exemple que nous présentons fait appel aux trois règles suivantes :

$$(\mathrm{Ax}) : \frac{}{\Gamma, A \vdash A} \qquad (I_{\Rightarrow}) : \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad (E_{\Rightarrow}) : \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

(Ax) correspond à un axiome et $(I_{\Rightarrow})$ (resp. $(E_{\Rightarrow})$) correspond à la règle d'introduction[1] (resp. d'élimination) du connecteur $\Rightarrow$. Muni de ces trois

1. dans laquelle l'hypothèse A est «déchargée»

$$
(I_\Rightarrow):\dfrac{(I_\Rightarrow):\dfrac{(I_\Rightarrow):\dfrac{(E_\Rightarrow):\dfrac{(\mathrm{Ax}):\dfrac{}{A \Rightarrow B, B \Rightarrow C, A \vdash B \Rightarrow C} \quad (E_\Rightarrow):\dfrac{(\mathrm{Ax}):\dfrac{}{A \Rightarrow B, B \Rightarrow C, A \vdash A \Rightarrow B}\;(\mathrm{Ax}):\dfrac{}{A \Rightarrow B, B \Rightarrow C, A \vdash A}}{A \Rightarrow B, B \Rightarrow C, A \vdash B}}{A \Rightarrow B, B \Rightarrow C, A \vdash C}}{A \Rightarrow B, B \Rightarrow C \vdash A \Rightarrow C}}{A \Rightarrow B \vdash (B \Rightarrow C) \Rightarrow (A \Rightarrow C)}}{\vdash (A \Rightarrow B) \Rightarrow ((B \Rightarrow C) \Rightarrow (A \Rightarrow C))}
$$

FIG. 1.1 – *Arbre de preuve en déduction naturelle*

règles, il est possible de construire une preuve de la célèbre tautologie :

$$(A \Rightarrow B) \Rightarrow ((B \Rightarrow C) \Rightarrow (A \Rightarrow C)) \tag{1.3}$$

dont l'arbre de preuve est présenté sur la figure 1.1. D'autre part, si l'on considère les jugements de typage des termes du calcul des constructions, de la forme $\Gamma \vdash t\colon T$, qui s'interprètent par : «*dans le contexte* Γ*, le terme* t *est de type* T», on peut établir le type d'un terme à l'aide de règles de typage. Les trois règles que nous utilisons dans l'exemple présenté sont :

$$(\mathrm{Var}) : \frac{}{\Gamma, x\colon A \vdash x\colon A}$$

$$(\mathrm{Abs}) : \frac{\Gamma, x\colon A \vdash t\colon B}{\Gamma \vdash (\lambda x\colon A.t)\colon A \to B} \qquad (\mathrm{App}) : \frac{\Gamma \vdash t_1\colon A \to B \quad \Gamma \vdash t_2\colon A}{\Gamma \vdash (t_1\ t_2)\colon B}$$

La règle (Abs) (resp. (App)) correspond à la construction du type d'une abstraction (resp. d'une application). La correspondance de Curry-Howard permet d'identifier types et propositions et les règles de typage (Var), (Abs) et (App) correspondent exactement aux règles de la déduction naturelle (Ax), $(I_\Rightarrow)$ et $(E_\Rightarrow)$ «décorées» par des λ-termes. Aussi, en «décorant» l'arbre de preuve de la tautologie (1.3), on obtient le jugement de typage :

$$\begin{array}{ll} \vdash & \lambda x\colon A \to B.\lambda y\colon B \to C.\lambda z\colon A.(y\ (x\ z))\colon \\ & (A \to B) \to ((B \to C) \to (A \to C)) \end{array}$$

où le terme $\lambda x\colon A \to B.\lambda y\colon B \to C.\lambda z\colon A.(y\ (x\ z))$ représente une preuve de la tautologie (1.3). Il existe donc un lien direct entre le λ-calcul typé et le formalisme de la déduction naturelle.

déduction naturelle	λ-calcul typé
proposition	type
dérivation (preuve)	λ-terme
conclusion d'une dérivation	type habité
décharge d'une hypothèse	λ-abstraction

1.2.2 L'assistant à la preuve Coq

L'isomorphisme de Curry-Howard permet de vérifier automatiquement la validité d'une preuve formelle. Cependant, l'écriture des termes de preuve dans le calcul des constructions semble fastidieuse et laborieuse. En effet, dès que la preuve à écrire dépasse «quelques pas d'inférence», l'écriture du λ-terme correspondant n'est plus humainement envisageable et il est souhaitable de disposer de procédures permettant de construire automatiquement le λ-terme de preuve en fonction d'indications fournissant le raisonnement à mener pour établir la preuve. D'autre part, certaines parties d'une preuve étant plus faciles à obtenir que d'autres, il est raisonnable d'envisager des procédures automatiques permettant de les établir. De tels programmes, appelés des assistants à la preuve, constituent une alternative au «tout ou rien» (démonstration automatique ou vérification automatique). Les plus connus sont Coq, Isabelle, Lego, Hol, Lcf, Alf, Pvs, ... Coq permet de construire des preuves formelles pour une logique d'ordre supérieur en utilisant un assistant à la preuve. L'arbre de preuve obtenu (i.e. le λ-terme) est construit, de manière incrémentale et interactive, en partant du but initial et en le décomposant suivant des règles logiques à l'aide d'un ensemble de tactiques prédéfinies : une tactique est une fonction qui construit une preuve d'un but donné à partir de preuves «élémentaires» de sous-buts (chaque sous-but étant associé à un contexte local d'hypothèses). Etant donné un but à prouver, l'application d'une tactique engendre donc les sous-buts, associés à des contextes d'hypothèses, qu'il reste à montrer :

$$\frac{\text{contexte } 0}{\text{But}} \quad \overset{\substack{\text{Application} \\ \text{d'une tactique}}}{\longrightarrow} \quad \frac{\text{contexte } 1}{\text{Sous-but } 1}, \ldots, \frac{\text{contexte } n}{\text{Sous-but } n}$$

Par exemple, étant donné un contexte d'hypothèses H et un but B à prouver, la tactique «`Cut` A» engendrera deux sous-buts : le premier, où il faudra montrer, dans le même contexte H, que $A \Rightarrow B$, et, le deuxième, où il faudra prouver A dans le contexte H.

$$\frac{H}{B} \overset{\texttt{Cut}\ A}{\longrightarrow} \frac{H}{A \rightarrow B}, \frac{H}{A} \tag{1.4}$$

La preuve d'un lemme ainsi codée constitue une preuve formelle (ou vérifiable ou explicite) de ce lemme et se présente sous la forme d'un «texte» écrit dans un langage formel, dont la lecture est, au mieux, laborieuse et, la plupart du temps, vaine. De manière (très) imagée, on peut comparer la différence qui existe entre une preuve formelle et une preuve «sur le papier» avec la différence qui existe, pour un logiciel, entre un «manuel de référence» et un «guide de l'utilisateur». Le premier a (ou devrait avoir) pour objectif d'être exhaustif et rigoureux, tandis que le second a pour objectif d'être didactique : sur des sujets complexes, comme les langages de programmation, il est quasiment impossible d'être à la fois rigoureux et didactique. C'est pourquoi, les démonstrations présentées dans ce qui suit, ont été re-transcrites dans le langage «naturel». Il s'agit bien sûr du résultat d'un compromis, puisque de manière «idéale», chaque étape d'une preuve devrait découler explicitement de l'application d'un lemme déjà établi ou d'une hypothèse posée au départ. Par souci de lisibilité, certains passages de preuves sont omis et nécessitent donc quelques connaissances élémentaires «implicites». D'un point de vue pratique, COQ est écrit en CAML, un langage de programmation fonctionnelle de la famille ML, développé à l'INRIA. CAML est aussi le langage «cible» dans lequel sont écrits les programmes extraits à partir des preuves.

1.2.3 Types inductifs

Le calcul des constructions a été étendu afin de permettre la définition de types inductifs et co-inductifs. De manière informelle, définir un type de manière inductive ou co-inductive revient à donner les règles de formation des éléments de ce type et à exprimer qu'un élément est de ce type uniquement s'il a été obtenu par application de ces règles. Pour une présentation plus théorique des définitions inductives, on pourra consulter [4, 23]. L'introduction de types inductifs dans le calcul des constructions est présentée dans [79, 97]. L'exemple le plus classique de définition inductive est la définition de l'ensemble des entiers naturels, défini (inductivement) par le plus petit ensemble satisfaisant : 0 est un entier naturel et si n est un entier naturel, alors le successeur de n, noté $S(n)$, est aussi un entier naturel (l'ensemble est clos par successeur). Cette définition s'exprime dans le langage de spécification de COQ par :

```
Inductive nat : Set := O : nat | S : nat ->nat.
```
(1.5)

Un type inductif est donc défini par la donnée d'un ensemble de constructeurs. Il existe toutefois une restriction sur le type des constructeurs qui permet, par exemple, de rejeter des constructeurs de type $(T \to T) \to T$ dans lequel la première occurrence de T correspond à une occurrence non «positive» de T (la notion d'occurrence positive est définie, par exemple, dans [79]). Cette restriction est nécessaire pour garantir la propriété de normalisation et la cohérence logique du système. Le mot clé `Inductive` sert à spécifier la nature inductive du type défini et permettra à COQ d'engendrer automatiquement les schémas d'induction (aussi appelés schémas d'élimination) associés à ce type. Les trois «principes» d'induction engendrés par la définition de `nat` sont :

nat_*id* : `(P:nat->`*s*`) (P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)`

où (id, s) peut prendre les valeurs (`ind,Prop`) ou (`rec,Set`) ou (`rect,Type`) ; `nat_ind` correspond au neuvième axiome de Péano.

Destructeurs sur un type inductif

L'opérateur `Case` permet d'exploiter le fait qu'un objet de type inductif ne peut être construit qu'en utilisant, un nombre de fois fini, les constructeurs présents dans la définition du type et permet de mener un raisonnement par cas sur ces objets. Sa syntaxe est :

`<` P `>` `Case` x `of` $f_1 \cdots f_n$ `end.`

où le terme x appartient à un type inductif spécifié par n constructeurs. P est le type de l'expression et chaque f_i représente un terme permettant d'obtenir un objet de type P dans le cas où x a été construit avec le i-ième constructeur. Cet opérateur permet notamment de définir des fonctions ou des prédicats de manière récursive. Par exemple, on peut définir récursivement le prédicat `IS_EVEN:nat->Prop`, caractérisant les entiers pairs, par :

```
Fixpoint IS_EVEN [n:nat] : Prop :=
<Prop>Case n of
(* n=0     *) True
(* n=(S k) *) [k:nat]<Prop>Case k of
                (* k=0     *)   False
                (* k=(S p) *)   [p:nat](IS_EVEN p)
                                end
        end.
```

`(*` et `*)` délimitent les commentaires ; le type indiqué entre `<` et `>` indique le type du résultat (nous verrons par la suite qu'il existe une restriction sur ce type) ; `False` et `True` sont deux constantes de type `Prop`.

Prédicats inductifs

Il est aussi possible de définir des prédicats de manière inductive. Par exemple, le prédicat `even:nat->Prop`, caractérisant les entiers pairs, peut se définir inductivement par :

```
Inductive even : nat ->Prop :=
even_0 : (even 0) |                                    (1.6)
even_S2 : (n:nat)(even n) ->(even (S (S n))).
```

Toutefois, seul le schéma d'élimination `even_ind` est engendré par cette définition (nous en évoquerons plus loin la raison). Bien entendu, cette définition inductive est équivalente à la définition récursive présentée dans le paragraphe précédent dans le style de la programmation fonctionnelle : tout entier naturel n vérifie $\texttt{even}(n)$ si et seulement si $\texttt{IS\_EVEN}(n) = \texttt{True}$. Cette définition est exprimée dans un style «*à la* PROLOG» (chaque constructeur correspond à une clause) et correspond à la plus petite relation satisfaisant l'ensemble des règles d'inférence :

$$\left\{ (\texttt{even-0}) : \frac{}{\texttt{even}(0)} \right\} \bigcup \left\{ (\texttt{even-S2-}n) : \frac{\texttt{even}(S^n(0))}{\texttt{even}(S^{n+2}(0))},\ n \in \mathbb{N} \right\} \quad (1.7)$$

Ici, les constructeurs peuvent être vus comme des règles d'introduction. Certains connecteurs logiques sont d'ailleurs définis de manière inductive, et dans ce cas, les constructeurs correspondent aux règles d'introduction du connecteur logique. Par exemple, la conjonction et la disjonction sont définies dans le système Coq à l'aide des constructeurs `conj`, `or_introl`, et `or_intror`, qui permettent l'introduction du connecteur et correspondent respectivement aux règles $(I_\wedge)$, $(I_\vee^l)$ et $(I_\vee^r)$:

```
Inductive and [A:Prop;B:Prop]:Prop:=
conj : A ->B ->(and A B).
```

$$(I_\wedge) : \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

```
Inductive or [A:Prop;B:Prop]:Prop:=
or_introl:A->(or A B) |
or_intror:B->(or A B).
```

$$(I_\vee^l) : \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad (I_\vee^r) : \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \tag{1.8}$$

Raisonnement par induction avec l'assistant à la preuve Coq

Les schémas d'induction engendrés lors de la définition d'un type inductif peuvent être utilisés lors de l'application de tactiques. Par exemple, si l'on doit montrer une propriété P: `nat` $\rightarrow$ `Prop` pour tout entier n dans un contexte d'hypothèses H, la tactique «`Induction` n» mettra en jeu le schéma d'induction `nat_ind`, et engendrera deux sous-buts. Le premier, où il faudra montrer, dans le contexte H, la propriété P pour 0 ; et le deuxième, où dans un contexte constitué des hypothèses de H, d'un entier n_0 et d'une preuve H_0 de la propriété P pour n_0, il faudra montrer la propriété P pour le successeur de n_0.

$$\frac{H}{(n\colon \mathtt{nat})(P\,n)} \quad \overset{\mathtt{Induction}\ n}{\longrightarrow} \quad \frac{H}{(P\,0)}, \frac{\begin{array}{c}H\\ n_0\colon \mathtt{nat}\\ H_0\colon (P\,n_0)\end{array}}{(P\,(S\,n_0))}$$

De manière similaire, la tactique `Elim` permet d'utiliser les schémas d'élimination associés à un type inductif. Par exemple, en utilisant le connecteur de disjonction défini en (1.8), la tactique `Elim` peut être appliquée à une hypothèse exprimant une disjonction comme suit :

$$\frac{H\colon A \vee B}{C} \overset{\mathtt{Elim}\ H}{\longrightarrow} \frac{H\colon A \vee B}{A \rightarrow C}, \frac{H\colon A \vee B}{B \rightarrow C}$$

ce qui correspond au fait que H a été obtenue en utilisant un des deux constructeurs du type `or`.

1.2.4 Types polymorphes et types dépendants

Puisque les types sont aussi des termes, il est possible de construire des abstractions sur un type. Par exemple, alors que le terme fonctionnel `[x:T]x` correspond à la fonction identité sur `T` et est de type `T -> T`, le terme `[T:Set][x:T]x` correspond à l'identité polymorphe et est de type `(T:Set)(T -> T)`, où la notation $(x{:}\,P)Q$ est une extension de $P \rightarrow Q$. Cette construction permet de définir des types «génériques». Un autre exemple classique de définition polymorphe est la définition du type paramétré des listes :

```
Inductive list [A:Set]:Set :=
nil:(list A) | cons:A->(list A)->(list A).
```
(1.9)

Similairement, nous pouvons définir un type paramétré par un autre type, mais aussi dépendant d'un objet. Par exemple, la famille de types associée aux listes «sur» `A` de longueur n ($n \in \mathbb{N}$) est définie par :

```
Inductive LIST [A:Set] : nat ->Set :=
NIL : (LIST A O) |
CONS : (n:nat)A->(LIST A n)->(LIST A (S n)).
```
(1.10)

En fait, `LIST` et `list` ne sont pas des types, mais des constructeurs de types. `list` est un constructeur (unaire) admettant pour argument un ensemble `A` et construisant le type des listes formées sur `A`. Il s'agit donc de la définition d'une famille de types paramétrée par les ensembles. De même, `LIST` est un constructeur (binaire) de types, qui, étant donné un entier `n` et un ensemble `A`, permet de construire le type `(LIST A n)` des listes de longueur `n` formées sur `A`. Ainsi, `(LIST A)` dénote la famille de types indicée par les entiers : `(LIST A O)`, `(LIST A (S O))`, $\cdots$

1.2.5 Extraction de programmes

En informatique, définir un programme dont on est sûr qu'il «fait bien ce pour quoi il est fait» ne relève pas d'une simple curiosité intellectuelle, certaines applications nécessitent, pour des raisons de sécurité, l'utilisation de logiciels «zéro-défaut». Une approche classique, pour répondre à ce besoin, consiste à écrire un algorithme censé résoudre un problème donné, puis à prouver (le plus formellement possible, en utilisant par exemple la «logique de Hoare») qu'il résoud bien ce problème en un temps fini : on dispose alors d'un programme «correct».

Ecriture d'un programme $\rightarrow$ *Preuve de sa correction/terminaison* $\rightarrow$ *Programme «correct»*

Une autre approche, plus récente, consiste à prouver formellement, et de manière constructive, la «satisfaisabilité» d'un problème et d'extraire automatiquement de cette preuve un programme qui résoud ce problème : on dispose alors d'un programme certifié.

Preuve constructive $\rightarrow$ *Extraction* $\rightarrow$ *Programme certifié*

L'existence d'une distinction entre les types Prop et Set permet de séparer le contenu «calculatoire» d'une preuve (c'est à dire le procédé effectif de construction des objets exhibés par la preuve) de son contenu logique (exprimant des propriétés sur les objets mis en jeu dans la preuve) ; un terme «bien typé» du calcul des constructions inductives peut être vu comme un programme contenant, de manière interne, la preuve de sa correction, c'est à dire la preuve qu'il satisfait bien une certaine spécification. Le mécanisme d'extraction d'un programme à partir d'une preuve, présenté dans [80], exploite cette distinction et permet l'obtention de programmes certifiés plus efficaces. En effet, la spécification d'un programme peut s'exprimer par une formule de la forme :

$$\forall x\, P(x) \Rightarrow \exists y\, Q(x,y)$$

traduisant le fait qu'un programme (i.e. un λ-terme) f prend un argument x satisfaisant une pré-condition P et produit un résultat y satisfaisant une post-condition $Q(x,y)$. L'interprétation du contenu «calculatoire» de la preuve de cette formule (c'est à dire le procédé de construction de y) par un programme (satisfaisant cette spécification) s'obtient en éliminant de la preuve toute la partie «non calculatoire» (c'est à dire la partie de la preuve qui établit $Q(x,y)$), afin de permettre l'obtention d'un programme efficace.

$$\begin{array}{rcccc} \text{Partie calculatoire :} & x \rightarrow & \boxed{\text{programme } f} & \rightarrow y = f(x) \\ & & \exists & \\ \text{Partie non calculatoire :} & P(x) & \Rightarrow & Q(x,y) \end{array}$$

Un tel programme est appelé une réalisation et est tel qu'il existe une preuve de :

$$\forall x\, P(x) \Rightarrow Q(x, f(x))$$

Aussi, le système Coq ne permet pas d'extraire des «informations calculatoires» à partir d'objets de type Prop[2], qualifiés d'objets non informatifs, tandis que pour tout objet A de type Set, interprété comme une spécification, tout objet a de type A pourra être interprété par un programme satisfaisant la spécification A. Par conséquent, seuls certains schémas d'élimination sur les types inductifs sont permis : il n'est pas possible, en général, de construire un objet «dans» Set à partir d'un objet «dans» Prop. C'est la raison pour laquelle, lors de la définition du prédicat inductif `even` défini en (1.6), seul le schéma d'élimination `even_ind` est engendré, alors que lors de la définition de l'ensemble `nat`, trois schémas d'induction sont engendrés.

2. bien sûr, les propositions peuvent aussi être définies «dans» Set, et c'est d'ailleurs ce que nous ferons dans le chapitre 2

C'est aussi la raison pour laquelle, lors de l'utilisation de l'opérateur `Case`, le type de l'objet sur lequel porte le `Case` ne peut être «dans» Prop si le type du résultat, indiqué entre `<` et `>`, est «dans» Set. L'utilisation d'un objet «non-calculatoire» pour construire un objet «calculatoire» poserait un problème évident lors de l'extraction d'un programme (qui ne disposerait pas de l'information contenue dans le premier). Toutefois, depuis la version 6.1 du logiciel COQ, certaines définitions inductives «dans» Prop engendrent un schéma d'élimination «dans» Set ; c'est le cas des définitions «singletons» : il s'agit des définitions inductives comportant un unique constructeur dont les arguments sont tous «dans» Prop. En effet, il existe pour ces définitions un procédé canonique d'extraction de la partie calculatoire des objets définis. Un exemple d'une telle définition est la définition inductive de la relation d'égalité, due à C. Paulin-Mohring, et équivalente à la définition de l'égalité *à la* Leibniz, par la plus petite relation binaire réflexive :

```
Inductive eq [A:Set;x:A]:A ->Prop := refl_eq : (eq A x x).
```

Cette définition engendre le schéma d'élimination suivant :

```
eq_rec : (A:Set)(x:A)(P:A->Set)(P x)->(y:A)(eq x y)->(P y) :=
[A:Set][x:A][P:A->Set][f:(P x)][y:A][e:(eq x y)]
<P>Case e of f end
```

(1.11)

L'axiome `False_rec`

Au cours du développement d'un programme, lorsque l'on se trouve dans une situation absurde, il est alors licite de produire «n'importe quoi» comme résultat, sans compromettre la correction du programme (par exemple, une fonction prenant en argument un entier n dont on a vérifié la parité avant l'appel pourra renvoyer «n'importe quoi» dans le cas où n est impair, ce cas n'étant jamais réalisé). Aussi, le système COQ contient l'axiome suivant (il s'agit de rendre le langage de programmation plus permissif que le langage de preuves) :

```
Axiom False_rec : (P:Set)False->P.
```

(1.12)

Contrairement à `False_ind`, il s'agit d'un axiome (le terme `False_rec` n'est pas construit), puisque la constante `False` du type `Prop` est définie inductivement par :

```
Inductive False : Prop := .
```

`False` est défini par un type inductif sans constructeur, et il ne s'agit donc pas d'une définition singleton ; il s'agit d'un type «non habité» (il n'existe pas d'élément de type `False` qui n'est donc pas une proposition prouvable). L'axiome `False_rec` correspond à la règle d'élimination du «faux» en logique intuitionniste :

$$(\textit{ex falso quodlibet sequitur}) : \frac{\mathsf{False}}{A}$$

Nous proposerons dans la suite un moyen de contourner, dans certains cas, l'utilisation de cet axiome.

1.2.6 Types co-inductifs

Les types co-inductifs permettent de décrire des objets infinis (c'est à dire des objets qui peuvent s'obtenir en appliquant un nombre infini de fois les constructeurs qui définissent ce type). Les questions théoriques relatives à l'introduction d'objets infinis en théorie des types sont présentées dans [20, 73], les définitions co-inductives du calcul des constructions sont détaillées dans [36]. L'exemple le plus classique de type co-inductif est le type des séquences infinies formées par des éléments de type `A` et est défini dans le système Coq par :

```
CoInductive Set Stream [A:Set]:=cons:A->(Stream A)->(Stream A).
```

Sans en présenter les fondements théoriques de manière détaillée, nous illustrons sur un exemple les concepts mis en jeu lors de la manipulation d'objets infinis. L'ensemble des listes d'entiers, noté $L_{\mathbb{N}}$, peut être défini de manière (co-)inductive à l'aide des deux règles (i.e. constructeurs) suivantes :

$$(\mathsf{nil}) : \frac{}{\mathsf{nil}\colon L_{\mathbb{N}}} \qquad (\mathsf{cons}) : \frac{n\colon \mathbb{N} \quad l\colon L_{\mathbb{N}}}{\mathsf{cons}(n,l)\colon L_{\mathbb{N}}}$$

Dans le cas d'une définition inductive, les éléments de $L_{\mathbb{N}}$ sont des listes finies et l'on dispose du schéma d'induction structurelle suivant :

$$(\mathsf{Ind}\text{-}L_{\mathbb{N}}) : \frac{\begin{array}{rcl} P & : & L_{\mathbb{N}} \to \mathsf{Prop} \\ l & : & L_{\mathbb{N}} \\ \pi_1 & : & P(\mathsf{nil}) \\ \pi_2 & : & (n_0\colon \mathbb{N})(l_0\colon L_{\mathbb{N}})P(l_0) \to P(\mathsf{cons}(n_0,l_0)) \end{array}}{\left(\begin{array}{l} \mathsf{Fixpoint}\ F\ [l' : L_{\mathbb{N}}] : P(l') \\ \mathsf{Case}\ l'\ \mathsf{of} \\ \pi_1 \\ [n_1 : \mathbb{N}][l_1 : L_{\mathbb{N}}]\pi_2(n_1, l_1, F(l_1)) \\ \mathsf{end} \end{array}\right)(l) : P(l)}$$

Dans le cas d'une définition co-inductive, les deux règles de formation des listes peuvent être appliquées un nombre de fois infini et l'on ne dispose alors plus du schéma d'induction structurelle mais d'une forme plus faible de schéma d'élimination :

$$(\mathsf{CoInd}\text{-}L_{\mathbb{N}}) : \frac{\begin{array}{rcl} P & : & L_{\mathbb{N}} \to \mathsf{Prop} \\ l & : & L_{\mathbb{N}} \\ \pi_1 & : & P(\mathsf{nil}) \\ \pi_2 & : & (n_0\colon \mathbb{N})(l_0\colon L_{\mathbb{N}})P(\mathsf{cons}(n_0,l_0)) \end{array}}{\mathsf{Case}\ l\ \mathsf{of}\ \ \pi_1\ \pi_2\ \mathsf{end} : P(l)}$$

Ce schéma exprime le fait que si l'on dispose d'une preuve π_1 de $P(\mathsf{nil})$ et d'une preuve π_2 de $\forall n_0{:}\,\mathbb{N}\forall l_0{:}\,L_{\mathbb{N}}P(\mathsf{cons}(n_0,l_0))$, alors, si la liste l est obtenue par application du constructeur nil, π_1 constitue une preuve de $P(l)$, sinon, l est obtenue par application du constructeur cons, et π_2 constitue une preuve de $P(l)$. Les deux règles de réduction associées à ce schéma sont donc :

$$\begin{array}{llll} \mathsf{Case} & \mathsf{nil} & \mathsf{of}\ \pi_1\ \pi_2\ \mathsf{end} \rightarrow & \pi_1 \\ \mathsf{Case} & \mathsf{cons}(n,l) & \mathsf{of}\ \pi_1\ \pi_2\ \mathsf{end} \rightarrow & \pi_2(n,l) \end{array}$$

Nous avons vu, au début de ce chapitre, qu'une interprétation possible pour cette définition co-inductive des listes d'entiers pouvait s'exprimer en termes d'union d'ensembles denses (relativement aux constructeurs) ou en termes de plus grand point fixe d'un opérateur (associé aux constructeurs). Cette interprétation est déclarative : aucun moyen effectif de construction n'est fourni. La contre-partie opérationnelle de cette interprétation fait intervenir la notion de terme productif définie comme suit :

- un terme de type T est en forme canonique s'il s'écrit $c(t_1, \cdots, t_n)$ où c est un constructeur du type T
- le terme t' est un composant (direct) d'un terme t de type T si t se réduit en un terme sous forme canonique $c(\cdots, t', \cdots)$ et si t' est de type T
- un terme est productif s'il se réduit en forme canonique et si tous ses composants sont productifs

De même que la terminaison d'une fonction définie récursivement sur un ensemble inductif peut être garantie de manière syntaxique (i.e. l'argument de l'appel récursif doit «structurellement» diminuer), il existe un critère syntaxique sur les termes qui, lorqu'il est vérifié, permet de garantir le caractère productif du terme défini. Les définitions récursives satisfaisant ce critère, appelées définition gardées par constructeurs, correspondent aux définitions de la forme $f(a_1, \cdots, a_n) = e$ où toutes les occurrences de f dans e sont de la forme $c(\cdots, f(t_1, \cdots, t_n), \cdots)$ où c est un constructeur et f n'apparaît pas dans les termes t_i. Par exemple, il est possible de définir récursivement la liste infinie des entiers consécutifs à partir de k. Le terme from correspondant à cette construction est défini par :

$$\mathsf{from} := \lambda n{:}\,\mathbb{N}.\mathsf{cons}(n, \mathsf{from}(S(n))) : \mathbb{N} \rightarrow L_{\mathbb{N}} \tag{1.13}$$

La définition de from est gardée puisque from est protégé par le constructeur cons. Toute définition récursive de liste ne correspond pas à une loi de construction valide : pour qu'une méthode de construction soit valide, il faut que l'on puisse déterminer tout «préfixe» de longueur finie de l'objet défini. Considérons, par exemple, la définition suivante de la liste infinie d'entiers ne contenant que des zéros :

$$\mathsf{zeros} := \mathsf{cons}(0, \mathsf{tail}(\mathsf{zeros})) : L_{\mathbb{N}}$$

où tail est une fonction qui retourne la liste de son argument privée du premier élément (nil si cette liste est vide) et est définie récursivement par :

$$\mathsf{tail} := \lambda l\colon L_{\mathbb{N}}\ .\ \mathsf{Case}\ l\ \mathsf{of}\ \mathsf{nil}\ [n_0\colon \mathbb{N}][l_0\colon L_{\mathbb{N}}]l_0\ \mathsf{end} :\ L_{\mathbb{N}} \to L_{\mathbb{N}}$$

Tandis que la méthode permettant de construire from est valide, la méthode définie pour construire zeros ne l'est pas puisqu'elle ne permet pas de connaître tout préfixe de zeros. Par exemple, on a :

$$\begin{array}{c}
\mathsf{zeros} \to\ \mathsf{cons(0,tail(zeros))} \to\ \mathsf{cons(0,tail(zeros))} \to \cdots \\
\mathsf{from}(n) \to \mathsf{cons}(n,\mathsf{from}(S(n))) \to \mathsf{cons}(n,\mathsf{cons}(S(n),\mathsf{from}(S^2(n)))) \to \cdots
\end{array}$$

$\mathsf{tail(zeros)}$ ne peut donc pas être réduit en forme canonique et zeros n'est donc pas un terme productif (la définition de zeros n'est pas gardée puisque zeros est argument d'une fonction). Un terme de type $L_{\mathbb{N}}$ est donc productif s'il se réduit en nil ou $\mathsf{cons}(a,b)$ où b est un terme productif. La condition de garde fournit donc une caractérisation syntaxique d'une classe de définitions récursives valides, appelées définitions gardées par constructeurs. Ce critère est toutefois trop restrictif puisqu'il existe des termes productifs qui peuvent être obtenus par des définitions non gardées. Considérons par exemple la définition gardée suivante de la fonction map.

$$\begin{array}{ll}
\mathsf{map} := \lambda f : \mathbb{N} \to \mathbb{N}\ . & \lambda l : L_{\mathbb{N}}\ . \\
& \mathsf{Case}\ l\ \mathsf{of} \\
& \mathsf{nil} \\
& [n_0 : \mathbb{N}][l_0 : L_{\mathbb{N}}]\ \mathsf{cons}(f(n_0),\mathsf{map}(f,l_0)) \\
& \mathsf{end.}
\end{array}$$

Cette fonction permet de définir le terme :

$$\mathsf{from\_map} := \lambda n : \mathbb{N}.\ \mathsf{cons}(n,\mathsf{map}(S,\mathsf{from\_map}(n))) : \mathbb{N} \to L_{\mathbb{N}}$$

Cette définition n'est pas gardée puisque l'appel récursif à $\mathsf{from\_map}$ est argument de la fonction map. Or, il est clair que le terme défini est productif puisque :

$$\begin{array}{ll}
& \mathsf{from_map}(n) \\
\to & \mathsf{cons}(n,\mathsf{map}(S,\mathsf{from_map}(n))) \\
\to & \mathsf{cons}(n,\mathsf{map}(S,\mathsf{cons}(n,\mathsf{map}(S,\mathsf{from_map}(n))))) \\
\to & \mathsf{cons}(n,\mathsf{cons}(S(n),\mathsf{map}(S,\mathsf{map}(S,\mathsf{cons}(n,\mathsf{map}(S,\mathsf{from_map}(n))))))) \\
\to & \cdots \\
\to & \mathsf{cons}(n,\mathsf{cons}(S(n),\mathsf{cons}(S^2(n),\ \cdots\ )))
\end{array}$$

Toutefois, il existe une définition gardée équivalente (from).

Preuves gardées

Cette condition de «garde» est aussi utile pour mener un «raisonnement infini». La notion de preuve gardée a été étudiée, dans le contexte de la théorie des types, par T. Coquand :

> *In order to establish that a proposition ϕ follows from other propositions $\phi_1, \cdots, \phi_q$, it is enough to build a proof term e for it, using not only natural deduction, case analysis, and already proven lemmas, but also using the proposition we want to prove recursively, provided such a recursive call is guarded by introduction rules.*
>
> T. Coquand [20]

Considérons, par exemple, la définition co-inductive du prédicat LN caractérisant les listes infinies d'entiers consécutifs :

$$\begin{array}{l} \texttt{CoInductive} \quad \mathsf{LN}: \mathbb{N} \to L_{\mathbb{N}} \to \mathsf{Prop} := \\ \mathsf{C}: (n : \mathbb{N})(l : L_{\mathbb{N}})(\mathsf{LN}\ (S\ n)\ l) \to (\mathsf{LN}\ n\ (\mathsf{cons}\ n\ l)). \end{array} \tag{1.14}$$

Il est possible de prouver par co-induction $\mathsf{LN}(n, \mathsf{from}(n))$ pour tout naturel n. Le terme de preuve π_{LN} utilise le schéma d'élimination $\mathsf{eq\_ind}$ (voir (1.11)) associé à la définition de l'égalité et est défini par :

$$\begin{array}{ll} \pi_{LN} := \lambda n{:}\,\mathbb{N}.\mathsf{eq\_ind}(& L_{\mathbb{N}}, \\ & \mathsf{cons}(n, \mathsf{from}(S(n))), \\ & \lambda u{:}\,L_{\mathbb{N}}.\mathsf{LN}(n, u), \\ & \mathsf{C}(n, \mathsf{from}(S(n)), \pi_{LN}(S(n))), \\ & \mathsf{from}(n), \\ & \mathsf{from\_unfold}(n)) \end{array} \tag{1.15}$$

où $\mathsf{from\_unfold}$ est la preuve de la proposition :

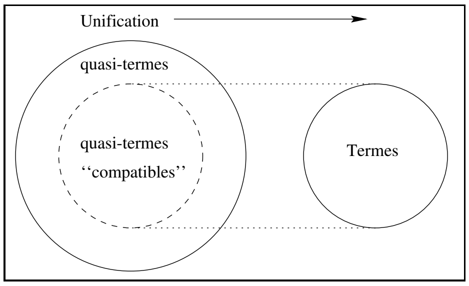
$$(n : \mathbb{N}) \quad \mathsf{from}(n) = \mathsf{cons}(n, \mathsf{from}(S(n)))$$

La définition de ce terme est bien gardée puisque l'appel récursif à π_{LN} est protégé par le constructeur C.

Chapitre 2

Transposition de propriétés formelles : l'unification

L'objectif de ce chapitre est de formaliser la preuve de la décidabilité du problème de l'unification des termes du premier ordre, mécanisme de base de la programmation logique. Toutefois, plutôt que de construire cette preuve formelle en suivant une approche classique (basée sur l'algorithme d'unification) et puisqu'une preuve formelle similaire a été construite par J. Rouyer [89] pour un ensemble plus général, l'ensemble des quasi-termes, nous allons transposer la propriété d'unification, déjà établie, de l'ensemble des quasi-termes vers l'ensemble des termes.



Il s'agit donc de mettre en relation l'ensemble sur lequel porte la preuve cherchée avec le sous-ensemble correspondant de l'ensemble des quasi-termes, et de prouver la conservation du résultat. Nous verrons donc à travers ce développement comment définir un procédé effectif permettant de mettre en relation un ensemble défini inductivement avec un sous-ensemble caractérisé par un prédicat. La principale difficulté provient du fait que, pour maintenir la cohérence du calcul, le formalisme du système Coq inclut un mini langage algorithmique (*à la* ML) avec lequel seules les fonctions totales sont représentables. Après avoir présenté deux techniques de preuve possibles pour résoudre ce problème, nous donnons les définitions formelles des objets syntaxiques mis en jeu dans l'algorithme d'unification (termes et substitutions). Les propriétés sur ces objets, constamment utilisées par la suite, sont alors établies. Enfin, l'unification au premier ordre est présentée à partir du travail de J. Rouyer [89].

2.1 Termes et quasi-termes

2.1.1 Définition des termes et algorithme d'unification

Etant donnés une signature Σ (i.e. un ensemble infini dénombrable muni d'une application $ar : \Sigma \to \mathbb{N}$) et un ensemble infini dénombrable X de symboles de variables, l'ensemble $T_\Sigma[X]$ des termes est habituellement défini de manière inductive par :

1. Un symbole de variable est un terme.
2. Si k est un symbole fonctionnel d'arité nulle, alors k est un terme.
3. Si f est un symbole fonctionnel d'arité $n > 0$, et si $t_1, \cdots, t_n$ sont des termes, alors $f(t_1, \cdots, t_n)$ est un terme.

Nous verrons par la suite comment formaliser cette définition à l'aide du langage de spécification de Coq. L'unification est une propriété définie sur les termes : deux termes t_1 et t_2 sont unifiables si il existe une substitution (i.e. une application de X dans $T_\Sigma[X]$ qui est l'identité presque partout) θ telle que $\theta t_1 = \theta t_2$. Introduit pour la première fois par J. Herbrand [46], sous la forme d'un système de simplification d'équations sur les termes, l'algorithme d'unification a été ensuite «redécouvert» par A.J. Robinson [86]. Décider si deux termes t_1 et t_2 sont unifiables revient à résoudre l'équation $t_1 = t_2$. Plus généralement, l'algorithme d'unification permet de résoudre un système d'équations E entre termes en transformant E en un système de la forme $\cup_1^n \{x_i = t_i\}$, tel que les variables x_i soient deux à deux distinctes et tel que $\{x_1, \cdots, x_n\} \cap \cup_1^n var(t_i) = \varnothing$, si E admet une solution (et dans ce cas, ce système correspond à une substitution) ou en un système noté $\{\perp\}$ si E n'admet pas de solution. Un tel système est dit résolu. L'algorithme d'unification consiste en la définition d'une relation de transition, notée $\leadsto$, et résoudre un système d'équations E entre termes revient alors à construire une suite de transitions $E \leadsto E_1 \leadsto \cdots \leadsto E_n$ telle que le système E_n soit résolu. Cet algorithme est classique et est rappelé dans le tableau 2.1. Les substitutions associées aux systèmes résolus produits par cet algorithme vérifient de «bonnes propriétés» qui seront exposées dans la suite.

2.1.2 Quasi-termes

Afin d'illustrer leur technique de synthèse de programmes [71], Z. Manna et R.J. Waldinger ont «dérivé» l'algorithme d'unification à partir de la preuve de la «satisfaisabilité» des spécifications du problème de l'unification [70]. Cette preuve a été formalisée par L.C. Paulson [81] dans le système Lcf [39].

$\{f(t_1, \cdots, t_n) = f(t'_1, \cdots, t'_n)\} \cup E$	$\overset{\text{décomposition}}{\leadsto}$	$\{t_1 = t'_1, \cdots, t_n = t'_n\} \cup E$
$\{f(t_1, \cdots, t_n) = g(t'_1, \cdots, t'_n)\} \cup E$	$\overset{\text{conflit}}{\leadsto}$	$\{\perp\}$
$\{x = t\} \cup E$	$\overset{\text{élimination}}{\leadsto}$	$\{x = t\} \cup E[x \leftarrow t]$ si $x \in var(E)$ et $x \notin var(t)$
$\{x = t\} \cup E$	$\overset{\text{occurrence}}{\leadsto}$	$\{\perp\}$ si $x \in var(t)$
$\{t = t\} \cup E$	$\overset{\text{effacement}}{\leadsto}$	E
$\{t = x\} \cup E$	$\overset{\text{inversion}}{\leadsto}$	$\{x = t\} \cup E$ si $t \notin X$

TAB. 2.1 – *Algorithme d'unification*

L'ensemble sur lequel était établie la propriété d'unification était défini inductivement à l'aide des trois constructeurs suivants :

$$\begin{array}{rcrl}
\text{VAR} & : & X & \rightarrow \text{TERM} \\
\text{CONST} & : & \Sigma & \rightarrow \text{TERM} \\
\text{COMB} & : & \text{TERM} & \rightarrow \text{TERM} \rightarrow \text{TERM}
\end{array}$$

Un développement similaire a été effectué par J. Rouyer [89], à l'aide de la version 5.8 du système COQ. Cette version ne permettant pas les définitions d'ensembles mutuellement inductifs, la transcription de la définition exacte des termes, dans le calcul des constructions inductives, par la définition d'un type inductif de type ensemble (Set), ne fut pas possible. En effet, un terme de la forme $f(t_1, \cdots, t_n)$ est construit à partir d'un symbole de fonction f d'arité n et d'une liste de termes ℓ de longueur n. Cependant, la position de «TERM» dans la définition d'un tel constructeur qui a pour type :

$$\Sigma \rightarrow \texttt{list}(\text{TERM}) \rightarrow \text{TERM}$$

en utilisant la définition polymorphe des listes, présentée en (1.9), correspond à une occurrence non positive de «TERM» [79]. Or, il n'est pas permis pour un objet A d'être défini inductivement à l'aide d'un constructeur dans lequel A n'apparaît pas positivement. D'autre part, le contrôle de l'arité des symboles fonctionnels ne peut se faire sans utiliser un type dépendant, ce qui ne permettait pas l'extraction ; nous utiliserons quand même par la suite un type dépendant, notre objectif n'étant pas d'extraire un programme des preuves. Une structure de données intermédiaire plus générale regroupant termes et listes de termes en un seul type, correspondant à l'ensemble $Q_\Sigma[X]$ des quasi-termes, avait alors été définie en supprimant de la définition des termes tout ce qui est «contrôle» pour ne conserver que les notions «constructives».

Définition 2.1 (Quasi-termes [89]) *L'ensemble $Q_\Sigma[X]$ des quasi-termes est défini inductivement, sur $\Sigma \cup X$, à l'aide des constructeurs suivants :*

$$\begin{array}{rcrl} \mathsf{V} & : & X & \to Q_\Sigma[X] \\ \mathsf{C} & : & \Sigma & \to Q_\Sigma[X] \\ \mathsf{Root} & : & \Sigma & \to Q_\Sigma[X] \to Q_\Sigma[X] \\ \mathsf{ConsArg} & : & Q_\Sigma[X] & \to Q_\Sigma[X] \to Q_\Sigma[X] \end{array}$$

V et C sont utilisés pour les variables et les constantes, Root permet d'obtenir des quasi-termes fonctionnels et $\mathsf{ConsArg}$ est utilisé pour construire des listes (non vides) de quasi-termes, correspondant aux arguments des symboles de fonction apparaissant dans les quasi-termes fonctionnels. Par exemple, le (quasi-)terme $f(x, a, g(x))$ s'obtient par :

$$\mathsf{Root}(f, \mathsf{ConsArg}(\mathsf{V}(x), \mathsf{ConsArg}(\mathsf{C}(a), \mathsf{Root}(g, \mathsf{V}(x)))))$$

Cependant, certains quasi-termes ne correspondent pas à des termes : c'est le cas des quasi-termes de la forme $\mathsf{ConsArg}(q_1, q_2)$ et des quasi-termes pour lesquels le nombre des arguments associés à un symbole de fonction f est différent de $ar(f)$. De manière similaire, certains quasi-termes ne correspondent pas à des listes de termes, comme par exemple le quasi-terme $\mathsf{ConsArg}(\mathsf{ConsArg}(x, x), x)$.

Dans les deux développements de L.C. Paulson et J. Rouyer, la propriété d'unification n'est pas obtenue directement sur l'ensemble des termes mais sur des structures de données plus générales. Toutefois, il existe une bijection entre un sous-ensemble de $Q_\Sigma[X]$ et l'ensemble des termes et il est possible de définir un prédicat caractérisant les quasi-termes «compatibles» avec les termes. Afin de transposer la propriété d'unification vers l'ensemble des termes, il est nécessaire de définir une fonction ayant pour domaine un sous-ensemble de $Q_\Sigma[X]$ caractérisé par ce prédicat. Une telle restriction, sur le domaine d'une fonction, n'est pas permise dans le calcul des constructions inductives. Une solution consiste alors à considérer simultanément un quasi-terme et la preuve de sa «compatibilité» avec les termes. Nous présentons dans le paragraphe suivant deux méthodes pour définir une telle fonction : la première utilise un axiome (`False_rec`), la deuxième, et c'est celle que nous utiliserons pour transposer la propriété d'unification, consiste à «exploiter» directement le contenu calculatoire des preuves de «compatibilité» (qui sont alors définies comme une structure de données quelconque «dans» Set).

2.2 Fonctions partielles

En informatique, les fonctions partielles apparaissent dans de nombreuses situations (division entière, queue de liste, ...). Les problèmes liés à la définition de telles fonctions dans des systèmes comme Coq, qui n'incorporent pas

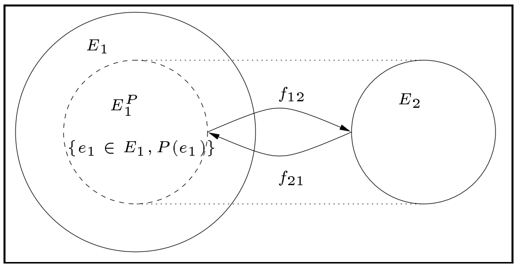


FIG. 2.1 – *Fonctions partielles*

la notion de partialité, sont discutés par C. Dubois et V. Viguié Donzeau-Gouge dans [27]. Outre l'intérêt que peut présenter la preuve de l'unification des termes du premier ordre, nous souhaitons développer une approche basée sur la réutilisabilité de preuves formelles. La construction de cette preuve conduit à définir une fonction partielle, c'est à dire une fonction dont le domaine est caractérisé par un prédicat. Plus précisément, à partir de deux ensembles, E_1 (quasi-termes) et E_2 (termes), définis inductivement, et d'un prédicat P sur E_1, tel qu'il existe une bijection entre le sous-ensemble $E_1^P = \{e \in E_1,\ P(e)\}$ de E_1 et E_2, on souhaite construire deux fonctions reliant E_1^P et E_2 (figure 2.1). Les deux caractéristiques de ce problème sont :

- Le sous-ensemble E_1^P est caractérisé par un prédicat, ce qui empêche toute définition «directe» de fonction ou de prédicat sur E_1^P (aussi, les formules $\forall e_1 \in E_1^P \varphi$ devront se lire $\forall e_1 \in E_1 (P(e_1) \Rightarrow \varphi)$).
- On ne dispose pas de la bijection, mais seulement de la preuve de son existence : cette bijection est caractérisée via une relation $\equiv$ définie sur $E_1 \times E_2$ et vérifiant :

$$\begin{array}{l} \forall e_1 \in E_1^P\,(\exists e_2 \in E_2\,(e_1 \equiv e_2 \wedge (\forall e_2' \in E_2\,(e_1 \equiv e_2' \Rightarrow e_2' = e_2)))) \\ \forall e_2 \in E_2\ \ (\exists e_1 \in E_1\,(e_1 \equiv e_2 \wedge (\forall e_1' \in E_1\,(e_1' \equiv e_2 \Rightarrow e_1' = e_1)))) \end{array}$$

On cherche à construire les deux fonctions f_{12} et f_{21} telles que :

$$\begin{array}{llrcl} f_{12} : E_1^P \to E_2 & \forall e_1 \in E_1^P & e_1 & \equiv & f_{12}(e_1) \\ f_{21} : E_2 \to E_1 & \forall e_2 \in E_2 & f_{21}(e_2) & \equiv & e_2 \end{array}$$

La fonction f_{21} se construit directement, puisqu'elle peut s'appliquer à l'ensemble E_2 tout entier, et vérifie $\forall e_2 \in E_2\, P(f_{21}(e_2))$. La définition de f_{12} est plus délicate, puisqu'elle est définie sur un ensemble caractérisé par un prédicat. Les deux solutions envisagées dans la suite consistent à considérer simultanément un élément de E_1 et la preuve qu'il satisfait bien le prédicat P [1]. Elles diffèrent selon que l'on s'autorise ou non l'utilisation de l'axiome

1. Afin de considérer l'ensemble E_1 tout entier (plutôt que E_1^P) comme domaine de la fonction f_{12}, nous pouvons ajouter à E_2 un élément «artificiel» et définir $E_2' = E_2 \cup$

`False_rec` défini en (1.12). En utilisant cet axiome, on envisage la définition de la fonction cherchée comme l'écriture d'un programme qui prend un argument e_1 satisfaisant une pré-condition P et qui peut donc produire un résultat quelconque si son argument ne satisfait pas P puisqu'un tel cas n'est pas possible (on dispose d'une preuve de $P(e_1)$) : cette approche «langage de programmation» est la première solution proposée, elle construit le résultat à partir de son argument e_1 sachant qu'il vérifie forcément le prédicat P. La deuxième solution consiste à exploiter non plus l'élément e_1, mais la preuve qu'il satisfait bien le prédicat P : cette approche «langage de preuves» est la deuxième solution proposée et n'utilise pas l'axiome `False_rec`. Ces deux solutions sont illustrées sur un exemple «jouet» présenté à la fin de ce paragraphe.

Une approche «langage de programmation»

L'approche «langage de programmation» consiste simplement à définir une fonction prenant en argument un élément e_1 de E_1 et une preuve p de $P(e_1)$. En envisageant tous les cas possibles pour e_1, c'est à dire, puisque E_1 est défini inductivement, tous les constructeurs possibles utilisés dans la formation de e_1, il suffit alors de renvoyer «n'importe quoi» dans les cas correspondant à un élément e_1 ne satisfaisant pas le prédicat P (ces cas ne pouvant être réalisés puisque l'on dispose d'une preuve de $P(e_1)$). C'est précisément ce que permet l'axiome `False_rec` : dans les «cas absurdes», il suffit de disposer d'un lemme `make_false` prouvant que pour un tel e_1, on a :

$$P(e_1) \Rightarrow \texttt{False}$$

et de renvoyer comme résultat `(False_rec` E_1 `(make_false` e_1 p`))` qui est bien un élément de E_1.

Une approche «langage de preuves»

Si l'on s'interdit l'utilisation de l'axiome `False_rec`, il faut alors prendre en compte, non plus la structure de e_1, mais la structure (c'est à dire le contenu) de la preuve de $P(e_1)$. Toutefois, puisque P est un prédicat sur E_1 ($P\colon E_1 \rightarrow \mathsf{Prop}$) et puisque E_2 est de type Set, il n'est pas possible de définir une fonction f_{12} en considérant la structure de $P(e_1)$: cela reviendrait utiliser un objet non informatif (i.e. sans contenu calculatoire) pour construire un objet informatif. Nous avons vu dans le paragraphe 1.2.5 que cela n'était pas permis. La solution proposée consiste à construire inductivement, pour tout élément e_1 de E_1^P, l'ensemble $\mathcal{P}[e_1]$ des preuves de $P(e_1)$ ($\mathcal{P}\colon E_1 \rightarrow \mathsf{Set}$). Même si $\mathcal{P}[e_1]$ et $P(e_1)$ semblent équivalents d'un point de vue logique, ils ne

$\{\mathsf{meaningless}\}$. La fonction f_{12} vérifierait alors $f_{12}(e_1) = \mathsf{meaningless}$ pour tous les éléments e_1 de E_1 ne satisfaisant pas le prédicat P. Mais cette solution ne fait que déplacer le problème.

sont pas équivalents d'un point de vue «calculatoire». Une fois cet ensemble construit, l'utilisation de types dépendants permet de définir la fonction :

$$f_{12} : \prod_{e_1 \in E_1} (\mathcal{P}[e_1] \to E_2)$$

Cette fonction est définie récursivement sur la structure d'une preuve et permet de prendre en compte le contenu algorithmique d'une telle preuve. La fonction ainsi construite doit vérifier :

$$\forall e_1 \in E_1 \quad \forall p \in \mathcal{P}[e_1] \quad e_1 \equiv f_{12}(e_1, p) \tag{2.1}$$

Ces définitions permettent de considérer des preuves comme des objets d'une collection mathématique et donc de mener des raisonnements par induction sur ces preuves ou bien de construire des fonctions récursives sur ces preuves (les lemmes sont alors explicitement utilisés en tant que fonctions).

Un exemple «jouet»

Illustrons ces deux approches sur un exemple. Les entiers naturels ont été définis en (1.5). Les entiers naturels pairs peuvent être définis inductivement dans le système Coq par :

```
Inductive nat_even:Set:=first:nat_even | next: nat_even->nat_even.
```

Le prédicat inductif `even`, défini en (1.6) permet de caractériser les éléments de `nat` qui sont pairs. Nous souhaitons définir les deux fonctions suivantes :

`nat_to_even`: $\{n$: `nat`, `even`$(n)\} \to$ `nat_even`   `even_to_nat`: `nat_even` $\to$ `nat`

La fonction `even_to_nat` se définit facilement par :

```
Fixpoint even_to_nat [n:nat_even] : nat :=
<nat>Case n of
          O
          [e:nat_even](S (S (even_to_nat e)))
     end.
```

Approche «langage de programmation»
Pour utiliser cette approche, il nous faut tout d'abord prouver les deux lemmes suivants :

```
Lemma is_even_SS : (n:nat)(even (S (S n)))->(even n).
Lemma make_false : (even (S O)) -> False.
```

Le premier permet d'obtenir une preuve de la parité de n à partir d'une preuve de la parité de $n+2$; le deuxième permet d'obtenir une preuve de la proposition `False` à partir d'une preuve de la parité de l'entier 1. Ces deux lemmes se prouvent très facilement et permettent alors de définir la fonction suivante par filtrage sur un entier :

```
Fixpoint nat_to_even [n:nat] : (even n) -> nat_even :=
<[n:nat] (even n)->nat_even>
 Cases n of
       O  => [p:(even O)]first |
    (S O) => [p:(even (S O))]
                 (False_rec nat_even (make_false p)) |
 (S (S l)) => [p:(even (S (S l)))]
                 (next (nat_to_even l (is_even_SS l p)))
 end.
```

Approche «langage de preuves»
Pour utiliser cette approche, on définit tout d'abord l'ensemble de preuves :

```
Inductive is_even_S : nat -> Set :=
O_is_even_S : (is_even_S O) |
even_next_S : (n:nat)(is_even_S n) -> (is_even_S (S (S n))).
```

La fonction `nat_to_even` peut alors être définie récursivement, sur la structure d'une preuve, par :

```
Fixpoint nat_to_even [n:nat;p:(is_even_S n)] : nat_even :=
<[_:nat]nat_even>
  Case p of
       first
       [n0:nat][p0:(is_even_S n0)] (next (nat_to_even n0 p0))
  end.
```

Remarque. Alors que dans l'approche «langage de programmation», le prédicat peut ne pas être défini de manière inductive (on peut, par exemple, utiliser le prédicat récursif `IS_EVEN` présenté dans le paragraphe 1.2.3, ce qui entraînerait toutefois des modifications dans les preuves des deux lemmes `is_even_SS` et `make_false`), l'approche «langage de preuves» nécessite une définition inductive. Signalons toutefois que ces deux approches ont en commun la manipulation d'objets «dans» Set, soit en définissant les preuves «dans» Set, soit en utilisant un axiome qui à partir d'un objet «dans» Prop produit un objet «dans» Set.

Application à la transposition de propriétés formelles

De telles fonctions peuvent être utilisées pour transposer des propriétés formelles d'un ensemble à un autre. Replaçons nous dans le cadre plus général du début de ce paragraphe. Puisque E_2 et un sous-ensemble de E_1 sont isomorphes, nous souhaitons transposer une propriété exprimée sur E_1

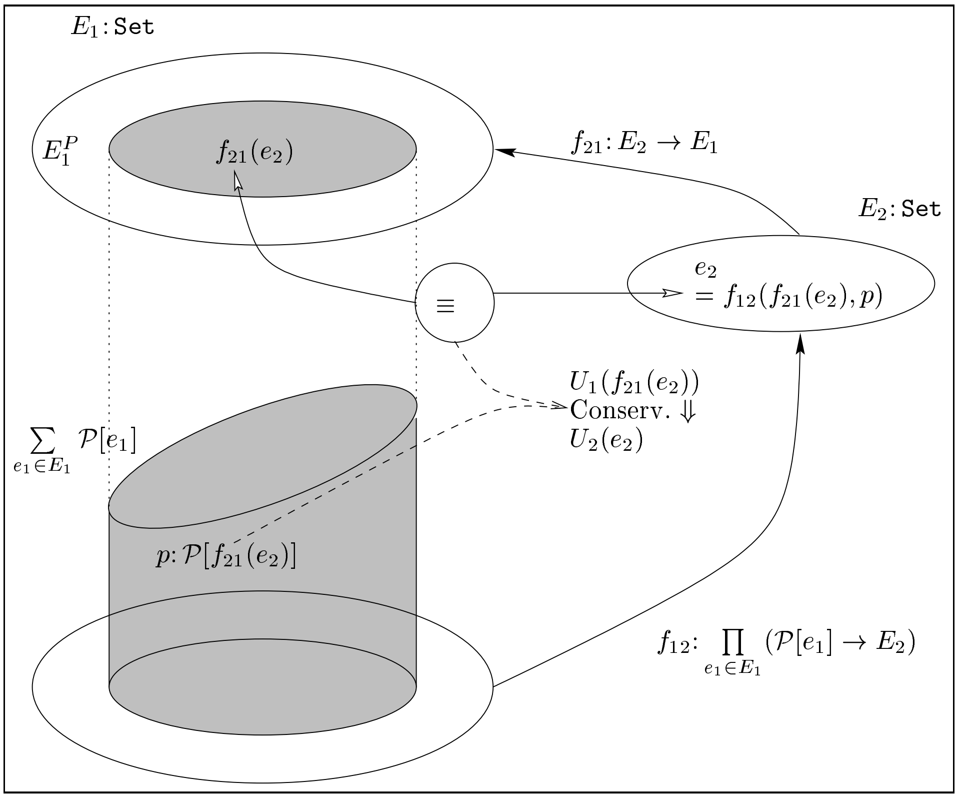


FIG. 2.2 – *Transposition de propriétés formelles*

vers E_2. Considérons deux propriétés formelles U_1 (quasi-unification) et U_2 (unification), respectivement sur E_1 et E_2, telles que :

$$\forall e_1 \in E_1 \quad \forall p \in \mathcal{P}(e_1) \quad U_1(e_1) \Rightarrow U_2(f_{12}(e_1, p))$$

Supposons que E_1 vérifie U_1 et montrons la propriété U_2 sur E_2. La preuve est obtenue en conduisant le raisonnement suivant (figure 2.2). Soit e_2 un élément de E_2, par hypothèse, on a $U_1(f_{21}(e_2))$ et puisque $f_{21}(e_2)$ admet une preuve p de $P(f_{21}(e_2))$ dans $\mathcal{P}[f_{21}(e_2)]$, on a alors $U_2(f_{12}(f_{21}(e_2), p))$. Or $f_{12}(f_{21}(e_2), p) = e_2$ ce qui prouve la propriété U_2 pour tout élément de E_2.

2.3 Liens entre termes et quasi-termes

2.3.1 Définition formelle des termes

En faisant intervenir la notion de liste, chère aux informaticiens, les assertions 2. et 3. de la définition classique des termes, donnée dans le paragraphe 2.1.1, peuvent s'exprimer de manière unique. Les types mutuellement inductifs suivants sont alors définis :

Définition 2.2 (Termes/Listes de termes)

- *L'ensemble* $T_\Sigma[X]$ *des termes sur* $\Sigma \cup X$ *est défini inductivement par :*
  - *Si* x *est un élement de* X*, alors* $\mathsf{tv}(x)$ *est un terme.*
  - *Si* f *est un élément de* Σ*, tel que* $ar(f) = n$*, et si* ℓ *est une liste de termes de longueur* n*, alors* $\mathsf{tf}(f, \ell)$ *est un terme.*
- *La famille* $\sum_{n \in \mathbb{N}} L_{n,\Sigma}[X]$ *des listes de termes, de longueur finie, est définie inductivement par :*
  - $\langle\rangle_0$ *est une liste de termes (c'est la liste vide de longueur nulle).*
  - *Si* t *est un terme et* ℓ *une liste de termes de longueur* n*, alors* $\langle t, \ell\rangle_{n+1}$ *est une liste de termes de longueur* $n + 1$*.*

Les schémas d'induction engendrés par ces définitions sont de la forme :

Induction sur $T_\Sigma[X]$ (resp. $\sum_{n \in \mathbb{N}} L_{n,\Sigma}[X]$) Soit P une propriété sur $T_\Sigma[X]$ et P_0 une propriété sur $\sum_{n\in\mathbb{N}} L_{n,\Sigma}[X]$. Si les conditions suivantes sont satisfaites :

1. $\forall v \in X \quad P(\mathsf{tv}(v))$
2. $\forall f \in \Sigma \quad \forall \ell \in L_{ar(f),\Sigma}[X] \quad (P_0(ar(f), \ell) \Rightarrow P(\mathsf{tf}(f, \ell)))$
3. $P_0(0, \langle\rangle_0)$
4. $\forall n \in \mathbb{N}\, \forall t \in T_\Sigma[X]$
   $(P(t) \Rightarrow \forall \ell \in L_{n,\Sigma}[X]\, P_0(n, \ell) \Rightarrow P_0(n + 1, \langle t, \ell\rangle_{n+1}))$

alors : $\forall t \in T_\Sigma[X] \quad P(t) \quad$ (resp. $\forall n \in \mathbb{N}\, \forall \ell \in L_{n,\Sigma}[X] \quad P_0(n, \ell)$).

Ainsi, lorsque l'on montre une propriété sur les termes, on montre une propriété analogue pour les listes de termes. Par exemple, ces schémas permettent d'établir la décidabilité de l'égalité de deux termes et de l'égalité de deux listes de termes (les deux preuves étant identiques).

Remarque. $(\sum\limits_{n\in\mathbb{N}} L_{n,\Sigma}[X]) \to s$ et $\prod\limits_{n\in\mathbb{N}} (L_{n,\Sigma}[X] \to s)$ sont isomorphes pour tout s.

Le tableau 2.2 contient la définition des termes et des quasi-termes dans le système Coq (qui, depuis la version 5.10, permet la définition d'ensembles mutuellement inductifs). L'ensemble `fun` correspond à Σ (il est indexé sur l'ensemble des naturels `nat`) et la fonction d'arité associée à cet ensemble est spécifiée par `arity:list_nat->fun->nat` et utilisée à l'aide de la liste de naturels `Lar` contenant les arités (il s'agit d'un paramètre). De même, `var` correspond à l'ensemble des symboles de variables.

2.3.2 Quasi-termes «compatibles»

Afin de pouvoir utiliser les résultats sur l'unification établis par J. Rouyer dans [89], nous allons construire la bijection existant entre l'ensemble des

```
Mutual inductive Term : Set :=
     tv : var ->Term |
     tf : (f:fun)(list_term (arity Lar f)) ->Term
with list_term : nat ->Set :=
     nil : (list_term 0) |
     cons: (n:nat)Term ->(list_term n) ->(list_term (S n)).

Inductive quasiterm : Set :=
     V : var ->quasiterm |
     C : fun ->quasiterm |
     Root : fun ->quasiterm ->quasiterm |
     ConsArg : quasiterm ->quasiterm ->quasiterm.
```

TAB. 2.2 – *Termes et Quasi-termes*

termes et un sous-ensemble des quasi-termes. Nous suivrons pour cela la méthode présentée dans le paragraphe 2.2. Les deux principales différences proviendront d'une part, du fait que les types $T_\Sigma[X]$ et $\sum_{n\in\mathbb{N}} L_{n,\Sigma}[X]$ sont mutuellement récursifs, ce qui nous amènera à construire en fait deux bijections et, d'autre part, du fait qu'il existe une liste de termes (c'est la liste vide) qui ne correspond pas à un quasi-terme.

Caractérisation de deux sous-ensembles de $Q_\Sigma[X]$

Nous commençons par identifier, à l'aide de deux prédicats, les deux sous-ensembles de $Q_\Sigma[X]$ pour lesquels il existe une bijection avec les ensembles $T_\Sigma[X]$ et $\sum_{n\in\mathbb{N}} L_{n,\Sigma}[X]$. Pour cela, deux prédicats, P^T et P^L, sont définis sur $Q_\Sigma[X]$: le premier caractérise les quasi-termes «qui ont une structure de terme» et le second caractérise les quasi-termes «qui ont une structure de liste de termes». La longueur d'un quasi-terme est tout d'abord définie récursivement par :

$$\forall q \in Q_\Sigma[X] \quad \mathrm{lg}(q) = \begin{cases} \mathrm{lg}(q_1) + \mathrm{lg}(q_2) & \text{si } q = \mathsf{ConsArg}(q_1, q_2) \\ 1 & \text{sinon} \end{cases}$$

Cette définition permet d'envisager le contrôle de l'arité des symboles fonctionnels. Nous pouvons à présent définir récursivement le prédicat P^L par :

$$\begin{array}{l} \forall x \in X \quad P^L(\mathsf{V}(x)) \\ \forall f \in \Sigma \quad (ar(f) = 0 \Rightarrow P^L(\mathsf{C}(f)) \\ \forall q \in Q_\Sigma[X]\, \forall f \in \Sigma \quad ((P^L(q) \wedge ar(f) = \mathrm{lg}(q)) \Rightarrow P^L(\mathsf{Root}(f, q))) \\ \forall q_1, q_2 \in Q_\Sigma[X] \ ((\mathrm{lg}(q_1) = 1 \wedge P^L(q_1) \wedge P^L(q_2)) \Rightarrow P^L(\mathsf{ConsArg}(q_1, q_2))) \end{array}$$

Ce prédicat est utilisé dans la définition du prédicat P^T : un quasi-terme a une structure de terme s'il a une structure de liste de termes de longueur 1.

$$\forall q \in Q_\Sigma[X] \quad (\mathrm{lg}(q) = 1 \wedge P^L(q)) \Rightarrow P^T(q)$$

Dans ce qui suit, nous utiliserons les notations suivantes :

$$Q_\Sigma^T[X] = \{q \in Q_\Sigma[X],\ P^T(q)\} \qquad Q_\Sigma^L[X] = \{q \in Q_\Sigma[X],\ P^L(q)\}$$

Puisque tout quasi-terme compatible avec les termes est compatible avec les listes de termes (de longueur 1), on a $Q_\Sigma^T[X] \subset Q_\Sigma^L[X]$.

Caractérisation de deux bijections

Nous définissons à présent deux relations d'équivalence permettant d'établir l'existence d'une bijection entre $Q_\Sigma^T[X]$ et $T_\Sigma[X]$ d'une part, et entre $Q_\Sigma^L[X]$ et $\sum_{n\in\mathbb{N}\setminus\{0\}} L_{n,\Sigma}[X]$ d'autre part. Ces relations, notées $\equiv_T$ et $\equiv_{L_n}$, sont définies respectivement sur $Q_\Sigma[X] \times T_\Sigma[X]$ et $Q_\Sigma[X] \times \sum_{n\in\mathbb{N}} L_{n,\Sigma}[X]$ ($\equiv_{L_n}$ correspond à une famille de relations). Il semble «naturel» de définir ces deux relations de façon mutuellement récursive : le tableau 2.3 contient une telle définition. Toutefois, pour des raisons «techniques», nous avons été amené à définir ces deux relations de manière indépendante, en «dérécursifiant» la relation $\equiv_{L_n}$ par un «déploiement». Les définitions obtenues sont données dans les tableaux 2.4 et 2.5 et permettent de caractériser deux bijections puisque l'on montre facilement (par induction) que :

Proposition 2.1

- *Pour tout quasi-terme q, tel que $P^L(q)$, il existe une unique liste de termes de longueur* $\lg(q)$*, telle que $q \equiv_{L_{\lg(q)}} l$.*
- *Pour toute liste de termes l de longueur $n > 0$, il existe un unique quasi-terme q tel que* $\lg(q) = n$ *et $q \equiv_{L_n} l$.*
- *Pour tout quasi-terme q, tel que $P^T(q)$, il existe un unique terme t tel que $q \equiv_T t$.*
- *Pour tout terme t, il existe un unique quasi-terme q tel que $q \equiv_T t$.*

Cependant, la preuve de ces assertions ne fournit pas de «procédé effectif» correspondant aux deux bijections. En effet, nous ne savons pas, à partir d'une hypothèse de la forme $\forall a \in A\, \exists b \in B\ P(a, b)$, construire une fonction qui pour tout a de A, calcule l'élément b de B, tel que $P(a, b)$; nous savons seulement que cet élément existe, la manière de l'obtenir étant contenue dans la preuve de l'hypothèse $\exists b \in B\, P(a, b)$. Ceci est dû au fait qu'il est impossible d'obtenir un objet «informatif» (i.e. de type Set) à partir de cette hypothèse qui est de type Prop.
Les éléments en relation via $\equiv_T$ et $\equiv_{L_n}$ correspondent bien aux éléments que l'on cherche à relier puisque l'on montre facilement par induction que :

Proposition 2.2

- $\forall q \in Q_\Sigma[X]\ \forall n \in \mathbb{N}\ \forall \ell \in L_{n,\Sigma}[X] \quad q \equiv_{L_n} \ell \Rightarrow (\lg(q) = n \wedge P^L(q))$
- $\forall q \in Q_\Sigma[X]\ \forall t \in T_\Sigma[X] \quad q \equiv_T t \Rightarrow P^T(q)$

$$q \equiv_T t$$

si $q = \mathsf{V}(x)$ alors $q \equiv_T t$ est vrai si $t = \mathsf{tv}(x)$.
si $q = \mathsf{C}(a)$ alors $q \equiv_T t$ est vrai si $t = \mathsf{tf}(a, \langle\rangle_0)$.
si $q = \mathsf{Root}(f, q_0)$ alors $q \equiv_T t$ est vrai si $t = \mathsf{tf}(f, l)$ et $q_0 \equiv_{ar(f)} l$.
si $q = \mathsf{ConsArg}(q_1, q_2)$ alors $q \equiv_T t$ est faux.

$$q \equiv_{L_n} l$$

si $q = \mathsf{V}(x)$ alors $q \equiv_{L_n} l$ est vrai si $n = 1$ et $l = \langle \mathsf{tv}(x), \langle\rangle_0 \rangle_1$.
si $q = \mathsf{C}(a)$ alors $q \equiv_{L_n} l$ est vrai si $n = 1$ et $l = \langle \mathsf{tf}(a, \langle\rangle_0), \langle\rangle_0 \rangle_1$.
si $q = \mathsf{Root}(f, q_0)$ alors $q \equiv_{L_n} l$ est vrai si $n = 1$, $l = \langle \mathsf{tf}(f, l), \langle\rangle_0 \rangle_1$
et $q_0 \equiv_{ar(f)} l$.
si $q = \mathsf{ConsArg}(q_1, q_2)$ alors $q \equiv_{L_n} l$ est vrai si $n = \lg(q_1) + \lg(q_2)$,
$l = \langle t_1, l_1 \rangle_{\lg(q_1)+\lg(q_2)-1}$, $q_1 \equiv_T t_1$ et $q_2 \equiv_{\lg(q_2)} l_1$.

TAB. 2.3 – *Définitions mutuellement récursives de* $\equiv_T$ *et* $\equiv_{L_n}$

si $l = \langle\rangle_0$, alors $q \equiv_{L_n} l$ est faux.
si $l = \langle t, l' \rangle_{m+1}$ $(n = m + 1)$, alors :
si $q = \mathsf{V}(x)$, alors $q \equiv_{L_n} l$ est vrai si $t = \mathsf{tv}(v)$, $x = v$ et $m = 0$
si $q = \mathsf{C}(f)$, alors $q \equiv_{L_n} l$ est vrai si $t = \mathsf{tf}(f', l_{f'})$, $f = f'$,
$ar(f) = 0$ et $m = 0$
si $q = \mathsf{Root}(f, q_0)$, alors $q \equiv_{L_n} l$ est vrai si $t = \mathsf{tf}(f', l_{f'})$,
$f = f'$, $m = 0$ et $q_0 \equiv_{L_{ar(f)}} l_{f'}$
si $q = \mathsf{ConsArg}(q_1, q_2)$, alors :
si $t = \mathsf{tv}(v)$, alors $q \equiv_{L_n} l$ est vrai si $q_1 = \mathsf{V}(x)$, $v = x$ et $q_2 \equiv_{L_m} l'$
si $t = \mathsf{tf}(f', l_{f'})$, alors :
si $q_1 = \mathsf{V}(x)$, alors $q \equiv_{L_n} l$ est faux.
si $q_1 = \mathsf{C}(f)$, alors $q \equiv_{L_n} l$ est vrai si $f' = f$, $ar(f) = 0$
et $q_2 \equiv_{L_m} l'$
si $q_1 = \mathsf{Root}(f, q_0)$, alors $q \equiv_{L_n} l$ est vrai si $f = f'$, $q_0 \equiv_{L_{ar(f)}} l_{f'}$
et $q_2 \equiv_{L_m} l'$
si $q_1 = \mathsf{ConsArg}(q_3, q_4)$, alors $q \equiv_{L_n} l$ est faux

TAB. 2.4 – *Définition de* $q \equiv_{L_n} l$

si $q = \mathsf{V}(x)$, alors $q \equiv_T t$ est vrai si $t = \mathsf{tv}(v)$ et $x = v$.
si $q = \mathsf{C}(f)$, alors $q \equiv_T t$ est vrai si $t = \mathsf{tf}(f', l_{f'})$, $f = f'$ et $ar(f) = 0$.
si $q = \mathsf{Root}(f, q_0)$, alors $q \equiv_T t$ est vrai si $t = \mathsf{tf}(f', l_{f'})$, $f = f'$
et $q_0 \equiv_{L_{ar(f)}} l_{f'}$.
si $q = \mathsf{ConsArg}(q_1, q_2)$, alors $q \equiv_T t$ est faux.

TAB. 2.5 – *Définition de* $q \equiv_T t$

2.3.3 Deux fonctions sur les quasi-termes «compatibles»

Nous allons à présent établir un «procédé effectif» correspondant aux deux bijections déterminées dans le paragraphe précédent. Après une présentation de l'approche «langage de programmation», nous détaillons l'approche «langage de preuves» qui sera utilisée dans toute la suite. Les sous-ensembles $Q^T_\Sigma[X]$ et $Q^L_\Sigma[X]$ étant définis par un prédicat, on ne peut pas construire directement de fonction de $Q^T_\Sigma[X]$ vers $T_\Sigma[X]$ ou de $Q^L_\Sigma[X]$ vers $\sum_{n\in\mathbb{N}} L_{n,\Sigma}[X]$, mais seulement de $Q_\Sigma[X]$ vers $T_\Sigma[X]$ ou $\sum_{n\in\mathbb{N}} L_{n,\Sigma}[X]$. C'est pourquoi, la fonction τ^t_q (resp. τ^l_q) qui pour tout quasi-terme q, vérifiant $P^T(q)$, (resp. vérifiant $P^L(q)$), calcule l'élément t de $T_\Sigma[X]$ (resp. l'élément l de $\sum_{n\in\mathbb{N}} L_{n,\Sigma}[X]$) tel que $q \equiv_T t$ (resp. $q \equiv_{L_n} l$), est une fonction qui prend pour arguments, d'une part le quasi-terme q, et d'autre part, la preuve de $P^T(q)$ (resp. la preuve de $P^L(q)$).

Approche «langage de programmation»

Les termes et les listes de termes étant définis de manière mutuellement récursives, et les listes de termes étant définies par un type dépendant, la situation est un peu plus compliquée que celle décrite sur l'exemple des entiers et des entiers pairs. Les deux fonctions à définir sont naturellement mutuellement récursives et ont pour signature :

$$\tau^t_q : \prod_{q\in Q_\Sigma[X]} (P^T[q] \to T_\Sigma[X]) \qquad \tau^l_q : \prod_{q\in Q_\Sigma[X]} (P^L[q] \to L_{\mathrm{lg}(q),\Sigma}[X])$$

Ces deux fonctions procèdent «par filtrage» sur le quasi-terme en argument et utilisent l'axiome `False_rec` à chaque fois que ce quasi-terme n'est pas «compatible», en considérant une preuve de la proposition False obtenue à l'aide du lemme suivant :

$$\texttt{Lemma make\_false:} \quad \forall q_1, q_2 \in Q_\Sigma[X]\ P^T(\mathsf{ConsArg}(q_1, q_2)) \Rightarrow \mathsf{False}.$$

Autrement dit, si l'on dispose d'une preuve de $P^T(\mathsf{ConsArg}(q_1, q_2))$, alors on dispose d'une preuve de la proposition False. D'autres lemmes plus techniques sont nécessaires afin de pouvoir obtenir une preuve de $P^T(\mathsf{ConsArg}(q_1, q_2))$ à partir de n'importe quel quasi-terme «non compatible» et afin de de pouvoir manipuler les listes de termes définies par une type dépendant. Puisqu'ils seront utilisés comme des fonctions, nous utilisons pour les exprimer le symbole fonctionnel $\to$ à la place de son équivalent logique $\Rightarrow$. Sans trop entrer dans les détails, en voici la liste. Etant donnés une liste de termes dans $L_{n_1,\Sigma}[X]$ et un entier n_2, nous souhaitons identifier les types $L_{n_1,\Sigma}[X]$ et $L_{n_2,\Sigma}[X]$ lorsque $n_1 = n_2$. C'est ce que permettent les trois lemmes suivants : le lemme ℓ_1 (resp. ℓ_2) permet, étant donnés un symbole fonctionnel $f \in \Sigma$ et une preuve de $P^T(\mathsf{C}(f))$ (resp. $P^L(\mathsf{C}(f))$), d'obtenir une liste de termes de longueur $ar(f)$ (c'est la liste vide) ; le lemme ℓ_3 permet de «transposer» une liste de $L_{n_1,\Sigma}[X]$ vers $L_{n_2,\Sigma}[X]$ lorsque $n_1 = n_2$.

$$
\begin{array}{llllll}
\ell_1\colon & (f:\Sigma) & P^T(\mathsf{C}(f)) & \to & L_{ar(f),\Sigma}[X] & \\
\ell_2\colon & (f:\Sigma) & P^L(\mathsf{C}(f)) & \to & L_{ar(f),\Sigma}[X] & \\
\ell_3\colon & (n_1,n_2:\mathbb{N}) & (n_1=n_2) & \to & L_{n_1,\Sigma}[X] & \to \quad L_{n_2,\Sigma}[X]
\end{array}
$$

Les autres lemmes permettent, étant donné un quasi-terme q satisfaisant un des deux prédicats P^T et P^L, d'établir des propriétés sur les «constituants» de ce quasi-terme.

$$
\begin{array}{llll}
\ell_4\colon (f:\Sigma)(q:Q_\Sigma[X]) & P^T(\mathsf{Root}(f,q)) & \to & P^L(q) \\
\ell_5\colon (q_1,q_2:Q_\Sigma[X]) & P^L(\mathsf{ConsArg}(q_1,q_2)) & \to & P^L(q_2) \\
\ell_6\colon (q_1,q_2:Q_\Sigma[X]) & P^L(\mathsf{ConsArg}(q_1,q_2)) & \to & P^T(q_1) \\
\ell_7\colon (f:\Sigma)(q:Q_\Sigma[X]) & P^T(\mathsf{Root}(f,q)) & \to & \lg(q) = ar(f) \\
\ell_8\colon (f:\Sigma)(q:Q_\Sigma[X]) & P^L(\mathsf{Root}(f,q)) & \to & P^T(\mathsf{Root}(f,q)) \\
\ell_9\colon (f:\Sigma)(q:Q_\Sigma[X]) & P^L(\mathsf{Root}(f,q)) & \to & P^L(q) \\
\ell_{10}\colon (q_1,q_2:Q_\Sigma[X]) & P^L(\mathsf{ConsArg}(q_1,q_2)) & \to & \lg(q_2)+1 = \\
 & & & \lg(\mathsf{ConsArg}(q_1,q_2))
\end{array}
$$

Nous pouvons à présent définir, récursivement sur la structure d'un quasi-terme, les deux fonctions τ^t_q et τ^l_q. Le tableau 2.6 contient la définition de ces deux fonctions en «pseudo-COQ».

Remarque. Dans la troisième branche de la fonction τ^l_q, correspondant au cas où q est le quasi-terme $\mathsf{Root}(f_6, q_6)$, une solution plus simple (plus naturelle ?) aurait été d'appeler récursivement τ^t_q avec $\mathsf{Root}(f_6, q_6)$ comme argument sans que cela compromette la terminaison de cette fonction. Cette solution n'est pas permise par le système COQ qui impose aux arguments d'un appel récursif de «diminuer» (c'est aussi pour cette raison que les deux définitions des relations $\equiv_T$ et $\equiv_{L_n}$ ne sont pas mutuellement récursives).

Approche «langage de preuves»

Dans toute la suite, nous utiliserons l'approche développée dans ce paragraphe et illustrée sur la figure 2.3. Sans utiliser l'axiome `False_rec`, on est amené à définir les fonctions τ^t_q et τ^l_q récursivement, non plus sur la structure d'un quasi-terme, mais sur la structure de la preuve de sa «compatibilité». On définit donc les ensembles mutuellement inductifs $\mathcal{P}^T[q]$ et $\mathcal{P}^L[q]$ correspondant aux preuves de $P^T(q)$ et de $P^L(q)$. Il ne s'agit pourtant pas d'un simple «recopiage» des prédicats P^T et P^L «dans» Set : en effet, le type des listes de termes dépendant d'un entier naturel, nous «enrichissons» la structure de P^L en considérant les preuves exprimant qu'un quasi-terme est «compatible» avec une liste de termes de longueur donnée. Aussi, la fonction τ^l_q n'aura pas exactement la même signature que dans le paragraphe précédent. Ces définitions vont permettre de considérer explicitement des preuves

`Fixpoint` $\tau^t_q[q\colon Q_\Sigma[X]]\colon P^T(q) \to T_\Sigma[X]$ `:=`
`<` $[q\colon Q_\Sigma[X]]P^T(q) \to T_\Sigma[X]$ `>`
 `Case` q `of`
 $[v_1\colon X][\_\colon P^T(\mathsf{V}(v_1))]\quad \mathsf{tv}(v_1)$

 $[f_2\colon \Sigma][p_2\colon P^T(\mathsf{C}(f_2))]\quad \mathsf{tf}(f_2, \ell_1(f_2, p_2))$

 $[f_3\colon \Sigma][q_3\colon Q_\Sigma[X]][p_3\colon P^T(\mathsf{Root}(f_3, q_3))]$
 $\mathsf{tf}(f_3, \ell_3(\lg(q_3), ar(f_3), \ell_7(f_3, q_3, p_3), \tau^l_q(q_3, \ell_4(f_3, q_3, p_3))))$

 $[q_4\colon Q_\Sigma[X]][q_5\colon Q_\Sigma[X]][p_4\colon P^T(\mathsf{ConsArg}(q_4, q_5))]$
 `False_rec`$(T_\Sigma[X],$`make_false`$(q_4, q_5, p_4))$
 `end`
`with` $\tau^l_q[q\colon Q_\Sigma[X]]\colon P^L(q) \to L_{\lg(q),\Sigma}[X]$ `:=`
`<` $[q\colon Q_\Sigma[X]]P^L(q) \to L_{\lg(q),\Sigma}[X]$ `>`
 `Case` q `of`
 $[v_2\colon X][\_\colon P^L(\mathsf{V}(v_2))]\quad \langle \mathsf{tv}(v_2), \langle\rangle_0\rangle_1$

 $[f_5\colon \Sigma][p_5\colon P^L(\mathsf{C}(f_5))]\quad \langle \mathsf{tf}(f_5, \ell_2(f_5, p_5)), \langle\rangle_0\rangle_1$

 $[f_6\colon \Sigma][q_6\colon Q_\Sigma[X]][p_6\colon P^L(\mathsf{Root}(f_6, q_6))]$
 $\langle \mathsf{tf}(f_6, \ell_3(\lg(q_6), ar(f_6), \ell_7(f_6, q_6, \ell_8(f_6, q_6, p_6)),$
 $\tau^l_q(q_6, \ell_9(f_6, q_6, p_6)))))\rangle_1$

 $[q_7\colon Q_\Sigma[X]][q_8\colon Q_\Sigma[X]][p_7\colon P^L(\mathsf{ConsArg}(q_7, q_8))]$
 $\ell_3(\lg(q_8) + 1, \lg(\mathsf{ConsArg}(q_7, q_8)), \ell_{10}(q_7, q_8, p_7),$
 $\langle \tau^t_q(q_7, \ell_6(q_7, q_8, p_7)), \tau^l_q(q_8, \ell_5(q_7, q_8, p_7))\rangle_{\lg(q_8)+1})$
 `end.`

TAB. 2.6 – *Fonctions τ^t_q et τ^l_q : approche «langage de programmation»* comme de véritables objets appartenant à une collection (Set) sur lesquels un raisonnement par induction pourra être conduit. Les ensembles $\mathcal{P}^T[q]$ et $\mathcal{P}_n^L[q]$ (en fait, on définit une famille) sont définis inductivement à l'aide de constructeurs dont les types sont :

$$\begin{array}{llll}
(x : X) & \mathcal{P}^T[\mathsf{V}(x)] & & \\
(f : \Sigma)(\ell : L_{ar(f),\Sigma}[X]) & (\ell = \langle\rangle_0) & \to & \mathcal{P}^T[\mathsf{C}(f)] \\
(f : \Sigma)(q : Q_\Sigma[X]) & \mathcal{P}_{ar(f)}^L[q] & \to & \mathcal{P}^T[\mathsf{Root}(f,q)]
\end{array}$$

$$\begin{array}{llll}
(x : X) & \mathcal{P}_1^L[\mathsf{V}(x)] & & \\
(f : \Sigma)(\ell : L_{ar(f),\Sigma}[X]) & (\ell = \langle\rangle_0) & \to & \mathcal{P}_1^L[\mathsf{C}(f)] \\
(f : \Sigma)(q : Q_\Sigma[X]) & \mathcal{P}_{ar(f)}^L[q] & \to & \mathcal{P}_1^L[\mathsf{Root}(f,q)] \\
(n : \mathbb{N})(q_1, q_2 : Q_\Sigma[X]) & \mathcal{P}_n^L[q_2] & \to & \mathcal{P}^T[q_1] \to \mathcal{P}_{n+1}^L[\mathsf{ConsArg}(q_1,q_2)]
\end{array}$$

On montre alors facilement l'adéquation entre les prédicats P^T et P^L et les éléments de $\mathcal{P}^T$ et $\mathcal{P}^L$ (i.e. un quasi-terme q satisfait $P^T(q)$ si et seulement si il existe un élément dans $\mathcal{P}^T[q]$; un quasi-terme q satisfait $P^L(q)$ si et seulement si il existe un élément dans $\mathcal{P}_{\mathrm{lg}(q)}^L[q]$). Nous pouvons à présent définir les deux fonctions τ_q^t et τ_q^l mutuellement récursives (sur la structure d'une preuve) de signature :

$$\tau_q^t : \prod_{q \in Q_\Sigma[X]} (\mathcal{P}^T[q] \to T_\Sigma[X]) \qquad \tau_q^l : \prod_{n \in \mathbb{N}, q \in Q_\Sigma[X]} (\mathcal{P}_n^L[q] \to L_{n,\Sigma}[X])$$

Ces définitions, présentées en «pseudo-Coq» dans le tableau 2.7, permettent d'associer à tout quasi-terme «compatible» le terme et/ou la liste de termes avec lequel il est en relation. En effet, on montre l'équivalent de l'assertion (2.1) du paragraphe 2.2 :

Proposition 2.3

- $\forall q \in Q_\Sigma[X] \quad \forall p \in \mathcal{P}^T[q] \qquad q \equiv_T \tau_q^t(q,p)$
- $\forall n \in \mathbb{N} \quad \forall q \in Q_\Sigma[X] \quad \forall p \in \mathcal{P}_n^L[q] \qquad q \equiv_{L_n} \tau_q^l(n,q,p)$

Cette approche «langage de preuves» exploite donc explicitement la structure des preuves, tandis que l'approche «langage de programmation», présentée plus haut, utilise implicitement les mêmes propriétés qui sont «cachées» dans les preuves des lemmes ℓ_1 à ℓ_{10}.

2.3.4 Deux fonctions dans les quasi-termes

Puisque tous les termes (et toutes les listes de termes non vides) «ont une structure de quasi-terme», les fonctions τ_t^q et τ_l^q s'obtiennent de manière classique en définissant les fonctions mutuellement récursives suivantes (tableau 2.8) :

```
Fixpoint τ_q^t[q: Q_Σ[X]; p: P^T(q)]: T_Σ[X] :=
< [q: Q_Σ[X]]T_Σ[X] >
 Case   p of
        [v: X]   tv(v)

        [f: Σ][l: L_{ar(f),Σ}[X]][p_0: (l = ⟨⟩_0]   tf(f, l)

        [f_1: Σ][q_1: Q_Σ[X]][p_1: P^L_{ar(f_1)}[q_1]]   tf(f_1, τ_q^l(ar(f_1), q_1, p_1))
 end
with τ_q^l[n: IN; q: Q_Σ[X]; p: P_n^L(q)]: L_{n,Σ}[X] :=
< [n: IN][q: Q_Σ[X]]L_{n,Σ}[X] >
 Case   p of
        [v_2: X]   ⟨tv(v_2), ⟨⟩_0⟩_1

        [f_3: Σ][l_3: L_{ar(f_3),Σ}[X]][p_3: (l_3 = ⟨⟩_0]   ⟨tf(f, l), ⟨⟩_0⟩_1

        [f_4: Σ][q_4: Q_Σ[X]][p_4: P^L_{ar(f_4)}[q_4]]   ⟨tf(f_4, τ_q^l(ar(f_4), q_4, p_4)), ⟨⟩_0⟩_1

        [n_5: IN][q_5: Q_Σ[X]][q_6: Q_Σ[X]][p_5: P^L_{n_5}[q_6]][p_6: P^T[q_5]]
              ⟨τ_q^t(q_5, p_6), τ_q^l(n_5, q_6, p_5)⟩_{n_5+1}
 end
```

TAB. 2.7 – *Fonctions τ_q^t et τ_q^l : approche «langage de preuves»*

$$\tau_t^q : T_\Sigma[X] \rightarrow Q_\Sigma[X] \qquad \tau_l^q : \prod_{n\in\mathbb{N}} (Q_\Sigma[X] \rightarrow (L_{n,\Sigma}[X] \rightarrow Q_\Sigma[X]))$$

La seule difficulté provient du fait que la liste de termes vide n'est associée à aucun quasi-terme (c'est la raison pour laquelle la fonction τ_l^q prend aussi un quasi-terme en argument). On montre alors :

Proposition 2.4

1. $\forall t \in T_\Sigma[X] \quad \tau_t^q(t) \equiv_T t$
2. $\forall n \in \mathbb{N}\, \forall \ell \in L_{n,\Sigma}[X]\, \forall t \in T_\Sigma[X] \quad \tau_l^q(n, \tau_t^q(t), \ell) \equiv_{L_{n+1}} \langle t, \ell\rangle_{n+1}$
3. $\forall q \in Q_\Sigma[X] \quad \forall p \in \mathcal{P}^T[q] \quad \tau_t^q(\tau_q^t(q,p)) = q$

2.4 Substitutions et quasi-substitutions

Nous présentons dans ce paragraphe les définitions formelles des substitutions et quasi-substitutions. Ces deux ensembles sont ensuite reliés en

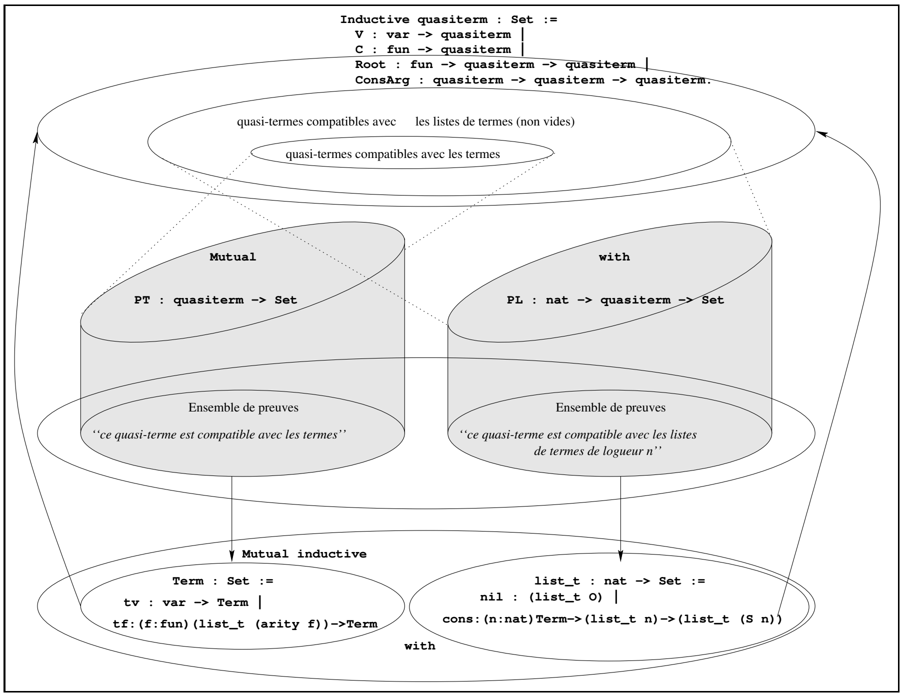


FIG. 2.3 – *Quasi-termes, termes et listes de termes*

`Fixpoint` $\tau_t^q[t : T_\Sigma[X]] : Q_\Sigma[X]$ `:=`
`<` $Q_\Sigma[X]$ `>`
`Case` t `of`
  $[v : X] \quad \mathsf{V}(v)$
  $[f : \Sigma][l : L_{ar(f),\Sigma}[X]]$
  `<` $[\_ : \mathbb{N}]Q_\Sigma[X]$ `>`
  `Case` l `of`
    $\mathsf{C}(f)$
    $[n_0 : \mathbb{N}][t_0 : T_\Sigma[X]][l_0 : L_{n_0,\Sigma}[X]] \quad \mathsf{Root}(f, \tau_l^q(n_0, \tau_t^q(t_0), l_0))$
  `end`
`end`
`with` $\tau_l^q[n : \mathbb{N}; q : Q_\Sigma[X]; l : L_{n,\Sigma}[X]] : Q_\Sigma[X]$ `:=`
`<` $[\_ : \mathbb{N}]Q_\Sigma[X]$ `>`
`Case` l `of`
  q
  $[n_0 : \mathbb{N}][t_0 : T_\Sigma[X]][l_0 : L_{n_0,\Sigma}[X]] \quad \mathsf{ConsArg}(q, \tau_l^q(n_0, \tau_t^q(t_0), l_0))$
`end.`

TAB. 2.8 – *Fonctions* τ_t^q *et* τ_l^q

suivant la même méthode que pour les ensembles $Q_\Sigma[X]$ et $T_\Sigma[X]$: les substitutions (resp. quasi-substitutions) étant définies comme les applications de X dans $T_\Sigma[X]$ (resp. $Q_\Sigma[X]$), il existe un lien de même nature entre $Q_\Sigma[X]$ et $T_\Sigma[X]$ et entre ces deux ensembles. Ensuite, les propriétés classiques sur les substitutions (étendues aux quasi-substitutions dans [89]) sont rappelées et la correspondance entre propriétés «analogues» sur des substitutions et quasi-substitutions «en relation» est établie. De même, les résultats de conservation des relations $\equiv_T$ et $\equiv_{L_n}$ lors de l'application de substitutions et quasi-substitutions «en relation» sont prouvés.

2.4.1 Définitions

L'ensemble $S[X]$ des substitutions est l'ensemble des applications de X vers $T_\Sigma[X]$. Dans [89], la définition des substitutions a été étendue aux éléments de $Q_\Sigma[X]$: l'ensemble $S_Q[X]$ des quasi-substitutions est l'ensemble des applications de X vers $Q_\Sigma[X]$. A toute quasi-substitution s_q est associée une application $\mathsf{Subst}(s_q)$, de $Q_\Sigma[X]$ vers $Q_\Sigma[X]$, définie récursivement sur la structure des quasi-termes, par :

$$
\begin{array}{l}
\forall x \in X \quad \mathsf{Subst}(s_q, \mathsf{V}(x)) = s_q(x) \\
\forall f \in \Sigma \quad \mathsf{Subst}(s_q, \mathsf{C}(f)) = \mathsf{C}(f) \\
\forall f \in \Sigma\, \forall q \in Q_\Sigma[X] \quad \mathsf{Subst}(s_q, \mathsf{Root}(f,q)) = \mathsf{Root}(f, \mathsf{Subst}(s_q, q)) \\
\forall q_1, q_2 \in Q_\Sigma[X] \\
\qquad \mathsf{Subst}(s_q, \mathsf{ConsArg}(q_1,q_2)) = \mathsf{ConsArg}(\mathsf{Subst}(s_q,q_1), \mathsf{Subst}(s_q,q_2))
\end{array}
$$

De même, on associe à toute substitution s_t deux applications $\mathsf{Subst\_t}(s_t)$ et $\mathsf{Subst\_l}(s_t)$, mutuellement récursives, l'une de $T_\Sigma[X]$ vers $T_\Sigma[X]$ et l'autre de $\sum_{n\in\mathbb{N}} L_{n,\Sigma}[X]$ vers $\sum_{n\in\mathbb{N}} L_{n,\Sigma}[X]$, définies récursivement par :

$$
\begin{array}{l}
\forall x \in X \quad \mathsf{Subst\_t}(s_t, \mathsf{tv}(x)) = s_t(x) \\
\forall f \in \Sigma\, \forall l \in L_{ar(f),\Sigma}[X] \quad \mathsf{Subst\_t}(s_t, \mathsf{tf}(f,l)) = \mathsf{tf}(f, \mathsf{Subst\_l}(s_t, l))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Subst\_l}(s_t, \langle\rangle_0) = \langle\rangle_0 \\
\forall t \in T_\Sigma[X]\, \forall n \in \mathbb{N}\, \forall l \in L_{n,\Sigma}[X] \\
\qquad \mathsf{Subst\_l}(s_t, \langle t, l\rangle_{n+1}) = \langle \mathsf{Subst\_t}(s_t,t), \mathsf{Subst\_l}(s_t,l)\rangle_{n+1}
\end{array}
$$

Dans ce qui suit, on écrira seulement s_q (resp. s_t) pour Subst (resp. $\mathsf{Subst\_t}$ ou $\mathsf{Subst\_l}$) lorsqu'aucune confusion ne sera possible. De plus, on notera désormais $s_1 s_2$ la substitution $\lambda x. s_1(s_2(x))$.

2.4.2 Liens entre substitutions et quasi-substitutions

De même que le sous-ensemble $Q^T_\Sigma[X]$ de $Q_\Sigma[X]$ est défini à l'aide du prédicat P^T, on caractérise un sous-ensemble $S^T_Q[X]$ de $S_Q[X]$, correspondant aux quasi-substitutions «compatibles» avec les termes, à l'aide d'un prédicat P^S défini par :

$$\forall s_q \in S_Q[X] \quad ((\forall x \in X\, P^T(s_q(x))) \Rightarrow P^S(s_q))$$

Autrement dit, $S_Q^T[X]$ est l'ensemble des applications de X vers $Q_\Sigma^T[X]$. On montre facilement, par induction, que si s_q vérifie $P^S(s_q)$ et si q est un élément de $Q_\Sigma^T[X]$, alors $s_q(q)$ est aussi un élément de $Q_\Sigma^T[X]$. Afin d'établir une correspondance entre $S_Q^T[X]$ et $S[X]$, on définit une relation $\equiv_S$ sur $S_Q[X] \times S[X]$ comme suit :

$$\forall s_q \in S_Q[X]\, \forall s_t \in S[X] \quad ((\forall x \in X\, s_q(x) \equiv_T s_t(x)) \Rightarrow s_q \equiv_S s_t)$$

Cette relation satisfait bien la proposition :

Proposition 2.5 $\forall s_q \in S_Q[X]\, \forall s_t \in S[X]\, (s_q \equiv_S s_t \Rightarrow P^S(s_q))$

PREUVE. Par hypothèse, pour toute variable x, $s_q(x) \equiv_T s_t(x)$. La proposition 2.2 permet alors d'établir $P^T(s_q(x))$ pour toute variable x, ce qui permet de conclure à $P^S(s_q)$. ◀

Nous allons à présent, de la même manière que nous avons établi la bijection reliant $T_\Sigma[X]$ et $Q_\Sigma^T[X]$, construire un procédé effectif, qui :

- pour toute quasi-substitution s_q vérifiant P^S associe une substitution s_t telle que $s_q \equiv_S s_t$
- pour toute substitution s_t associe une quasi-substitution s_q (vérifiant $P^S(s_q)$) telle que $s_q \equiv_S s_t$

Ici encore, nous allons définir inductivement une ensemble (Set) de preuves sur lesquelles un raisonnement par induction pourra être conduit et des fonctions construites. Pour toute quasi-substitution s, un élément de $\mathcal{P}^S[s]$ est obtenu à l'aide d'un constructeur de type :

$$((x : X)(\mathcal{P}^T[s_q(x)])) \to \mathcal{P}^S[s_q]$$

Bien entendu, une quasi-substitution s est dans $S_Q^T[X]$ si et seulement si elle admet une preuve de $P^S(s)$ dans $\mathcal{P}^S(s)$. Nous pouvons alors construire la fonction :

$$\tau_s^t : \prod_{s_q \in S_Q[X]} (\mathcal{P}^S[s_q] \to S[X])$$

$\tau_s^t(s_q, p)$ est défini récursivement sur la structure de p par : si p est un élément de $\mathcal{P}^S(s_q)$ obtenu à partir d'une preuve p_0 dans $\prod_{x \in X} \mathcal{P}^T(s_q(x))$, alors :

$$\tau_s^t(s_q, p) = \lambda x.\tau_q^t(s_q(x), p_0(x))$$

Réciproquement, mais de manière plus classique, on construit la fonction suivante :

$$\tau_s^q : S[X] \to S_Q[X] \qquad \tau_s^q(s_t) = \lambda x.\tau_t^q(s_t(x))$$

Ces deux fonctions satisfont bien la proposition :

Proposition 2.6

1. $\forall s_q \in S_Q[X]\, \forall p \in \mathcal{P}^S[s_q] \quad s_q \equiv_S \tau_s^t(s_q, p)$
2. $\forall s_t \in S[X] \quad \tau_s^q(s_t) \equiv_S s_t$

2.4.3 Conservation des propriétés de «compatibilité»

Nous étudions dans ce paragraphe la conservation du lien entre quasi-termes et termes lors de l'application de quasi-substitutions et de substitutions. Le lemme qui suit, dont la démonstration découle (presque) directement des définitions, traduit le fait que :

- pour tout quasi-terme q admettant une preuve de $P^T(q)$ dans $\mathcal{P}^T[q]$ et pour toute quasi-substitution s admettant une preuve de $P^S(s)$ dans $\mathcal{P}^S[s]$, $s(q)$ admet une preuve de $P^T(s(q))$ dans $\mathcal{P}^T[s(q)]$
- pour tout naturel n et tout quasi-terme q, admettant une preuve de $P^L(q)$ dans $\mathcal{P}_n^L[q]$, et pour toute quasi-substitution s admettant une preuve de $P^S(s)$ dans $\mathcal{P}^S[s]$, $s(q)$ admet une preuve de $P^L(s(q))$ dans $\mathcal{P}_n^L[s(q)]$.

Puisque $\mathcal{P}^T[q]$, $\mathcal{P}_n^L[q]$ et $\mathcal{P}^S[s]$ sont des ensembles, les deux assertions de ce lemme constituent explicitement deux fonctions, notées τ_p^t et τ_p^l, ayant pour arguments des preuves et construisant une preuve. Ici encore, ce lemme est utilisé par la suite comme une fonction et nous utilisons le symbole fonctionnel $\to$ à la place de son équivalent logique $\Rightarrow$.

Lemme 2.1 (Définition des fonctions τ_p^t et τ_p^l)

1. $\tau_p^t : (q : Q_\Sigma[X])\,(s : S_Q[X]) \quad \mathcal{P}^T[q] \to \mathcal{P}^S[s] \to \mathcal{P}^T[s(q)]$
2. $\tau_p^l : (n : \mathbb{N})\,(q : Q_\Sigma[X])\,(s : S_Q[X]) \quad \mathcal{P}_n^L[q] \to \mathcal{P}^S[s] \to \mathcal{P}_n^L[s(q)]$

Ce lemme, utilisé directement en tant que fonction, permet d'établir l'égalité suivante :

Lemme 2.2 $\forall q \in Q_\Sigma[X]\, \forall p_q \in \mathcal{P}^T[q]\, \forall s \in S_Q[X]\, \forall p_s \in \mathcal{P}^S[s]$

$$(\tau_s^t(s, p_s))(\tau_q^t(q, p_q)) = \tau_q^t(s(q), \tau_p^t(q, s, p_q, p_s))$$

PREUVE. D'après le lemme 2.1, $\tau_p^t(q, s, p_q, p_s)$ constitue bien une preuve de $P^T(s(q))$ et la démonstration se fait par induction sur la preuve p_q. ◀

D'autre part, on montre que les relations $\equiv_T$ et $\equiv_L$ sont conservées lors de l'application de substitutions et de quasi-substitutions «équivalentes» :

Lemme 2.3 *Soit s_q une quasi-substitution et s_t une substitution telles que $s_q \equiv_S s_t$.*

1. $\forall n \in \mathbb{N}\, \forall l \in L_{n,\Sigma}[X]\, \forall q \in Q_\Sigma[X]\, (q \equiv_{L_n} l \Rightarrow s_q(q) \equiv_{L_n} s_t(l))$
2. $\forall t \in T_\Sigma[X]\, \forall q \in Q_\Sigma[X]\, (q \equiv_T t \Rightarrow s_q(q) \equiv_T s_t(t))$

PREUVE. La preuve de 1. se fait par induction sur l, celle de 2. par induction sur t. Le schéma d'induction utilisé est celui décrit page 40 dans lequel la propriété P (resp. P_0) est exprimée par 2. (resp. 1.). Cette preuve s'obtient donc en quatre étapes :
(1). Si $q \equiv_T \mathsf{tv}(v)$, c'est que $q = \mathsf{V}(v)$ et alors $s_q \equiv_S s_t$ entraîne :

$$s_q(q) = s_q(v) \equiv_T s_t(v) = s_t(\mathsf{tv}(v))$$

(2). L'hypothèse d'induction exprime que si $q_0 \equiv_{L_{ar(f)}} l$, alors $s_q(q_0) \equiv_{L_{ar(f)}} s_t(l)$. On cherche à montrer que si q est un quasi-terme tel que $q \equiv_T \mathsf{tf}(f, l)$, alors $s_q(q) \equiv_T s_t(\mathsf{tf}(f, l))$. Deux cas se présentent. Si $q = \mathsf{C}(f)$, alors c'est que $l = \langle\rangle_0$ et il est alors clair que :

$$s_q(\mathsf{C}(f)) = \mathsf{C}(f) \equiv_T \mathsf{tf}(f, \langle\rangle_0) = s_t(\mathsf{tf}(f, \langle\rangle_0))$$

Si $q = \mathsf{Root}(f, q_0)$ avec $q_0 \equiv_{L_{ar(f)}} l$, alors par hypothèse d'induction, on a $s_q(q_0) \equiv_{L_{ar(f)}} s_t(l)$ et il vient finalement :

$$s_q(q) = \mathsf{Root}(f, s_q(q_0)) \equiv_T \mathsf{tf}(f, s_t(l)) = s_t(\mathsf{tf}(f, l))$$

(3). L'hypothèse $q \equiv_{L_0} \langle\rangle_0$ étant toujours fausse, la propriété est vérifiée.
(4). Soit un terme t et une liste de termes l de longueur n. Par hypothèse d'induction, on a :

- si q_1 est un quasi-terme tel que $q_1 \equiv_T t$, alors $s_q(q_1) \equiv_T s_t(t)$
- si q_2 est un quasi-terme tel que $q_2 \equiv_{L_n} l$, alors $s_q(q_2) \equiv_{L_n} s_t(l)$

Si q est un quasi-terme tel que $q \equiv_{L_{n+1}} \langle t, l\rangle_{n+1}$, alors deux cas se présentent. Si $l = \langle\rangle_0$, alors $s_q(q) \equiv_{L_1} \langle s_t(t), \langle\rangle_0\rangle_1$ si $s_q(q) \equiv_T s_t(t)$. Or, par hypothèse, $q \equiv_{L_1} \langle t, \langle\rangle_0\rangle_1$, c'est à dire $q \equiv_T t$. L'hypothèse d'induction permet alors de conclure à $s_q(q) \equiv_T s_t(t)$. Si l n'est pas la liste vide, alors $q = \mathsf{ConsArg}(q_1, q_2)$ et les hypothèses permettent de conclure à :

$$s_q(q) = \mathsf{ConsArg}(s_q(q_1), s_q(q_2)) \equiv_{L_{n+1}} \langle s_t(t), s_t(l)\rangle_{n+1} = s_t(\langle t, l\rangle_{n+1})$$

◀

2.4.4 Principales propriétés

Etant donnés deux termes, décider s'ils sont unifiables consiste à chercher une substitution satisfaisant certaines propriétés. Nous rappelons donc à présent les propriétés «classiques» des substitutions, étendues aux quasi-substitutions, et montrons que ces propriétés sont, pour la plupart, conservées par les applications τ_s^t et τ_s^q.

Propriétés

Domaine, image Le domaine et l'image d'une substitution (resp. d'une quasi-substitution) s sont définis par les prédicats :

$$\begin{array}{rcl} dom(s) & = & \{x \in X,\ s(x) \neq \mathsf{tv}(x)\} \\ range(s) & = & \{x \in X,\ \exists y \in dom(s)\ x \in var(s(y))\} = \bigcup\limits_{y \in dom(s)} var(s(y)) \end{array}$$

L'appartenance d'une variable à un terme (resp. un quasi-terme) est définie par un prédicat récursif sur la structure du terme (resp. du quasi-terme) considéré. De plus, notons que cette relation est décidable (puisque l'égalité sur les variables l'est). Aussi, l'appartenance d'une variable au domaine d'une substitution (resp. d'une quasi-substitution) est décidable.

Support, couverture Etant donnée une substitution (resp. quasi-substitution) s, le support et la couverture de s sont définis par les deux prédicats suivants. Soit un terme ou une liste de termes (resp. un quasi-terme) t, on dit que :

- s est supportée par t, si aucune variable n'appartenant pas à t n'est affectée par s (i.e. $dom(s) \subseteq var(t)$).
- s est couverte par t, si toute variable de l'image de s est dans t.

Idempotence Une substitution (resp. une quasi-substitution) s est idempotente si pour toute variable x, $s(s(x)) = s(x)$, ou de manière équivalente, comme le montre le lemme suivant, si $dom(s) \cap range(s) = \varnothing$.

Lemme 2.4 *Une substitution s est idempotente si et seulement si elle vérifie $dom(s) \cap range(s) = \varnothing$.*

Preuve. ($\Leftarrow$). Soit $x \in X$, montrons que pour toute variable $v \in s(x)$, on a $v \notin dom(s)$. Deux cas se présentent. Si $x \in dom(s)$, alors, par définition, $v \in range(s)$ et, par hypothèse, $v \notin dom(s)$. Sinon, $s(s(x)) = s(x) = \mathsf{tv}(x)$, et on peut conclure.
($\Rightarrow$). Soit $x \in range(s)$. Par définition, il existe une variable $v \in dom(s)$ telle que $x \in s(v)$. Si $x \in dom(s)$, alors on a $s(s(x)) \neq s(x)$ ce qui est contradictoire. ◀

$S[X]$	`Definition subst := var -> Term.`
domaine	`Definition dom_t:subst->var->Prop:=` `[s:subst][x:var]~(tv x)=(s x).`

image	`Inductive range_t[s:subst;x:var]:Prop:=`
	`range_t_init:(y:var)(dom_t s y)->`
	`(IS_IN_T x (s y))->(range_t s x).`

support	`Definition over_t:subst->Term->Prop:=`
	`[f:subst][t:Term]`
	`(x:var)(~(IS_IN_T x t)) -> (tv x)=(f x).`

couverture	`Definition under_t:subst->Term->Prop:=`
	`[s:subst][t:Term]`
	`(x,y:var)(dom_t s y)->(IS_IN_T x (s y))`
	`->(IS_IN_T x t).`

idempotence	`Definition idempotent_t:subst->Prop:=` `[s:subst](x:var)(s x)=(Subst_t s (s x)).`

préordre ($\leqslant$)	`Inductive less_subst_t [f,g:subst] : Prop :=`
	`less_subst_t_init:(h:subst)`
	`((x:var)((Subst_t h (f x))=(g x)))`
	`->(less_subst_t f g).`

TAB. 2.9 – *Propriétés sur les substitutions*

Préordre On définit un préordre sur les substitutions (resp. sur les quasi-substitutions) par :

$$\forall s_1, s_2 \in S[X] \quad (s_1 \leqslant s_2) \Leftrightarrow (\exists s \in S[X]\, \forall x \in X\; s(s_1(x)) = s_2(x))$$

Ce préordre induit une relation d'équivalence correspondant à l'équivalence au renommage près des variables. Deux substitutions s_1 et s_2 sont dites $\leqslant$-équivalentes si $s_1 \leqslant s_2$ et $s_2 \leqslant s_1$.

Lemme 2.5 *Si deux substitutions σ et θ sont $\leqslant$-équivalentes, alors il existe une substitution r telle que pour tout terme t, $\sigma(t) = r(\theta(t))$ et telle que pour toute variable x de $\theta(t)$, il existe une variable v telle que $r(x) = \mathsf{tv}(v)$ (r peut être vue comme une substitution de renommage).*

PREUVE. Soit t un terme. Par hypothèse ($\sigma \leqslant \theta$), il existe une substitution μ_1 telle que $\mu_1\sigma(t) = \theta(t)$. De même ($\theta \leqslant \sigma$), il existe une substitution μ_2 telle que $\mu_2\theta(t) = \sigma(t)$. r existe donc : c'est μ_2. De plus, on a $\mu_1\mu_2\theta(t) = \theta(t)$. Montrons que $\forall x \in X\; (x \in \theta(t) \Rightarrow \exists v \in X\; \mu_2(x) = \mathsf{tv}(v))$. Si x appartient au domaine de μ_2, alors $\mu_1\mu_2\theta(x) = \theta(x)$ et $\mu_2(x) \neq \mathsf{tv}(x)$. Il existe donc une variable v appartenant au domaine de μ_1, telle que $\mu_2(x) = \mathsf{tv}(v)$. Si x n'appartient pas au domaine de μ_2, alors v existe, c'est la variable x. ◀

Le tableau 2.9 contient ces définitions dans le système COQ.

Conservation

Afin d'alléger les notations, nous écrirons dans toute la suite des égalités de la forme $s_1 = s_2$. Il s'agit d'un abus de notation : en effet, cette relation doit se lire $\forall x \in X \;\; s_1(x) = s_2(x)$ [2].

Lemme 2.6 *Soit s_t une substitution et s_q une quasi-substitution idempotente admettant une preuve p dans $\mathcal{P}^S[s_q]$. Si $s_q \leqslant \tau_s^q(s_t)$, alors $\tau_s^t(s_q, p) \leqslant s_t$.*

PREUVE. Puisque $s_q \leqslant \tau_s^q(s_t)$, il existe une quasi-substitution h_q telle que $h_q s_q = \tau_s^q(s_t)$. $h_q s_q$ est donc un élément de $S_Q^T[X]$ et admet alors une preuve p_0 dans $\mathcal{P}^S[h_q s_q]$. On cherche à montrer qu'il existe une substitution h_t telle que $h_t \tau_s^t(s_q, p) = s_t$. Alors que s_q et $h_q s_q$ sont des quasi-substitutions «compatibles» avec les termes, on ne peut établir $P^S(h_q)$ [3]. Toutefois, puisque s_q est idempotente, on peut choisir pour h_t la substitution $\tau_s^t(h_q s_q, p_0)$. Montrons alors que $\forall x \in X \; \tau_s^t(h_q s_q, p_0)\tau_s^t(s_q, p)(x) = s_t(x)$. Par définition, on a $\tau_s^t(h_q s_q, p_0)\tau_s^t(s_q, p)(x) = \tau_s^t(h_q s_q, p_0)\tau_s^t(s_q, p)(\mathsf{tv}(x))$. Or, $\mathsf{V}(x)$ admet une preuve p_1 dans $\mathcal{P}^T[\mathsf{V}(x)]$ telle que $\mathsf{tv}(x) = \tau_q^t(\mathsf{V}(x), p_1)$. En appliquant deux fois le lemme 2.2, il vient :

$$\begin{array}{ll} & \tau_s^t(h_q s_q, p_0)\tau_s^t(s_q, p)(\tau_q^t(\mathsf{V}(x), p_1)) \\ = & \tau_s^t(h_q s_q, p_0)\tau_q^t(s_q(\mathsf{V}(x)), \tau_p^t(\mathsf{V}(x), s_q, p_1, p)) \\ = & \tau_q^t(h_q s_q s_q(\mathsf{V}(x)), \tau_p^t(s_q(\mathsf{V}(x)), h_q s_q, \tau_p^t(\mathsf{V}(x), s_q, p_1, p), p_0)) \end{array}$$

s_q étant idempotente, on a $h_q s_q s_q(\mathsf{V}(x)) = h_q s_q(\mathsf{V}(x))$ et donc :

$$\begin{array}{ll} & \tau_q^t(h_q s_q s_q(\mathsf{V}(x)), \tau_p^t(s_q(\mathsf{V}(x)), h_q s_q, \tau_p^t(\mathsf{V}(x), s_q, p_1, p), p_0)) \\ = & \tau_q^t(h_q s_q(\mathsf{V}(x)), \tau_p^t(\mathsf{V}(x), h_q s_q, p_1, p_0)) \end{array}$$

En appliquant à nouveau le lemme 2.2, il vient :

$$\begin{array}{ll} \tau_q^t(h_q s_q(\mathsf{V}(x)), \tau_p^t(\mathsf{V}(x), h_q s_q, p_1, p_0)) & = (\tau_s^t(h_q s_q, p_0))(\tau_q^t(\mathsf{V}(x), p_1)) \\ & = s_t(x) \quad \text{(par hypothèse)} \end{array}$$

ce qui permet de conclure. ◀

Lemme 2.7 *Si s_q est une quasi-substitution et s_t une substitution satisfaisant $s_q \equiv_S s_t$, alors x est une variable du domaine de s_q si et seulement si x est une variable du domaine de s_t.*

---

2. Le système COQ n'utilisant pas la règle de η-réduction $\lambda x.(s\ x) \to_\eta s$ où x n'est pas une variable libre dans s, il n'est pas possible de déduire de $\forall x \in X \;\; s_1(x) = s_2(x)$ l'égalité $s_1 = s_2$. Les fonctions extensionnellement égales ne peuvent être identifiées sans utiliser un axiome.

3. En effet, pour $s_q = \{x/\mathsf{V}(y)\}$ et $h_q = \{x/\mathsf{ConsArg}(y, z)\}$, on a bien $P^S(s_q)$ et $P^S(h_q s_q)$, tandis que h_q ne vérifie pas le prédicat P^S puisque $h_q(x)$ ne vérifie pas $P^T(h_q(x))$.

PREUVE. Soit x une variable du domaine de s_q (resp. s_t), on a $s_q(x) \neq \mathsf{V}(x)$ (resp. $s_t(x) \neq \mathsf{tv}(x)$). Or par hypothèse, $s_q(x) \equiv_T s_t(x)$ et donc $s_t(x) \neq \mathsf{tv}(x)$ (resp. $s_q(x) \neq \mathsf{V}(x)$). ◀

Lemme 2.8 *Soit s_q une quasi-substitution idempotente et s_t une substitution. Si $s_q \equiv_S s_t$, alors s_t est idempotente.*

PREUVE. Puisque $s_q \equiv_S s_t$, pour tout $x \in X$, $s_q(x) \equiv_T s_t(x)$ ce qui permet de conclure à $s_q(s_q(x)) \equiv_T s_t(s_t(x))$. Or, par hypothèse, $s_q(s_q(x)) = s_q(s)$, ce qui prouve l'idempotence de s_t. ◀

Lemme 2.9 *Soit s_q une quasi-substitution et s_t une substitution telles que $s_q \equiv_S s_t$. Soit q un quasi-terme et l une liste de termes de longueur n tels que $q \equiv_{L_n} l$. Si s_q est supportée et couverte par q, alors s_t est supportée et couverte par l.*

PREUVE. (support). Soit x une variable n'appartenant pas à q. Puisque $q \equiv_{L_n} l$, x n'apparaît pas dans l. Puisque, par hypothèse, $s_q(x) = \mathsf{V}(x) \equiv_T s_t(x)$, c'est que $s_t(x) = \mathsf{tv}(x)$ et on a bien $\forall x \in X\ (x \notin l \Rightarrow s_t(x) = \mathsf{tv}(x))$.
(couverture). Soit y une variable du domaine de s_t et x une variable de $s_t(y)$. Puisque $s_q \equiv_S s_t$, y est aussi dans le domaine de s_q et $s_q(y) \equiv_T s_t(y)$. x est donc aussi dans $s_q(y)$. Puisque s_q est couverte par q, x appartient à q, et l'hypothèse $q \equiv_{L_n} l$ permet de conclure à l'appartenance de x à l. ◀

Lemme 2.10 *Soit l_1 et l_2 deux listes de termes de longueur n_1 et n_2, q_1 et q_2 deux quasi-termes tels que $q_1 \equiv_{L_{n_1}} l_1$ et $q_2 \equiv_{L_{n_2}} l_2$, s_t une substitution et s_q une quasi-substitution telles que $s_q \equiv_S s_t$. Si s_q est supportée et couverte par $\mathsf{ConsArg}(q_1, q_2)$, alors s_t est supportée et couverte par la liste de termes obtenue par concaténation des listes l_1 et l_2, notée $l_1 \bowtie_l l_2$.*

PREUVE. (support). Soit x une variable n'appartenant pas à $\mathsf{ConsArg}(q_1, q_2)$. s_q étant supportée par $\mathsf{ConsArg}(q_1, q_2)$, x n'appartient pas au domaine de la substitution s_q et d'après le lemme 2.7, x n'appartient pas au domaine de s_t. Or, si x n'est pas dans $ConsArg(q_1, q_2)$, c'est que x n'est pas dans $l_1 \bowtie_l l_2$ et donc s_t est supportée par $l_1 \bowtie_l l_2$.
(couverture). Soit y une variable du domaine de s_q et x une variable de $s_q(y)$. D'après le lemme 2.7, y est aussi une variable du domaine de s_t et puisque $s_q \equiv_S s_t$, le lemme 2.3 permet d'établir que $s_q(y) \equiv_T s_t(y)$ et donc x appartient à $s_t(y)$. s_q étant couverte par $\mathsf{ConsArg}(q_1, q_2)$, x est donc dans $\mathsf{ConsArg}(q_1, q_2)$ et donc dans $l_1 \bowtie_l l_2$. s_t est donc bien couverte par la liste $l_1 \bowtie_l l_2$. ◀

Unificateur
`Definition unif_t:subst->Term->Term->Prop:=`
`          [f:subst][t,u:Term](Subst_t f t)=(Subst_t f u).`

Unificateur minimal
`Definition min_unif_t:subst->Term->Term->Prop:=`
`          [f:subst][t,u:Term]`
`              (g:subst)(unif_t g t u)->(less_subst_t f g).`

Problème de l'unification
`Inductive Unification_t[t1,t2:Term]:Set:=`
`Unif_succeed_t : (f:subst)`
`    (unif_t f t1 t2)`
`  ->(idempotent_t f)`
`  ->(over_lt  f (S (S O)) (cons (S O) t1 (cons O t2 nil)))`
`  ->(under_lt f (S (S O)) (cons (S O) t1 (cons O t2 nil)))`
`  ->(min_unif_t f t1 t2)`
`  ->(Unification_t t1 t2)          |`
`Unif_fail_t:((f:subst)~(Subst_t f t1)=(Subst_t f t2))`
`  ->(Unification_t t1 t2).`

Théorème d'unification `Lemma UNIFICATION_T : (t1,t2:Term)(Unification_t t1 t2).`

TAB. 2.10 – *Spécification du problème de l'unification*

2.5 Unification

Nous sommes à présent en mesure de formaliser le théorème d'unification. Pour cela, rappelons qu'étant donnés deux termes (resp. deux quasi-termes) t_1 et t_2, la substitution (resp. la quasi-substitution) s unifie t_1 et t_2 si $s(t_1) = s(t_2)$; s est un unificateur minimal de t_1 et t_2, si tout unificateur s' de t_1 et t_2 vérifie $s \leqslant s'$. Un unificateur est dit principal (ou le plus général), s'il est minimal, idempotent, supporté et couvert par la liste de termes de longueur 2 constituée des termes t_1 et t_2 (resp. par le quasi-terme $\mathsf{ConsArg}(t_1, t_2)$). Ces définitions sont étendues aux listes de termes. Nous cherchons à montrer que le problème de l'unification est décidable : étant donnés deux termes, soit ils ne sont pas unifiables, soit ils le sont et il existe alors pour ces deux termes un unificateur principal. Les spécifications de ces définitions et du problème d'unification dans le système COQ sont données dans le tableau 2.10. Le résultat final de [89] s'énonce :

Théorème 2.1 (Unification des quasi-termes [89]) *Etant donnés q_1 et q_2 des quasi-termes, soit q_1 et q_2 ne sont pas unifiables, soit ils sont unifiables et il existe alors pour ces deux quasi-termes un unificateur minimal, idempotent, supporté et couvert par le quasi-terme* $\mathsf{ConsArg}(q_1, q_2)$.

Lorsqu'il formalise ce théorème, J. Rouyer prend soin de montrer que si q_1 et q_2 sont des quasi-termes de $Q_\Sigma^T[X]$ ou de $Q_\Sigma^L[X]$ tels que $\lg(q_1) = \lg(q_2)$, alors si q_1 et q_2 sont unifiables, leur unificateur principal est une quasi-substitution de $S_Q^T[X]$. Cette précaution va nous permettre, à l'aide des fonctions et des lemmes définis dans les paragraphes précédents, d'obtenir un théorème équivalent pour les termes et pour les listes de termes.

Théorème 2.2 (Unification des termes) *Etant donnés deux termes t_1 et t_2, soit t_1 et t_2 ne sont pas unifiables, soit ils sont unifiables et il existe alors pour ces deux termes un unificateur minimal, idempotent, supporté et couvert par $l = \langle t_1, \langle t_2, \langle\rangle_0\rangle_1\rangle_2$.*

PREUVE. Soit t_1 et t_2 deux termes. Le théorème 2.1 permet de distinguer deux cas.
(1). Les quasi-termes $\tau_t^q(t_1)$ et $\tau_t^q(t_2)$ sont unifiables, et admettent comme unificateur principal la quasi-substitution s_q de $S_Q^T[X]$. s_q admet donc une preuve p dans $\mathcal{P}^S[s_q]$. Montrons alors que $\tau_s^t(s_q, p)$ est l'unificateur principal de t_1 et t_2. Tout d'abord, montrons que $\tau_s^t(s_q, p)(t_1) = \tau_s^t(s_q, p)(t_2)$. D'après les propositions 2.4 et 2.6, on a :

$$\tau_t^q(t_1) \equiv_T t_1 \quad \tau_t^q(t_2) \equiv_T t_2 \quad s_q \equiv_S \tau_s^t(s_q, p)$$

Le lemme 2.3 permet alors d'établir :

$$s_q(\tau_t^q(t_1)) \equiv_T (\tau_s^t(s_q, p))(t_1) \quad \text{et} \quad s_q(\tau_t^q(t_2)) \equiv_T (\tau_s^t(s_q, p))(t_2)$$

Or, $s_q(\tau_t^q(t_1)) = s_q(\tau_t^q(t_2))$ et donc $\tau_s^t(s_q, p)(t_1) = \tau_s^t(s_q, p)(t_2)$. De plus, puisque s_q est idempotente, alors d'après le lemme 2.8, $\tau_s^t(s_q, p)$ est aussi idempotente. Supposons qu'une substitution s_t unifie t_1 et t_2. D'après le lemme 2.3, $\tau_s^q(s_t)$ unifie $\tau_t^q(t_1)$ et $\tau_t^q(t_2)$. s_q étant minimale, on a alors $s_q \leqslant \tau_s^q(s_t)$. Puisque s_q est idempotente, le lemme 2.6 permet alors d'établir $\tau_s^t(s_q, p) \leqslant s_t$, prouvant ainsi que $\tau_s^t(s_q, p)$ est un unificateur minimal. Reste alors à montrer que $\tau_s^t(s_q, p)$ est supportée et couverte par la liste de termes constituée des deux termes t_1 et t_2. C'est ce que montre le lemme 2.9, puisque, par hypothèse, s_q est supportée et couverte par le quasi-terme $\mathsf{ConsArg}(\tau_t^q(t_1), \tau_t^q(t_2))$.
(2). Les quasi-termes $\tau_t^q(t_1)$ et $\tau_t^q(t_2)$ ne sont pas unifiables. Supposons qu'il existe une substitution s_t qui unifie les termes t_1 et t_2. D'après le lemme 2.3, la quasi-substitution $\tau_s^q(s_t)$ unifie les quasi-termes $\tau_t^q(t_1)$ et $\tau_t^q(t_2)$, ce qui amène une contradiction. ◀

Puisque le problème de l'unification sur les quasi-termes est résolu, et puisque les quasi-termes peuvent représenter des listes de termes, on montre de la même manière que :

Théorème 2.3 (Unification des listes de termes) *Etant données l_1 et l_2 deux listes de termes, de longueur n_1 et n_2, soit l_1 et l_2 ne sont pas* *unifiables, soit elles sont unifiables et il existe alors pour ces deux listes de termes un unificateur minimal, idempotent, supporté et couvert par* $l_1 \bowtie_l l_2$.

PREUVE. Trois cas se présentent :
(1). Si $n_1 \neq n_2$, alors il est clair que l_1 et l_2 ne sont pas unifiables.
(2). Si $n_1 = n_2 = 0$, alors il est facile de montrer que la substitution identité est un unificateur principal de l_1 et l_2.
(3). Si $n_1 = n_2 = n + 1 > 0$, alors l_1 et l_2 peuvent s'écrire :

$$l_1 = \langle t_3, l_3 \rangle_{n+1} \quad \text{et} \quad l_2 = \langle t_4, l_4 \rangle_{n+1}$$

Le théorème 2.1 permet de distinguer deux sous-cas :
(3.1). Les quasi-termes $q_3 = \tau_l^q(n, \tau_t^q(t_3), l_3)$ et $q_4 = \tau_l^q(n, \tau_t^q(t_4), l_4)$ sont unifiables, et admettent la quasi-substitution s_q de $S_Q^T[X]$ comme unificateur principal. s_q admet donc une preuve p dans $\mathcal{P}^S[s_q]$. D'après la proposition 2.4, on a $q_3 \equiv_{L_{n+1}} l_1$ et $q_4 \equiv_{L_{n+1}} l_2$. Le lemme 2.3 permet alors d'établir :

$$s_q(q_3) \equiv_{L_{n+1}} \tau_s^t(s_q, p)(l_1) \quad \text{et} \quad s_q(q_4) \equiv_{L_{n+1}} \tau_s^t(s_q, p)(l_2)$$

Or, par hypothèse, $s_q(q_3) = s_q(q_4)$ ce qui prouve :

$$\tau_s^t(s_q, p)(l_1) = \tau_s^t(s_q, p)(l_2)$$

Montrons alors que $\tau_s^t(s_q, p)$ est l'unificateur principal de l_1 et l_2. Puisque s_q est idempotente, alors d'après le lemme 2.8, $\tau_s^t(s_q, p)$ est aussi idempotente. Supposons qu'une substitution s_t unifie l_1 et l_2. D'après le lemme 2.3, $\tau_s^q(s_t)$ unifie q_3 et q_4. s_q étant minimale, on a alors $s_q \leqslant \tau_s^q(s_t)$. Puisque s_q est idempotente, le lemme 2.6 permet alors d'établir $\tau_s^t(s_q, p) \leqslant s_t$, prouvant ainsi que $\tau_s^t(s_q, p)$ est un unificateur minimal. Reste alors à montrer que $\tau_s^t(s_q, p)$ est supportée et couverte par la liste de termes $l_1 \bowtie_l l_2$. C'est ce que prouve le lemme 2.10, puisque, par hypothèse, s_q est supportée et couverte par $\mathsf{ConsArg}(q_3, q_4)$.
(3.2). Les quasi-termes q_3 et q_4 ne sont pas unifiables. Supposons qu'il existe une substitution s_t qui unifie les listes de termes l_1 et l_2, d'après le lemme 2.3, la quasi-substitution $\tau_s^q(s_t)$ unifie les quasi-termes q_3 et q_4, ce qui amène une contradiction. ◀

Dans ce chapitre l'unification des termes a été obtenue en réutilisant une preuve formelle de l'unification des quasi-termes. Bien sûr, si un programme devait être extrait de cette preuve, il serait probablement moins «efficace» qu'un programme issu d'une preuve obtenue en suivant l'algorithme d'unification. Toutefois, notre objectif n'étant pas d'extraire un programme de cette preuve, cette approche s'est révélée plus rapide et plus courte.

Chapitre 3

Propriétés formelles de la SLD-Résolution

Ce chapitre présente une formalisation de la SLD-résolution dans le calcul des constructions inductives. Il est naturellement divisé en deux parties. Tout d'abord la sémantique opérationnelle des programmes logiques, définie par la SLD-résolution et basée sur le mécanisme d'unification, est formalisée. Comme nous l'avons suggéré dans l'introduction, une des conséquences de cette formalisation est l'explicitation totale et *a priori* des conditions de renommage des variables mises en jeu lors d'une SLD-dérivation. Ces définitions permettront ensuite de formaliser les preuves de deux propriétés «purement» syntaxiques : il s'agit des deux lemmes classiques de généralisation et de commutation. Enfin, la sémantique déclarative des programmes définis est formalisée et les preuves des deux résultats fondamentaux de validité et de complétude de la SLD-résolution sont formalisées. Ces deux preuves sont obtenues en suivant l'approche «traditionnelle» introduite par M.H. van Emden et R.A. Kowalski dans [96]. Cette approche utilise les notions d'interprétations de Herbrand et de points fixes d'un opérateur continu associé à tout programme défini et correspond à celle que l'on peut trouver dans la plupart des ouvrages de base consacrés à la programmation logique. Bien sûr, d'autres approches plus récentes ont été développées pour établir ces résultats, mais notre objectif n'était pas d'écrire *une* preuve formalisée de ces résultats, mais *la* preuve traditionnelle de ces résultats.

3.1 Motivations

Le développement qui suit relève d'une démarche minutieuse dont l'objectif est de *détailler* tous les mécanismes des preuves de la théorie classique de la programmation logique en mettant en évidence le rôle crucial qu'occupent certains *détails* généralement ignorés. Cette formalisation correspond donc à un souci récent de corriger et/ou de préciser des résultats standards incorrectement formulés et/ou prouvés. Un des exemples les plus célèbres d'énoncé inexact concerne une propriété fondamentale de la programmation logique. Il s'agit du théorème de complétude de la SLD-résolution, énoncé par J.W. Lloyd [67], dans un ouvrage qui fait référence en la matière, sous la forme suivante :

> **(Completeness [67])** *Let P be a definite program and G a definite goal. For every correct answer θ for $P \cup \{G\}$, there exists a computed answer σ for $P \cup \{G\}$ and a substitution γ such that $\theta = \gamma\sigma$.*

En 1994, J.C. Shepherdson écrit à propos de cet énoncé : « *This is not true.* » [93] et présente le contre-exemple suivant :

> **(Contre-exemple [93])** Si G est le but $\leftarrow p(x)$ et P le programme contenant l'unique clause $p(f(y,z)) \leftarrow$, alors pour une constante a du langage, la substitution $\theta = \{x/f(a,a)\}$ est une réponse correcte (ce que nous appelerons dans la suite une solution). Les variantes de la clause de P, de la forme $p(f(x_1,x_2)) \leftarrow$, s'unifient avec G via l'unificateur le plus général $\{x/f(x_1,x_2)\}$ (lorsque $x_1 \neq x$ et $x_2 \neq x$). Aussi, la réponse calculée par une dérivation à partir de G est de la forme $\sigma = \{x/f(x_1,x_2)\}$. Toutefois, il n'existe pas de substitution γ telle que $\theta = \gamma\sigma$. En effet, les variables x_1 et x_2 devraient appartenir au domaine d'une telle substitution, alors qu'elles n'appartiennent pas au domaine de la substitution θ.

Pour obtenir un énoncé correct, il suffit donc de remplacer la conclusion $\theta = \gamma\sigma$ par $\theta G = \gamma\sigma G$.

3.1.1 Objets informels et définitions formelles

La sémantique de la programmation logique s'exprime à l'aide d'objets bien connus : substitutions, unificateurs, conditions de renommage. S'agissant d'objets à l'«allure mathématique», il semblerait «naturel» que le nom de ces objets évoque à tous le même objet. Bien sûr, selon les goûts et les besoins de chacun, plusieurs définitions équivalentes peuvent exister. Toutefois, en informatique et tout particulièrement dans le domaine de la programmation logique, il est surprenant de constater que plusieurs définitions non équivalentes coexistent pour un même objet. C'est le cas pour les substitutions : la notion de substitution n'est pas aussi simple qu'il n'y paraît. En 1991, dans [60], H.P. Ko et M.E. Nadel recensent les six définitions suivantes non équivalentes :

- Une substitution est une fonction de X dans $T_\Sigma[X]$ (c'est d'ailleurs la définition que nous avons adopté dans ce développement).

- [28] Une substitution est une fonction θ de X dans $T_\Sigma[X]$ pour laquelle l'ensemble $\{x \in X,\ \theta(x) \neq x\}$ est fini (i.e. θ est l'identité presque partout).
- [91] Une substitution est une fonction θ d'un sous-ensemble fini de X dans $T_\Sigma[X]$ (i.e. θ est une fonction partielle).
- [5, 64, 67] Une substitution est une fonction θ d'un sous-ensemble fini $D \subseteq X$ dans $T_\Sigma[X]$ telle que pour tout $x \in D$, $\theta(x) \neq x$.
- [75] Une substitution est une fonction θ d'un sous-ensemble fini $D \subseteq X$ dans $T_\Sigma[X]$ telle que pour tout $x \in D$, $x \notin \theta(x)$.
- [94] Une substitution est une fonction θ d'un sous-ensemble fini $D \subseteq X$ dans $T_\Sigma[X]$ telle que pour tout $x \in D$ et tout $y \in D$, $x \notin \theta(y)$.

H.P. Ko et M.E. Nadel [60] présentent alors une dicussion sur l'influence de la forme d'une définition dans les théorèmes en étudiant les propriétés vérifiées par la relation «*est une instance de*» en fonction de la définition utilisée. Ainsi, selon la définition employée, cette relation peut ne pas être transitive ou même ne pas impliquer l'unifiabilité des deux termes qu'elle relie. Ces constatations sont surprenantes et ne correspondent pas toujours aux propriétés de l'«objet informel» que l'on croit manipuler. Cette étude permet aux auteurs de corriger certaines erreurs dans la preuve du lemme de généralisation présentée dans [67]. Les substitutions sont donc des objets qu'il faut définir formellement et manipuler avec précautions. D'ailleurs, afin d'éviter les problèmes liés à l'utilisation des substitutions lors de la définition de la sémantique des programmes définis, R.S. Kemp et G.A. Ringwood [58] préfèrent baser directement cette sémantique sur la relation «*est une instance de*».

Evidemment ces problèmes réapparaissent lors de la définition d'un unificateur, de la caractérisation de ses propriétés et de la conception d'algorithmes d'unification. En 1987, E. Eder [28] consacre un article entier aux propriétés «algébriques» des substitutions idempotentes : l'ensemble de substitutions idempotentes auquel est ajouté un «plus grand élément» forme un treillis complet. Ce résultat est utilisé lors de l'unification «simultanée» d'un ensemble fini d'ensembles finis de termes : l'unificateur principal peut être obtenu en considérant la borne supérieure dans le treillis des substitutions idempotentes de l'ensemble des unificateurs (obtenus de manière indépendante) de chacun des ensembles de termes. En 1987, dans le même esprit, J.L. Lassez, M.J. Maher et K. Marriott [64] «revisitent» l'unification en considérant un treillis de contraintes d'égalité et exhibent les différences qui existent entre les définitions d'un unificateur principal que l'on peut rencontrer dans la littérature. Parmi elles, citons les quatre suivantes :

- [86] L'unificateur principal est défini comme étant la substitution produite par l'algorithme présenté dans [86].

- [88] Une substitution θ est un unificateur principal si pour tout autre unificateur σ, $\sigma = \sigma\theta$.
- [67] Une substitution θ est un unificateur principal si pour tout autre unificateur σ, il existe une substitution μ telle que $\sigma = \mu\theta$.
- [94] Un unificateur principal est défini relativement à un ordre partiel sur les termes comme étant la substitution qui transforme deux termes en leur instance commune la plus générale.

Ces travaux ont bien entendu des répercussions dans le domaine de la programmation logique et servent à «reformuler» les principaux résultats et leurs preuves avec plus de rigueur. Ainsi, en 1990, C. Palamidessi [77] définit une algèbre des subtitutions idempotentes, proche de celle proposée par E. Eder dans [28], munie d'opérateurs satisfaisant de «bonnes» propriétés permettant d'envisager la définition d'une sémantique déclarative «compositionnelle» et la parallélisation du processus d'exécution des programmes logiques.

Comme pour les définitions des substitutions ou des unificateurs principaux, plusieurs définitions des conditions de renommage des variables des clauses mises en jeu lors de l'exécution des programmes logiques coexistent dans la littérature. Rappelons tout d'abord qu'étant donnés une clause $A \leftarrow A_1, \cdots, A_n$ et un but $\leftarrow B_1, \cdots, B_q$, la règle de SLD-résolution est une règle d'inférence, dérivée de la règle de coupure du calcul des séquents intuitionniste, définie par :

$$\text{(SLD-résolution)} : \frac{A \leftarrow A_1, \cdots, A_n \qquad \leftarrow B_1, \cdots, B_q}{\leftarrow \theta(B_1, \cdots, B_{i-1}, A_1, \cdots, A_n, B_{i+1}, \cdots, B_q)} \tag{3.1}$$

où θ est un unificateur principal du i-ième atome ($1 \leqslant i \leqslant q$) du but et de A. Afin de pouvoir décomposer le calcul effectué par un programme logique à partir d'une requête en «pas de calcul» élémentaires, cette règle peut être vue comme une relation de transition notée :

$$\begin{array}{lll} & A \leftarrow A_1, \cdots, A_n & \\ & \qquad\qquad\searrow & \\ \leftarrow B_1, \cdots, B_q & \xrightarrow{\theta} & \leftarrow \theta(B_1, \cdots, B_{i-1}, A_1, \cdots, A_n, B_{i+1}, \cdots, B_q) \end{array}$$

Une dérivation est alors définie comme une succession de transitions :

$$\begin{array}{ccccccccc} & C_0 & & C_1 & & & C_{i-1} & & C_i \\ & \searrow & & \searrow & & & \searrow & & \searrow \\ R_0 & \xrightarrow{\theta_0} & R_1 & \xrightarrow{\theta_1} & \cdots & R_{i-1} & \xrightarrow{\theta_{i-1}} & R_i & \xrightarrow{\theta_i} & \cdots \end{array}$$

Différentes conditions de renommage des clauses non équivalentes apparaissent dans la littérature :

- [7, 63, 65] Chaque clause C_i ne partage pas de variables avec R_i : $var(C_i) \cap var(R_i) = \varnothing$.

- [67] Chaque clause C_i ne contient pas de variables apparaissant déjà dans la dérivation jusqu'à R_i.
- [76] Chaque clause C_i ne partage pas de variables avec R_i et R_0 : $var(C_i) \cap (var(R_0) \cup var(R_i)) = \varnothing$.
- [5] Chaque clause C_i ne partage aucune variable avec R_0, C_0, C_1 , ... , C_{i-1} : $var(C_i) \cap (var(R_0) \cup var(C_0) \cup \cdots \cup var(C_{i-1})) = \varnothing$.

Considérons par exemple le programme $P = \{p(f(x)) \leftarrow p(y)\}$. Avec les conditions de renommage issues de [7, 63, 65], on peut construire la dérivation suivante :

$$p(z) \xrightarrow{C_1, \left[\begin{array}{c} z \\ f(x) \end{array}\right]} p(y) \xrightarrow{C_2, \left[\begin{array}{c} y \\ f(x) \end{array}\right]} p(z) \rightarrow \cdots$$

où C_1 est la clause $p(f(x)) \leftarrow p(y)$ et C_2 est la clause $p(f(x)) \leftarrow p(z)$. Si l'on considère à présent les conditions de renommage issues de [76], on ne peut plus utiliser la clause C_2 lors de la deuxième transition puisque $z \in var(C_2) \cap var(p(z))$. On peut utiliser à la place la clause $p(f(x)) \leftarrow p(w)$, notée C_2', et obtenir la dérivation :

$$p(z) \xrightarrow{C_1, \left[\begin{array}{c} z \\ f(x) \end{array}\right]} p(y) \xrightarrow{C_2', \left[\begin{array}{c} y \\ f(x) \end{array}\right]} p(w) \rightarrow \cdots$$

Or, en utilisant les conditions de renommage issues de [5], cette clause C_2' ne peut plus être utilisée puisque $x \in var(C_1) \cap var(C_2')$. On peut utiliser à la place la clause $p(f(v)) \leftarrow p(w)$, notée C_2'', et obtenir la dérivation :

$$p(z) \xrightarrow{C_1, \left[\begin{array}{c} z \\ f(x) \end{array}\right]} p(y) \xrightarrow{C_2'', \left[\begin{array}{c} y \\ f(v) \end{array}\right]} p(w) \rightarrow \cdots$$

Si le renommage des clauses permet d'établir les propriétés fondamentales de la SLD-résolution, il est aussi nécessaire pour définir la sémantique opérationnelle des programmes logiques : ce renommage a au moins deux rôles :

- Il permet d'éviter un échec du mécanisme d'unification. Par exemple, soit P le programme défini $\{p(f(x)) \leftarrow\}$ et R la requête $p(x)$. Sans un renommage de la clause de P *en dehors* des variables de R, l'unification de $p(x)$ avec $p(f(x))$ échoue. En effet, le «test d'occurrence» est une opération fondamentale de l'algorithme d'unification qui élimine les équations de la forme $x = t[x]$ (où $t[x]$ désigne un terme fonctionnel contenant la variable x) qui n'ont pas de solutions sur les termes finis.
- Il garantit l'obtention d'une substitution résultat la plus générale. Par exemple, soit P le programme $\{p(y) \leftarrow\}$ et R la requête $p(x), p(y)$.

Supposons que la clause de P ne soit pas renommée et qu'à partir de R, le premier atome soit sélectionné. On obtient alors la dérivation :

$$p(x), p(y) \overset{\{x/y\}}{\rightarrow_P} p(y) \rightarrow_P \Box$$

dont la réponse est $\rho_1 = \{x/y\}$. Le renommage de la clause de P *en dehors* des variables apparaissant dans R permet d'obtenir une réponse plus générale. En effet, si l'on renomme cette clause en $p(x_1) \leftarrow$ pour la première transition puis en $p(x_2) \leftarrow$ pour la deuxième transition, on obtient la dérivation :

$$p(x), p(y) \overset{\{x/x_1\}}{\rightarrow_P} p(y) \overset{\{y/y_1\}}{\rightarrow_P} \Box$$

dont la réponse est $\rho_2 = \{x/x_1, y/y_1\}$ et est plus générale que ρ_1.

3.1.2 Preuves explicites

Prendre en compte de manière explicite la «partie renommage» de la SLD-résolution complique considérablement les preuves des propriétés classiques de la programmation logique. Le processus de renommage doit être considéré comme une composante à part entière de la sémantique opérationnelle des programmes logiques. C'est lui qui permet d'établir des propriétés sur les variables apparaissant dans une dérivation. Ces propriétés sont essentielles et ne constituent pas un simple *détail* «technique» comme le montrent les exemples qui suivent.

Lemme de généralisation

Le lemme de généralisation, préliminaire au théorème de complétude de la SLD-résolution, s'énonce habituellement par :

> **(Lifting lemma [5])** *Let P be a program, N a goal and θ a substitution. Suppose that there exists an SLD-refutation of $P \cup \{\theta N\}$ with the sequence of mgu's $\theta_0, \ldots, \theta_n$. Then, there exists an SLD-refutation of $P \cup \{N\}$ with the sequence of mgu's $\theta'_0, \ldots, \theta'_n$ such that $\theta'_n \ldots \theta'_0$ is more general than $\theta_n \ldots \theta_0 \theta$.*

Aucune hypothèse sur la substitution θ n'est posée dans cet énoncé. Néanmoins, si l'on veut pouvoir utiliser exactement les mêmes variantes de clauses dans les deux dérivations, ce qui est toujours implicitement suggéré, certaines hypothèses sont nécessaires.

En effet, soit C_1 la variante de clause utilisée lors de la première étape de résolution de la dérivation en hypothèse du lemme. Tandis que les conditions de renommage de C_1 stipulent que C_1 ne partage pas de variables avec la requête θN, il faut, pour pouvoir construire la dérivation en conclusion du lemme, que C_1 ne partage pas non plus de variables avec N. Or, sans aucune hypothèse supplémentaire, il se peut très bien qu'une variable de N n'apparaisse pas dans θN et soit donc présente dans C_1. Par exemple, si P est le programme $\{p(f(v), f(z)) \leftarrow\}$, N le but $\leftarrow p(f(a), y)$ et θ est la substitution $\{y/f(x)\}$, alors à partir du but initial $\theta N = \leftarrow p(f(a), f(x))$, rien n'empêche de renommer la clause de P à l'aide de la substitution de renommage $r = \{z/y\}$ et d'obtenir la dérivation :

$$\underbrace{p(f(a), f(x))}_{\theta N} \overset{\{v/a, x/y\}}{\rightarrow_P} \Box$$

Considérons à présent la dérivation que l'on peut obtenir à partir du but N : la clause de P doit être renommée *en dehors* des variables de N. La substitution de renommage r, utilisée dans la dérivation précédente ne convient donc plus, puisqu'elle renomme z en y qui apparaît dans N. En effet, lors de la dérivation à partir de θN, le renommage de la clause C utilisée satisfait seulement $var(\theta N) \cap var(rC) = \varnothing$ alors que la construction d'une dérivation à partir de N nécessite la condition $var(N) \cap var(rC) = \varnothing$. A chaque transition, la clause C doit donc être renommée *en dehors* des variables de N.

D'autres précautions à prendre concernent le lien que doit entretenir la substitution θ avec N. En effet, si θ affecte des variables non présentes dans N, quelques problèmes peuvent apparaître. Par exemple, si P est le programme $\{p(f(y), f(z)) \leftarrow\}$, N le but $\leftarrow p(f(a), y)$ et θ la substitution $\{x/w\}$, alors on peut renommer la clause de P à l'aide de la substitution de renommage $r = \{y/x\}$ et construire les dérivations :

$$\underbrace{p(f(a), y)}_{\theta N} \overset{\rho_1 = \{x/a\,;\, y/f(z)\}}{\rightarrow_P} \Box \qquad \underbrace{p(f(a), y)}_{N} \overset{\rho_2 = \{x/a\,;\, y/f(z)\}}{\rightarrow_P} \Box$$

Cependant, $\rho_2 = \{x/a\,;\, y/f(z)\} \leqslant \rho_1\theta = \{x/w\,;\, y/f(z)\}$ n'est pas vérifié. Pour éviter cette situation, nous supposerons que les clauses utilisées sont renommées *en dehors* des variables du domaine de θ.

Lemme de commutation

Un problème similaire se pose lors de la preuve du lemme de commutation. Ce résultat classique, assurant que le choix de l'atome sélectionné, lors d'une étape de résolution, relève d'un non-déterminisme par indifférence (*don't care non-determinism*), s'énonce habituellement :

> **(Switching lemma [63])** *If during a derivation, two atoms L_1 and L_2 are successively selected, then they can also be selected in the reverse order and the derived states are the same up to renaming of variables.*

$$
\begin{array}{ccc}
& \underbrace{(\cdots, L_1, \cdots, L_2, \cdots)}_{R_0} & \\
& C_1 \swarrow \quad \searrow C_2 & \\
\underbrace{\theta(\cdots, C_1^-, \cdots, L_2, \cdots)}_{R_1} & & \sigma(\cdots, L_1, \cdots, C_2^-, \cdots) \\
C_2 \downarrow & & \downarrow C_1 \\
\eta\theta(\cdots, C_1^-, \cdots, C_2^-, \cdots) & \approx & \mu\sigma(\cdots, C_1^-, \cdots, C_2^-, \cdots)
\end{array}
$$

Ici encore, quelques précautions sont nécessaires si l'on veut pouvoir utiliser les mêmes variantes des clauses C_1 et C_2 dans les deux dérivations (ce qui est toujours suggéré dans la preuve de ce lemme). En effet, le renommage des clauses C_1 et C_2 dans la dérivation en hypothèse du lemme satisfait normalement :

$$var(C_1) \cap var(R_0) = \varnothing \qquad \text{et} \qquad var(C_2) \cap var(R_1) = \varnothing$$

De plus les variables présentes dans la requête R_1 proviennent soit de la clause utilisée, soit de la requête R_0, et on a :

$$\forall x \quad x \in var(R_1) \Rightarrow (x \in var(C_1) \vee x \in var(R_0)) \tag{3.2}$$

Cependant, si l'on veut pouvoir sélectionner l'atome L_2, avant L_1, en utilisant la même variante de la clause C_2, il faut que cette variante satisfasse :

$$var(C_2) \cap var(R_0) = \varnothing$$

Or rien dans l'hypothèse du lemme de commutation ne garantit une telle propriété, et selon les conditions de renommage imposées dans la définition d'une dérivation, il est tout à fait possible qu'une variable présente dans la requête R_0, mais n'apparaissant pas dans R_1, ait été utilisée dans la variante de C_2. Pour contourner ce problème, et d'après (3.2), il faut que le renommage des clauses soit tel que :

$$var(C_2) \cap var(C_1) = \varnothing \tag{3.3}$$

C'est heureusement le cas si l'on utilise les hypothèses de renommage formulées dans [5] (i.e. du fait du renommage de la clause C_i *en dehors* des clauses $C_0, C_1, \cdots, C_{i-1}$) mais ce n'est plus le cas si l'on utilise les conditions énoncées dans [7, 63, 65, 67, 76].

«Renommer» une dérivation

Formaliser une preuve nous conduit donc à l'établir à un niveau de *détail* supérieur à celui généralement exposé. Dans le domaine de la programmation logique, un exemple typique de *détail* concerne le renommage de la clause utilisée lors d'une étape de résolution. La raison pour laquelle il est fréquent et admis de s'affranchir de ces *détails* est l'existence d'un lemme dû à J.W. Lloyd et J.C. Shepherdson [68], affirmant que si deux dérivations diffèrent seulement dans le choix des variables utilisées pour renommer les clauses (et par conséquent dans les unificateurs utilisés), alors les états dérivés sont équivalents à un renommage près des variables.

> **(Uniqueness)** *If two SLDNF-derivations differ only in the variants of clauses and the mgu which are used, then the resultants are variants of each other.*

Autrement dit, l'existence d'une dérivation ne dépend pas du choix des variables de renommage : il suffit que de «bonnes» conditions de séparation des variables soit satisfaites. C'est précisément ce résultat qui permet de supposer à tout moment des hypothèses sur les variables présentes dans une clause. Ainsi, la plupart du temps, ce renommage est fait implicitement et afin d'alléger les théorèmes et leurs preuves, on passe sous silence cette partie du «calcul» : on parle de variables «fraîches» et les théorèmes sont énoncés «à un renommage près». Toutefois, pour pouvoir supposer telle ou telle hypothèse, il est nécessaire d'appliquer ce lemme en instanciant les ensembles de variables mis en jeu. L'adaptation de ce lemme d'indépendance dans le calcul des constructions inductives nous a donc conduit à expliciter comment, à partir d'une requête R admettant une dérivation d_1 et d'un ensemble fini de variables Z, on peut construire une dérivation d_2, à partir de la même requête, n'utilisant pour renommer les clauses que des variables appartenant à un ensemble Y tel que Z et Y soient disjoints. Cette construction passe bien sûr par l'explicitation des substitutions de renommage et des unificateurs utilisés et peut être vue comme une opération de renommage portant sur une dérivation.

$$\boxed{\begin{array}{c}\underbrace{R \overset{*}{\rightarrow} R_1}_{d_1} \\ \\ Z\end{array}} \quad \begin{array}{c}\rightarrow \\ (\Rightarrow \exists)\end{array} \quad \boxed{\begin{array}{c}\underbrace{R \overset{(Y)*}{\rightarrow} R_2}_{d_2} \\ \\ Z \cap Y = \varnothing \\ \\ R_1 \approx R_2\end{array}}$$

Nous allons voir que ce résultat (formalisé par le lemme 3.18) est absolument indispensable si l'on veut pouvoir formaliser la preuve du théorème de complétude.

«Combiner» des dérivations

En effet la preuve du théorème de complétude s'obtient par induction sur la longueur de la requête initiale et une étape de cette preuve consiste à affirmer que si chaque atome d'un ensemble d'atomes ne contenant pas de variables admet une réfutation, alors en combinant ces réfutations, on peut obtenir une réfutation à partir de la requête constituée de tous les atomes présents dans cet ensemble.

> [67] (...) *by induction hypothesis,* $P \cup \{\theta B_i\}$ *has a refutation for* $i = 1, \ldots, k$. *Because each* θB_i *is ground, these refutations can be combined into a refutation of* $P \cup \{\theta B_1, \ldots \theta B_k\}$ (...)

Cependant «combiner» des dérivations est une opération délicate, puisque, afin que la dérivation à construire satisfasse à chaque étape les hypothèses de séparation des variables, il est nécessaire de pouvoir renommer chacune des réfutations initiales. Ici encore, il s'agit d'expliciter comment on peut construire la dérivation finale en exhibant les substitutions de renommage et les unificateurs qui vont permettre d'obtenir une telle dérivation.

$$\boxed{\begin{array}{c} \theta B_1 \stackrel{*}{\rightarrow} \Box \\ \theta B_2 \stackrel{*}{\rightarrow} \Box \\ \vdots \\ \theta B_k \stackrel{*}{\rightarrow} \Box \end{array}} \quad \underset{(\Rightarrow \exists)}{\rightarrow} \quad \boxed{\theta B_1, \theta B_2, \cdots, \theta B_k \stackrel{*}{\rightarrow} \Box}$$

Ce résultat (formalisé par le lemme 3.19) s'obtient par induction sur le nombre de réfutations initiales et sa preuve utilise constamment le lemme de «renommage» des dérivations présenté dans le paragraphe précédent.

Pour finir, signalons que l'article qui semble le plus récent et le plus *détaillé* concernant le rôle du renommage des variables en programmation logique a été écrit en 1994 par J.C. Shepherdson [93]. Si la formalisation présentée dans ce chapitre ne contient aucun résultat nouveau, elle permet toutefois d'éclairer toute une partie du calcul effectué par un programme logique : le renommage des variables. La formalisation de ce processus dans le calcul des constructions inductives constitue une approche permettant de garantir la validité des résultats prouvés.

3.2 Aspects syntaxiques

3.2.1 Programmes définis

Nous donnons tout d'abord les définitions formelles des atomes, requêtes, clauses et programmes définis. Les définitions de ces objets dans le système Coq sont présentées dans le tableau 3.2. Ces objets sont construits à partir des termes et d'une signature relationnelle Π (i.e. un ensemble infini dénombrable de symboles de prédicat, muni d'une application $ar : \Pi \rightarrow \mathbb{N}$).

Atomes

L'ensemble $At_{\Sigma,\Pi}[X]$ des atomes est défini inductivement par : si p est un symbole de Π d'arité n et l une liste de termes de longueur n, alors $\mathsf{pl}(p,l)$ est un atome. L'application d'une substitution à un atome est définie par :

$$\forall s \in S[X] \quad \forall p \in \Pi \quad \forall l \in L_{ar(p),\Sigma}[X] \quad s(\mathsf{pl}(p,l)) = \mathsf{pl}(p, s(l))$$

Deux atomes $a_1 = \mathsf{pl}(p_1,l_1)$ et $a_2 = \mathsf{pl}(p_2,l_2)$ sont unifiables si $p_1 = p_2$ et si il existe un unificateur principal θ pour les listes de termes l_1 et l_2. Dans ce cas, θ est un unificateur principal de a_1 et a_2, ce qui sera noté par la suite $\mathsf{mgu}(\theta, a_1, a_2)$.

Requêtes

Une requête est une liste finie, éventuellement vide, d'atomes : $r_\varnothing$ est une requête (c'est la requête «vide», parfois notée $\square$) et si a est un atome et r une requête, alors $c_r(a,r)$ est une requête. L'ensemble $R_{\Sigma,\Pi}[X]$ est donc défini inductivement par :

$$R_{\Sigma,\Pi}[X] ::= r_\varnothing \mid c_r(At_{\Sigma,\Pi}[X], R_{\Sigma,\Pi}[X])$$

Les fonctions «classiques» sur les listes sont définies pour les requêtes (voir tableau 3.1) et serviront lors de la spécification de la sémantique opérationnelle des programmes définis. Requêtes et buts (i.e. clauses négatives) ont la même structure syntaxique : il s'agit de listes d'atomes. Aussi, ces deux objets ne seront distingués qu'à partir du moment où nous souhaiterons les interpréter : deux schémas d'interprétation seront définis pour les «requêtes».

Clauses définies

L'ensemble des clauses définies est l'ensemble produit :

$$C_{\Sigma,\Pi}[X] ::= At_{\Sigma,\Pi}[X] \times R_{\Sigma,\Pi}[X]$$

L'application d'une substitution à une clause est définie par :

$$\forall c = \langle a, r\rangle \in C_{\Sigma,\Pi}[X] \quad \forall s \in S[X] \quad s(c) = \langle s(a), s(r)\rangle$$

Si c est la clause obtenue à partir de l'atome a et de la requête r, nous désignerons par c^+ l'atome a, appelé tête de la clause, et par c^- la requête r, appelée le corps de la clause. Dans le système COQ, `Fst` et `Snd` correspondent aux projections sur un type produit et permettent de définir les fonctions `head_c` et `body_c` retournant respectivement la tête d'une clause et son corps.

longueur d'une requête (`Length_r:request->nat`)
$\lg(r) = \begin{cases} 0 & \text{si}\, r = r_\varnothing \\ 1 + \lg(r') & \text{si}\, r = c_r(a, r') \end{cases}$
concaténation de deux requêtes (`App_r:request->request->request`)
$r_1 \bowtie_r r_2 = \begin{cases} r_2 & \text{si}\, r_1 = r_\varnothing \\ c_r(a, r' \bowtie_r r_2) & \text{si}\, r_1 = c_r(a, r') \end{cases}$
$(n+1)$-*ième* atome d'une requête (`at_n_req:nat->atom->request->atom`)
$r_{/n,a} = \begin{cases} a & \text{si}\, r = r_\varnothing \\ \begin{cases} a' & \text{si}\, n = 0 \\ (r')_{/n-1,a} & \text{si}\, n > 0 \end{cases} & \text{si}\, r = c_r(a', r') \end{cases}$
requête obtenue en remplaçant le $(n+1)$-*ième* atome d'une requête r par la requête r_1 (`change_n_r:nat->request->request->request`)
$r[n \leftarrow r_1] = \begin{cases} r_\varnothing & \text{si}\, r = r_\varnothing \\ \begin{cases} r_1 \bowtie_r r' & \text{si}\, n = 0 \\ c_r(a', r'[n-1 \leftarrow r_1]) & \text{si}\, n > 0 \end{cases} & \text{si}\, r = c_r(a', r') \end{cases}$
application d'une substitution à une requête
$\forall s \in S[X]\, \forall r \in R_{\Sigma,\Pi}[X] \quad s(r) = \begin{cases} r_\varnothing & \text{si}\, r = r_\varnothing \\ c_r(s(a), s(r')) & \text{si}\, r = c_r(a, r') \end{cases}$

TAB. 3.1 – *Opérations sur les requêtes*

Programmes définis et clauses de Horn

Les ensembles $P_{\Sigma,\Pi}[X]$ et $H_{\Sigma,\Pi}[X]$ des programmes définis et des clauses de Horn sont définis inductivement par :

$$P_{\Sigma,\Pi}[X] ::= P_\varnothing \mid c_p(C_{\Sigma,\Pi}[X], P_{\Sigma,\Pi}[X])$$
$$H_{\Sigma,\Pi}[X] ::= h_p(P_{\Sigma,\Pi}[X]) \mid h_r(R_{\Sigma,\Pi}[X], H_{\Sigma,\Pi}[X])$$

3.2.2 SLD-Résolution

La SLD-résolution (*Selection Linear Definite*) détermine la sémantique opérationnelle des programmes définis. Le mécanisme de base d'exécution de ces programmes, basé sur l'unification, est un cas particulier du principe de résolution de A.J. Robinson [86, 87]. Il s'agit d'une méthode syntaxique, correcte et complète, de dérivation de formules à partir de formules exprimées sous forme clausale.

SLD-Résolution et transitions

La SLD-résolution (définie en 3.1) est une règle d'inférence définissant une relation de déduction et peut être vue comme une règle de transition, notée $\overset{n,r,C}{\rightarrow}_P$, entre «états de résolution». Un état de résolution est un couple

Π	`Definition predic : Set := nat.` `arity_p : predic->nat`
$At_{\Sigma,\Pi}[X]$	`Inductive atom : Set :=` `  pl:(p:predic)(list_term (arity_p p))->atom.`
mgu	`at_mgu:subst->atom->atom->Prop`

$R_{\Sigma,\Pi}[X]$	`Inductive request : Set :=`
	`  true_req:request |`
	`  cons_req:atom->request->request.`

$C_{\Sigma,\Pi}[X]$	`Definition clause:Set:=(atom * request).`

$P_{\Sigma,\Pi}[X]$	`Inductive program : Set :=`
	`  nil_pgm:program |`
	`  cons_pgm:clause->program->program.`

$H_{\Sigma,\Pi}[X]$	`Inductive horn : Set :=`
	`  hp:program->horn |`
	`  hr:request->horn->horn.`

TAB. 3.2 – *Objets syntaxiques*

constitué d'une substitution, qui correspond à la construction incrémentale de la substitution résultat, et d'une requête résiduelle. Etant donnés un état de résolution $\rho.R$, et une clause C, si le n-ième atome de R s'unifie avec la tête de la clause C, renommée à l'aide d'une substitution de renommage r, via l'unificateur principal θ, et si certaines conditions de séparation des variables, notées HV, sont satisfaites, alors on peut passer de l'état $\rho.R$ à l'état :

- dont la substitution est obtenue par composition de ρ et θ
- dont la requête résiduelle est la requête R dans laquelle le n-ième atome a été remplacé par le corps de la clause C renommée, et sur laquelle la substitution θ est appliquée

$$(\text{Transition}) : \frac{\mathsf{mgu}(\theta, R_{/n}, r(C^+)) \wedge \mathsf{HV}}{\rho.R \overset{n,r,C}{\rightarrow}_P \theta\rho.\theta R[n \leftarrow r(C^-)]} \tag{3.4}$$

A chaque étape de résolution, deux choix sont faits : le choix de l'atome et le choix de la clause dont la tête s'unifie avec l'atome sélectionné. Ces choix introduisent deux non-déterminismes de nature différente : le premier correspond à un non-déterminisme par indifférence «*don't care*» (voir lemme de commutation), tandis que le second correspond à un non-déterminisme par ignorance «*don't know*» (lorqu'il est nécessaire d'envisager toutes les dérivations possibles, on les organise alors sous forme d'arbre SLD).

Nous définissons à présent de manière formelle les transitions définies par la règle de résolution en explicitant les conditions de séparation des variables

HV (voir tableau 3.3). Tout d'abord, l'ensemble des états de résolution est défini comme l'ensemble produit $S[X] \times R_{\Sigma,\Pi}[X]$. L'ensemble des substitutions de renommage, $S_R[X]$, est l'ensemble des applications de X vers X, sur lequel sont définies les propriétés classiques des substitutions (domaine, image, ...). De plus, un prédicat, noté P^R, caractérisant les substitutions de renommage idempotentes et «injectives sur leur domaine», est défini par :

$$\forall s_r \in S_R[X] \begin{pmatrix} \forall x, y \in dom(s_r)\,(x \neq y \Rightarrow s_r(x) \neq s_r(y)) \wedge \\ \forall x \in X\,(x \in dom(s_r) \Rightarrow x \notin range(s_r)) \end{pmatrix} \Rightarrow P^R(s_r)$$

Nous présentons maintenant, de manière explicite, les conditions de renommage des objets intervenant lors d'une transition. Comme nous l'avons vu, la clause utilisée doit être renommée *en dehors* des variables de la requête. De plus, nous imposons à l'état initial $\rho.R$ de vérifier $\rho R = R$. Une condition supplémentaire est nécessaire si l'on veut pouvoir utiliser, de manière suffisament simple, la règle de résolution telle qu'elle est formulée en (3.4). En effet, lors d'une transition :

$$\rho.R \overset{n,r,C}{\rightarrow_P} \theta\rho.\theta R[n \leftarrow r(C^-)]$$

la substitution ρ doit laisser intactes les variables de l'image de la substitution de renommage r (on souhaite que $\rho r(C) = r(C)$) afin que lorsque R s'écrit $\rho R'$, on ait :

$$(\rho R')[n \leftarrow r(C^-)] = \rho(R'[n \leftarrow r(C^-)])$$

Enfin, nous imposons le renommage de toutes les variables apparaissant dans la clause utilisée afin de pouvoir, par la suite, manipuler les transitions, puis les dérivations, de manière suffisamment simple. Pour ce faire, les trois prédicats suivants sont définis : soit r une substitution de renommage, C une clause définie, R une requête et ρ une substitution :

- $P_{RC}(r, C)$ est satisfait si la substitution de renommage r affecte toutes les variables de la clause C et seulement ces variables.
- $P_{RR}(r, R)$ est satisfait si aucune variable de l'image de la substitution de renommage r n'apparaît dans la requête R.
- $P_{RS}(r, \rho)$ est satisfait si aucune variable de l'image de la substitution de renommage r n'apparaît dans le domaine de la substitution ρ.

L'ensemble des transitions, noté Γ, et un prédicat, noté P^Γ et caractérisant les transitions satisfaisant la règle de résolution munie des conditions de renommage que nous venons de décrire, peuvent à présent être définis :

Définition 3.1 (Transitions)

- *Une transition est obtenue à partir d'un état initial de résolution $\rho_i.R_i$, d'un programme défini P, d'un entier n (position de l'atome sélectionné* *dans la requête), d'une clause définie C, d'une substitution de renommage r, d'une substitution (unificateur utilisé) θ, d'un atome (pour pouvoir utiliser la fonction $r_{/n,a}$ lorsque la requête est vide) et d'un état de résolution final $\rho_f.R_f$. Elle est notée $\langle \rho_i.R_i, P, n, C, r, \theta, a, \rho_f.R_f \rangle$.*

- *Pour toute transition $\langle \rho_i.R_i, P, n, C, r, \theta, a, \rho_f.R_f \rangle$, le prédicat P^Γ, sur Γ, est défini inductivement par :*

$$(P^\Gamma) : \frac{\begin{array}{ll} n+1 \leqslant \lg(R_i) & \rho_i R_i = R_i \\ C \in P & \mathsf{mgu}(\theta, r_{/n,a}(R_i), r(C^+)) \\ P^R(r) & P_{RC}(r, C) \\ P_{RS}(r, \rho_i) & P_{RR}(r, R_i) \\ \rho_f = \theta\rho_i & R_f = \theta R_i[n \leftarrow r(C^-)] \end{array}}{P^\Gamma(\langle \rho_i.R_i, P, n, C, r, \theta, a, \rho_f.R_f \rangle)}$$

qui exprime $\rho_i.R_i \overset{n,r,C}{\rightarrow_P} \rho_f.R_f$.

Afin de pouvoir construire des dérivations constituées de transitions satisfaisant le prédicat P^Γ, nous montrons que la propriété $\rho_i R_i = R_i$ est conservée lors d'une transition satisfaisant ce prédicat. Pour cela, nous montrons tout d'abord :

Lemme 3.1 *Si la transition $\rho.R \overset{n,r,C}{\rightarrow_P} \theta\rho.\theta R[n \leftarrow r(C^-)]$ satisfait le prédicat P^Γ, alors $\rho\theta R[n \leftarrow r(C^-)] = \theta R[n \leftarrow r(C^-)]$.*

PREUVE. Soit, $x \in var(\theta R[n \leftarrow r(C^-)])$ et montrons que $x \notin dom(\rho)$. Deux cas se présentent. Soit $x \in var(R[n \leftarrow r(C^-)])$, soit $x \in range(\theta)$. Dans ces deux cas, puisque θ est, par hypothèse, un unificateur principal de $R_{/n}$ et $r(C^+)$, soit $x \in var(R)$, soit $x \in r(C)$. Dans le premier cas, l'hypothèse $\rho R = R$ permet de conclure. Dans le second cas, les hypothèses de séparation $P_{RC}(r, C)$ et $P_{RS}(r, \rho)$ permettent de conclure. ◀

Ce lemme permet de montrer :

Lemme 3.2 *Si la transition $\rho_i.R_i \overset{n,r,C}{\rightarrow_P} \rho_f.R_f$ satisfait le prédicat P^Γ, alors l'état final vérifie $\rho_f R_f = R_f$.*

PREUVE. Soit $\rho.R \overset{n,r,C}{\rightarrow_P} \theta\rho.\theta R[n \leftarrow r(C^-)]$ une transition satisfaisant P^Γ. D'après le lemme 3.1, on a $\theta\rho\theta R[n \leftarrow r(C^-)] = \theta\theta R[n \leftarrow r(C^-)]$. Par définition, θ est idempotente (unificateur principal), et on obtient alors $\theta\rho\theta R[n \leftarrow r(C^-)] = \theta R[n \leftarrow r(C^-)]$. ◀

SLD-dérivations

Nous cherchons à construire l'ensemble des dérivations constituées d'un nombre fini de transitions «composables» et vérifiant le prédicat P^Γ (voir tableau 3.5) :

$$\rho_0.R_0 \overset{n_0,r_0,C_0}{\rightarrow_P} \ldots \overset{n_{k-1},r_{k-1},C_{k-1}}{\rightarrow_P} \rho_k.R_k$$

Etats de résolution	Definition state:Set:=(subst*request).
$S_R[X]$	Definition rename:=var->var.
P^R	Definition good_rename:rename->Prop:= [sr:rename](x:var) (((y:var)(~x=y)->(rdom sr x)->(rdom sr y)-> ~(sr x)=(sr y)) /\ ((rdom sr x)->~(rrange sr x))).
P_{RC}	Definition rename_c:rename->clause->Prop:= [r:rename][c:clause] (((x:var)(IS_IN_LV x (var_cl c))->(rdom r x)) /\ ((x:var)~(IS_IN_LV x (var_cl c))->~(rdom r x))).
P_{RR}	Definition rename_out_req:rename->request->Prop:= [r1:rename][r2:request] ((x:var)(IS_IN_LV x (var_req r2))->~(rrange r1 x)).
P_{RS}	Definition rename_out_dom:rename->subst->Prop:= [r:rename][s:subst] ((x:var)(rdom r x)->(s (r x))=(tv (r x))).
Γ	Inductive trans:Set:=trans_cons:state->program->nat ->clause->rename->subst->atom->state->trans.
P^Γ	Inductive rsl [ei:state;p:program;n:nat; c:clause;r:rename;s:subst;a:atom;ef:state] : Prop := rsl_init : (*H1*) (le (S n) (Length_r (Snd ei))) (*H2*) ->(is_in_p c p) (*H3*) ->(at_mgu s (at_n_req n a (Snd ei)) (rsubst_at r (head_c c))) (*H4*) ->(good_rename r) (*H5*) ->(subst_req (Fst ei) (Snd ei)) = (Snd ei) (*H6*) ->(rename_c r c) (*H7*) ->(rename_out_dom r (Fst ei)) (*H8*) ->(rename_out_req r (Snd ei)) (*H9*) ->ef=(<subst,request> (([x:var](Subst_t s ((Fst ei) x))) , (subst_req s (change_n_r n (rsubst_req r (body_c c)) (Snd ei))))) ->(rsl ei p n c r s a ef).
	Definition RSL:trans->Prop:=[t:trans]<Prop>Case t of [ei:state][p:program][n:nat][c:clause] [r:rename][s:subst][a:atom][ef:state] (rsl ei p n c r s a ef) end.

TAB. 3.3 – *Transitions*

«concatenation» sur $\mathbb{D}_g$		
$d_1 \bowtie_g d_2$	$=$	$\begin{cases} d_c^g(d_1,t) & \text{si}\, d_2 = d_t^g(t) \\ d_c^g(d_1 \bowtie_g d, t) & \text{si}\, d_2 = d_c^g(d,t) \end{cases}$
«concatenation» sur $\mathbb{D}_d$		
$d_1 \bowtie_d d_2$	$=$	$\begin{cases} d_c^d(t,d_2) & \text{si}\, d_1 = d_t^d(t) \\ d_c^d(t, d \bowtie_d d_2) & \text{si}\, d_1 = d_c^d(t,d) \end{cases}$
$\tau_d^g : \mathbb{D}_d \to \mathbb{D}_g$		
$\tau_d^g(d)$	$=$	$\begin{cases} d_t^g(t) & \text{si}\, d = d_t^d(t) \\ d_t^d(t) \bowtie_g \tau_d^g(d_0) & \text{si}\, d = d_c^d(t,d_0) \end{cases}$
$\tau_g^d : \mathbb{D}_g \to \mathbb{D}_d$		
$\tau_g^d(d)$	$=$	$\begin{cases} d_t^d(t) & \text{si}\, d = d_t^g(t) \\ \tau_g^d(d_0) \bowtie_d d_t^d(t) & \text{si}\, d = d_c^g(d_0,t) \end{cases}$

TAB. 3.4 – *Liens entre $\mathbb{D}_g$ et $\mathbb{D}_d$*

que nous écrirons $\rho_0.R_0 \overset{*}{\to}_P \rho_k.R_k$.

Comme pour les transitions, nous allons définir un ensemble $\mathbb{D}$, plus général (l'ensemble des listes de transitions), puis nous allons définir un prédicat sur cet ensemble, caractérisant les dérivations satisfaisant les conditions souhaitées. Toutefois, de la forme des constructeurs de la définition inductive de $\mathbb{D}$, dépendra la forme de l'hypothèse d'induction obtenue lors d'un raisonnement par induction sur un élément de $\mathbb{D}$. Alors que la plupart de ces raisonnements se font en utilisant un «schéma d'induction gauche», le théorème de validité de la SLD-résolution s'obtiendra à l'aide d'un «schéma d'induction droit». C'est pourquoi nous définissons deux ensembles de dérivations (et les fonctions de transformations correspondantes, présentées dans le tableau 3.4), ce qui nous permettra par la suite de choisir la «forme» de l'hypothèse d'induction :

$$\underbrace{e_1 \overset{*}{\to}_P e_2}_{\substack{\text{hypothèse} \\ \text{d'induction} \\ \text{«gauche»}}} \to_P e_3 \qquad\qquad e_1 \to_P \underbrace{e_2 \overset{*}{\to}_P e_3}_{\substack{\text{hypothèse} \\ \text{d'induction} \\ \text{«droite»}}}$$

Deux ensembles de dérivations sont donc définis à partir de Γ.

Définition 3.2 (Dérivations) *$\mathbb{D}_g$ et $\mathbb{D}_d$ sont définis inductivement par :*

$$\mathbb{D}_g ::= d_t^g(\Gamma) \mid d_c^g(\mathbb{D}_g, \Gamma) \qquad \mathbb{D}_d ::= d_t^d(\Gamma) \mid d_c^d(\Gamma, \mathbb{D}_d)$$

Nous caractérisons à présent, par un prédicat sur $\mathbb{D}_g$, les dérivations constituées de transitions «composables» et respectant une condition supplémentaire de renommage.

- Deux transitions $e_i^1 \overset{t_1}{\to}_{P_1} e_f^1$ et $e_i^2 \overset{t_2}{\to}_{P_2} e_f^2$ forment un couple de transitions composables (prédicat P_{TC}) si $P_1 = P_2$, $P^{\Gamma}(t_1)$, $P^{\Gamma}(t_2)$ et $e_f^1 = e_i^2$.

- D'autre part, en plus des conditions de renommage satisfaites par les transitions mises en jeu, nous imposons une condition supplémentaire (nécessaire, entre autres, à la preuve du lemme de commutation, voir l'hypothèse (3.3) présentée au début de ce chapitre). Lors de la dérivation :

$$\rho_0.R_0 \overset{n_0,r_0,C_0}{\rightarrow_P} \dots \overset{n_{k-1},r_{k-1},C_{k-1}}{\rightarrow_P} \rho_k.R_k$$

  chaque clause C_i $(0 \leqslant i \leqslant k-1)$ doit être renommée *en dehors* des variables apparaissant dans R_0 et dans les $r_j(C_j)$ $(0 \leqslant j < i)$.

Pour ce faire, une fonction ϑ, retournant la liste des variables de renommage utilisées lors d'une dérivation $d \in \mathbb{D}_g$ est définie récursivement par :

$$\vartheta(d) = \begin{cases} var(r(C)) & \text{si } d = d_t^g(\langle \rho_i.R_i, P, n, C, r, \theta, a, \rho_f.R_f \rangle) \\ \vartheta(d_0) \bowtie_v var(r(C)) & \text{si } d = d_c^g(d_0, \langle \rho_i.R_i, P, n, C, r, \theta, a, \rho_f.R_f \rangle) \end{cases}$$

où $\bowtie_v$ est une fonction de concaténation de deux listes de variables. Ces définitions permettent de définir récursivement, le prédicat $P^{\mathbb{D}}$ sur $\mathbb{D}_g$, caractérisant les dérivations correctes relativement aux propriétés décrites.

Définition 3.3 ($P^{\mathbb{D}}$) *Pour toute dérivation $d \in \mathbb{D}_g$:*

- *si $d = d_t^g(t)$, alors $P^{\Gamma}(t) \Rightarrow P^{\mathbb{D}}(d)$*
- *si $d = d_c^g(d_0, t_0)$, alors :*

$$\left(\begin{array}{c} P^{\mathbb{D}}(d_0) \wedge P_{TC}(t_d, t_0) \wedge \\ \forall x \in X\ ((x \in \vartheta(d_0)\ \vee\ x \in var(R)) \Rightarrow x \notin range(r)) \end{array} \right) \Rightarrow P^{\mathbb{D}}(d)$$

  où t_d désigne la dernière transition de d_0, R désigne la requête de l'état initial de d, et r désigne la substitution de renommage utilisée lors de la transition t_0.

La fonction τ_d^g permet de «transposer» ce prédicat sur l'ensemble $\mathbb{D}_d$: une dérivation $d \in \mathbb{D}_d$ satisfait le prédicat $P^{\mathbb{D}_d}$ si $P^{\mathbb{D}}(\tau_d^g(d))$ (on retrouve ici une démarche similaire à celle développée dans le chapitre 2).

Propriétés sur les variables

Les preuves des résultats, formalisées par la suite, prennent en compte de manière explicite la «partie renommage» des dérivations. Aussi, après avoir montré un lemme relatif à l'idempotence des subtitutions obtenues par composition de substitutions idempotentes, trois lemmes portant sur les variables mises en jeu lors d'une dérivation, et qui seront constamment utilisés par la suite, sont établis. s_{id} désigne la substitution identité `tv`.

Lemme 3.3 *Soient s et r deux substitutions idempotentes et l une liste de termes. Si r est couverte par $s(l)$, alors $\theta = \lambda x.r(s(x))$ est une substitution idempotente.*

<table>
<tr><td>$\mathbb{D}_g$</td><td>Inductive deriv:Set :=
  deriv_init : trans -> deriv |
  deriv_cons : deriv -> trans -> deriv.</td></tr>
<tr><td>$\mathbb{D}_d$</td><td>Inductive deriv_f : Set :=
  deriv_f_init : trans -> deriv_f |
  deriv_f_cons : trans -> deriv_f -> deriv_f.</td></tr>
<tr><td>P_{TC}</td><td>Inductive couple_trans_ok [t1:trans;t2:trans]:Prop:=
ctokinit:(p_trans t1)=(p_trans t2)->
         (state_end_t t1)=(state_init_t t2)->
         (RSL t1)->(RSL t2)->
         (couple_trans_ok t1 t2).</td></tr>
<tr><td>ϑ</td><td>Fixpoint list_var_c_d [d:deriv]:listv:=
<listv>Case d of
 [t0:trans]
   (var_cl (rsubst_cl (sr_trans t0) (c_trans t0)))
 [d1:deriv][t1:trans](Appv (list_var_c_d d1)
   (var_cl (rsubst_cl (sr_trans t1) (c_trans t1))))
       end.</td></tr>
<tr><td>$P^{\mathbb{D}}$</td><td>Fixpoint Deriv_ok [d:deriv]:Prop:=<Prop>Case d of
 [t1:trans](RSL t1)
 [d2:deriv][t2:trans]
 ((couple_trans_ok (end_d d2) t2) /\ (Deriv_ok d2) /\
  ((x:var)((IS_IN_LV x (list_var_c_d d2)) \/
    (IS_IN_LV x (var_req (Snd (state_init_d d2)))))
     ->~(rrange (sr_trans t2) x))) end.</td></tr>
</table>

TAB. 3.5 – *Dérivations*

PREUVE. D'après le lemme 2.4, il suffit de montrer que :

$$\forall x \in X \quad (x \in range(\theta) \Rightarrow \theta(x) = \mathsf{tv}(x))$$

Soit x une variable de l'image de θ. Par définition, il existe une variable $y \in dom(\theta)$ telle que $x \in var(\theta(y))$. Montrons que x n'appartient ni au domaine de r ni au domaine de s. Si $x \in dom(r)$, alors, puisque r est idempotente, d'après le lemme 2.4, on a $x \notin range(r)$. Deux cas sont alors possibles. Si $x \in var(s(y))$, alors x ne peut pas appartenir au domaine de r (puisque $x \in var(r(s(y)))$ et $x \notin range(r)$) ce qui est contradictoire. Sinon $x \notin var(s(y))$, et x n'apparaît pas dans $r(s(y))$ (puisque $x \notin range(r)$) ce qui est aussi contradictoire. On a donc bien $x \notin dom(r)$. Supposons maintenant que x appartienne au domaine de s. D'après le lemme 2.4, on a $x \notin range(s)$. Montrons tout d'abord que $x \notin range(r)$. Pour cela, supposons $x \in range(r)$. Puisque r est couverte par $s(l)$, x apparaît dans $s(l)$ et alors $x \in dom(s)$ implique $x \in range(r)$ ce qui est contradictoire. Aussi, de $x \in var(r(s(y)))$, on déduit $x \in var(s(y))$ ce qui permet d'établir $x \in range(s)$ (puisque x appartient au domaine de s) ce qui est encore contradictoire. x n'est donc ni dans le domaine de s, ni dans le domaine de r et on peut alors conclure à l'idempotence de θ. ◀

Lemme 3.4 *Si d: $s_{id}.R \xrightarrow{*}_P \sigma.R'$ satisfait $P^{\mathbb{ID}}$, alors σ est une substitution idempotente.*

PREUVE. Induction sur d.
• Pour $d = d_t^g(t)$: $s_{id}.R \overset{n,r,C}{\rightarrow}_P \sigma.R'$. Par définition, σ est une substitution idempotente.
• Pour $d = d_c^g(d_0, t_0)$: $s_{id}.R \xrightarrow{*}_P \theta_0.R_0 \overset{n,r,C}{\rightarrow}_P \theta\theta_0.R'$. Par hypothèse d'induction, θ_0 est une substitution idempotente. De plus, θ étant un unificateur principal de $R_{0_{/n}}$ et $r(C^+)$, θ est supportée et couverte par la liste l obtenue par concaténation des listes de termes présentes dans les deux atomes unifiés. D'après le lemme 3.2, $\theta_0 R_0 = R_0$ et puisque, par hypothèse, $P_{RS}(r, \theta_0)$, il vient $\theta_0 r(C^+) = r(C^+)$. Par conséquent, θ est aussi supportée et couverte par la liste $\theta_0(l)$. Le lemme 3.3 permet alors de conclure à l'idempotence de la substitution $\theta\theta_0$. ◀

Lemme 3.5 *Si d: $\mu.R \xrightarrow{*}_P \theta\mu.R'$ satisfait $P^{\mathbb{ID}}$, alors pour toute variable x, $x \in var(R') \Rightarrow (x \in var(R) \vee x \in \vartheta(d))$.*

PREUVE. Induction sur d.
• Pour $d = d_t^g(t)$: $\mu.R \overset{n,r,C}{\rightarrow}_P \theta\mu.\theta R[n \leftarrow r(C^-)]$. Si $x \in var(\theta R[n \leftarrow r(C^-)])$, deux cas se présentent. Si $x \in var(\theta R)$, alors soit $x \in var(R)$ et on peut conclure, soit $x \in range(\theta)$. Puisque θ est supportée et couverte par $R_{/n}$ et $r(C^+)$, soit $x \in var(R_{/n})$, soit $x \in var(r(C^+))$, et on peut conclure. Sinon, $x \in var(\theta r(C^-))$, et alors soit $x \in var(r(C^-))$, ce qui termine la preuve, soit $x \in range(\theta)$ et un raisonnement similaire au cas précédent permet de conclure.
• Pour $d_c^g(d_0, t_0)$: $\mu.R_0 \xrightarrow{*}_P \rho\mu.R \overset{n,r,C}{\rightarrow}_P \theta\rho\mu.\theta R[n \leftarrow r(C^-)]$. Soit x une variable apparaissant dans $\theta R[n \leftarrow r(C^-)]$, les deux cas possibles sont $x \in var(\theta R)$ et $x \in var(\theta r(C^-))$. Dans le premier cas, deux sous-cas se présentent. Si $x \in var(R)$, alors, l'hypothèse d'induction permet de conclure. Sinon, c'est que $x \in range(\theta)$, et alors, puisque θ est supportée et couverte par $R_{/n}$ et $r(C^+)$, soit $x \in var(R_{/n})$, soit $x \in var(r(C^+))$. Dans ces deux cas, par hypothèse d'induction, on peut conclure. Si $x \in var(\theta r(C^-))$, alors soit $x \in var(r(C^-))$, soit $x \in range(\theta)$ et un raisonnement similaire au cas précédent permet de conclure. ◀

Lemme 3.6 *Si d: $s_{id}.R \xrightarrow{*}_P \sigma.R'$ satisfait $P^{\mathbb{ID}}$, alors :*

$$\forall x \in X \quad (x \in dom(\sigma) \vee x \in range(\sigma)) \Rightarrow (x \in R \vee x \in \vartheta(d))$$

PREUVE. Induction sur d (preuve similaire à celle du lemme 3.5). ◀

3.2.3 Deux lemmes classiques

Nous présentons à présent une formalisation des preuves de deux résultats classiques en programmation logique : il s'agit des lemmes de généralisation et de commutation (l'énoncé de ces deux lemmes dans le système Coq est donné dans les tableaux 3.6 et 3.7).

Lemme de généralisation

Le lemme de généralisation que nous formalisons est légèrement différent de celui énoncé dans la discussion présentée au début de ce chapitre : il est plus général puisqu'il porte sur les dérivations finies quelconques et non nécessairement sur les réfutations. D'autre part, il met en jeu une liste de variables ℓ qui permet de contraindre le renommage des clauses lors de la dérivation. Ce lemme, qui sera utilisé durant la preuve du théorème de complétude de la SLD-résolution, s'énonce :

Lemme 3.7 (Lifting) *Soit* d: $s_{id}.\eta R \overset{*}{\rightarrow}_P \rho.R_1$ *une dérivation satisfaisant le prédicat* $P^{\mathbb{ID}}$. *Si pour une liste finie de variables* ℓ*, on a* $dom(\eta) \subseteq \ell$*, alors si* $\vartheta(d) \cap (var(R) \cup \ell) = \varnothing$*, il existe une dérivation vérifiant* $P^{\mathbb{ID}}$: $s_{id}.R \overset{*}{\rightarrow}_P \sigma.R_2$ *telle que* $\sigma \leqslant \rho\eta$ *et pour une requête* R_f *on a* $\rho\eta R_f = R_1$ *et* $\sigma R_f = R_2$.

Cet énoncé diffère de celui que l'on peut trouver dans la littérature, il semble plus technique. En effet, comme nous l'avons remarqué dans l'introduction, certaines précautions sont à prendre vis à vis des variables mises en jeu dans la dérivation initiale. Pour obtenir la version «classique» de ce lemme, il suffit d'instancier ℓ avec $var(R)$ et R_1 avec la requête vide. Comme beaucoup d'autres, sa preuve s'obtient par induction sur la dérivation en hypothèse du lemme. Les deux états dérivés sont explicitement reliés par la construction de la requête R_f.

Preuve. Induction sur d.

- Pour $s_{id}.\eta R \overset{n,r,C}{\rightarrow}_P \rho.\rho\eta \underbrace{R[n \leftarrow r(C^-)]}_{R_f}$.

Puisque, par définition, $\rho\eta R_{/n} = \rho r(C^+)$ et, par hypothèse, $\eta r(C) = r(C)$, il vient $\rho\eta R_{/n} = \rho\eta r(C^+)$. $R_{/n}$ et $r(C^+)$ sont donc unifiables et admettent, un unificateur principal σ vérifiant $\sigma \leqslant \rho\eta$. On peut alors facilement vérifier que la dérivation :

$$s_{id}.R \overset{n,r,C}{\rightarrow}_P \sigma.\sigma \underbrace{R[n \leftarrow r(C^-)]}_{R_f}$$

satisfait le prédicat $P^{\mathbb{ID}}$.

- Pour $\underbrace{s_{id}.\eta R_0 \overset{*}{\rightarrow}_P \mu.R_1}_{d_0} \overset{n,r,C}{\rightarrow}_P \theta\mu.\theta R_1[n \leftarrow r(C^-)]$.

Par hypothèse d'induction, il existe une substitution ρ, telle que $\rho \leqslant \mu\eta$, et une dérivation $s_{id}.R_0 \overset{*}{\rightarrow}_P \rho.R_2$ vérifiant $P^{\mathbb{ID}}$. De plus, pour une requête R', on a $\mu\eta R' = R_1$ et $\rho R' = R_2$. Puisque $\rho \leqslant \mu\eta$, il existe une substitution ε telle que $\varepsilon\rho = \mu\eta$. D'après le lemme 3.4, ρ est une substitution idempotente. Montrons tout d'abord que $\mu\eta r(C) = r(C)$ et $\rho r(C) = r(C)$.

- Si x est une variable du domaine de $\mu\eta$, alors soit $x \in dom(\mu)$, soit $x \in dom(\eta)$. Dans le premier cas, d'après le lemme 3.6, soit $x \in var(R_0)$, soit $x \in \vartheta(d_0)$, et par hypothèse, $x \notin var(r(C))$. Dans le second cas, par définition des conditions de renommage, $x \notin var(r(C))$. On a donc bien $\mu\eta r(C) = r(C)$.
- Si $\rho r(C) \neq r(C)$, c'est qu'il existe une variable $x \in dom(\rho)$ apparaissant dans $r(C)$. D'après le lemme 3.6, deux cas sont possibles. Soit, puisqu'on utilise exactement les mêmes variantes de clauses dans les deux dérivations, $x \in \vartheta(d_0)$ et on obtient une contradiction, soit $x \in var(R_0)$ ce qui contredit l'hypothèse du lemme.

Montrons à présent que les atomes $R_{2/n}$ et $r(C^+)$ sont unifiables. En effet, par définition, $\theta R_{1/n} = \theta r(C^+)$ et donc $\theta\mu\eta R'_{/n} = \theta r(C^+)$. De plus, puisque $\mu\eta r(C) = r(C)$, il vient $\theta\mu\eta R'_{/n} = \theta\mu\eta r(C^+)$ et donc $\theta\varepsilon\rho R'_{/n} = \theta\varepsilon\rho r(C^+)$. D'autre part, puisque $\rho r(C) = r(C)$, il vient $\theta\varepsilon\rho R'_{/n} = \theta\varepsilon r(C^+)$ ce qui permet de conclure à $\theta\varepsilon R_{2/n} = \theta\varepsilon r(C^+)$. Il existe donc un unificateur principal σ des atomes $R_{2/n}$ et $r(C^+)$, tel que $\sigma \leqslant \theta\varepsilon$. Ceci nous permet de montrer facilement que la dérivation :

$$s_{id}.R_0 \overset{*}{\rightarrow}_P \rho.R_2 \overset{n,r,C}{\rightarrow}_P \sigma\rho.\sigma R_2[n \leftarrow r(C^-)]$$

satisfait le prédicat $P^{\mathbb{ID}}$. Par ailleurs, on montre que :

$$\begin{array}{rcl} \sigma R_2[n \leftarrow r(C^-)] & = & \sigma\rho R'[n \leftarrow r(C^-)] \\ \theta R_1[n \leftarrow r(C^-)] & = & \theta\mu\eta R'[n \leftarrow r(C^-)] \end{array}$$

Enfin, puisque $\sigma \leqslant \theta\varepsilon$, il existe une substitution ν telle que $\nu\sigma = \theta\varepsilon$ et il vient alors $\theta\mu\eta = \theta\varepsilon\rho = \nu\sigma\rho$, ce qui permet de conclure à $\sigma\rho \leqslant \theta\mu\eta$. ◀

Lemme de commutation

A chaque étape de résolution, un atome est selectionné. Le lemme de commutation assure que ce choix relève d'un non-déterminisme «par indifférence». Signalons que cette propriété est plus faible que la propriété de confluence. Par exemple, à partir du programme :

$$P = \{p(a,a) \leftarrow ;\ p(b,b) \leftarrow\}$$

on peut construire les dérivations :

$$s_{id}.p(a,x), p(b,x) \rightarrow \{x/a\}.p(b,a) \quad \text{et} \quad s_{id}.p(a,x), p(b,x) \rightarrow \{x/b\}.p(a,b)$$

```
Lemma llifting:(d:deriv)(r:request)(eta:subst)(l:listv)
  (Deriv_ok d)
->(Fst (state_init_d d))=tv
->(Snd (state_init_d d))=(subst_req eta r)
->((x:var)~(IS_IN_LV x l)->~(dom_t eta x))
->((t:trans)(IS_IN_D t d)->
      ((x:var)(IS_IN_LV x (Appv l (var_req r)))
                  ->~(rrange (sr_trans t) x)))
->(Ex [d0:deriv](
     ((t1,t2:trans)(IS_IN_D t1 d)->(IS_IN_D t2 d0)->
                     (p_trans t1)=(p_trans t2))
  /\ (Deriv_ok d0)
  /\ (list_var_c_d d0)=(list_var_c_d d)
  /\ (Fst (state_init_d d0))=tv
  /\ (Snd (state_init_d d0))=r
  /\ (Ex [rf:request](
         (Snd (state_end_d d0))=
         (subst_req (Fst (state_end_d d0)) rf)
      /\ (Snd (state_end_d d))=
         (subst_req (Fst (state_end_d d))
                    (subst_req eta rf))))
      /\ (less_subst_t (Fst (state_end_d d0))
                       ([x:var](Subst_t
                          (Fst (state_end_d d))
                          (eta x)))))).
```

TAB. 3.6 – *Lemme de généralisation*

qui ne «confluent» pas.

Lemme 3.8 (Switching) *Si, lors d'une dérivation, deux atomes sont sélectionnés successivement, alors ils peuvent être sélectionnés dans l'ordre inverse, les états dérivés ne diffèrent que par renommage des variables.*

Par souci de lisibilité, nous n'avons pas donné un énoncé formalisé de ce lemme (l'énoncé formel se trouve dans le tableau 3.7). Par contre, nous présentons, de manière (très) *détaillée*, sa preuve en suivant exactement la preuve formelle développée dans le système COQ. Cette construction illustre bien le rôle primordial des hypothèses de séparation des variables dans une dérivation. En effet, cette preuve ne s'obtient pas par induction mais repose exclusivement sur les conditions de renommage qui sont constamment exploitées. Bien sûr, à la fin de la démonstration, les deux états dérivés sont explicitement mis en relation par la construction d'une substitution. Signalons que nous supposons ici que le premier atome sélectionné se trouve avant le second dans la requête initiale. Nous commençons par prouver certaines propriétés sur les objets présents dans la dérivation en hypothèse du lemme et afin de disposer de ces propriétés durant toute la preuve, nous utilisons pour cela la tactique `Cut` du système Coq, présentée en (1.4). Ces «Cut» permettent d'utiliser des propriétés durant la preuve sans les redémontrer à chaque fois tout en partageant un même contexte d'hypothèses (tout comme un programme «efficace» n'effectue pas les mêmes calculs plusieurs fois). Ils supposent vérifiées les hypothèses du lemme.

Preuve «quasi-formelle» du lemme de commutation.

Transitions en hypothèse La dérivation en hypothèse du lemme peut s'écrire :

$$d_c^g(d_c^g(d,t_1),t_2) : \underbrace{s_{id}.R_0 \xrightarrow{*}_P \rho.R}_{d} \quad \overset{t_1}{\xrightarrow{n_1, r_1, C_1}}_P \quad \theta\rho.\theta R_1 \quad \overset{t_2}{\xrightarrow{n_2, r_2, C_2}}_P \quad \sigma\theta\rho.\sigma R_2$$

avec $\theta R_1 = \theta R[n_1 \leftarrow r_1(C_1^-)]$ et $R_2 = (\theta R_1)[n_2 \leftarrow r_2(C_2^-)]$.
De plus, on suppose que l'atome sélectionné lors de la transition t_1 se trouve avant l'atome sélectionné lors de la transtion t_2 dans la requête R, ce qui se traduit par l'hypothèse :

$$n_1 + \lg(C_1^-) \leqslant n_2 \tag{3.5}$$

Les «Cut» Nous montrons tout d'abord les propriétés suivantes.

Cut 1 $\rho r_2 C_2 = r_2 C_2$

Preuve. Soit $z \in dom(\rho)$, d'après le lemme 3.6, soit $z \in var(R_0)$, soit $z \in \vartheta(d)$. Dans les deux cas, du fait des conditions de renommage, z n'apparaît pas dans r_2C_2, ce qui permet de conclure. ◀

Cut 2 *$\sigma\theta$ est un unificateur de $R_{/n_2-\lg(C_1^-)+1}$ et $r_2C_2^+$ et il existe un unificateur principal μ de ces deux atomes tel que $\mu \leqslant \sigma\theta$ (i.e. il existe une substitution ε telle que $\varepsilon\mu = \sigma\theta$).*

Preuve. Par définition (transition t_2), on a $\sigma\theta R_{1\,/n_2} = \sigma r_2(C_2^+)$. Puisque $P^\Gamma(t_2)$, on a $\theta\rho r_2(C_2) = r_2(C_2)$ et il vient $\sigma\theta R_{1\,/n_2} = \sigma\theta\rho r_2(C_2^+)$. D'après le Cut 1, on obtient $\sigma\theta R_{1\,/n_2} = \sigma\theta r_2(C_2^+)$. Puisque $\theta R_1 = \theta R[n_1 \leftarrow r(C_1^-)]$ et $n_1 + \lg(C_1^-) \leqslant n_2$, on a alors $\theta R_{1\,/n_2} = \theta R_{/n_2-\lg(C_1^-)+1}$ [1], ce qui permet d'établir $\sigma\theta R_{/n_2-\lg(C_1^-)+1} = \sigma\theta r_2 C_2^+$. ◀

Cut 3 $\mu r_1 C_1 = r_1 C_1$

1. Nous omettons les preuves (assez techniques) de ce type d'égalités qui s'obtiennent par induction.

PREUVE. Montrons que si $z \in r_1(C_1)$, alors z n'appartient pas au domaine de μ. D'après le Cut 2, μ est «supportée et couverte» par $R_{/n_2-\lg(C_1^-)+1}$ et $r_2(C_2^+)$. z apparaît donc dans l'un de ces deux atomes. Si $z \in var(R)$, alors (renommage sur t_1) $z \notin var(r_1(C_1))$ ce qui est contradictoire. Sinon, $z \in var(r_2(C_2^+))$, et alors z appartient à l'image de r_2 et ne peut donc appartenir à $r_1(C_1)$. ◀

Cut 4 *ε est un unificateur de $r_1(C_1^+)$ et $(\mu R[n_2 - \lg(C_1^-) + 1 \leftarrow r_2(C_2^-)])_{/n_1}$ qui admettent alors un unificateur principal ν tel que $\nu \leqslant \varepsilon$ (i.e. il existe une substitution ξ telle que $\xi\nu = \varepsilon$).*

PREUVE. Puisque :

$$\begin{array}{rcll} \theta R_{/n_1} & = & \theta r_1(C_1^+) & \text{(transition } t_1) \\ R_{/n_1} & = & (R[n_2 - \lg(C_1^-) + 1 \leftarrow r_2(C_2^-)])_{/n_1} & (3.5) \end{array}$$

il vient :

$$\begin{array}{rcll} \theta(R[n_2 - \lg(C_1^-) + 1 \leftarrow r_2(C_2^-)])_{/n_1} & = & \theta r_1(C_1^+) & \\ \sigma\theta(R[n_2 - \lg(C_1^-) + 1 \leftarrow r_2(C_2^-)])_{/n_1} & = & \sigma\theta r_1(C_1^+) & \\ \varepsilon\mu(R[n_2 - \lg(C_1^-) + 1 \leftarrow r_2(C_2^-)])_{/n_1} & = & \varepsilon\mu r_1(C_1^+) & \text{(Cut 2)} \\ \varepsilon\mu(R[n_2 - \lg(C_1^-) + 1 \leftarrow r_2(C_2^-)])_{/n_1} & = & \varepsilon r_1(C_1^+) & \text{(Cut 3)} \end{array}$$

ce qui permet de conclure. ◀

Construction de la dérivation finale Le Cut 2 permet de construire la transition t_3, puisque μ est un unificateur principal de $R_{/n_2-\lg(C_1^-)+1}$ et $r_2(C_2^+)$. De la même manière, le Cut 4 permet de construire la transition t_4, puisque ν est un unificateur principal de $(\mu R[n_2 - \lg(C_1^-) + 1 \leftarrow r_2(C_2^-)])_{/n_1}$ et $r_1(C_1^+)$:

$$\rho.R \quad \overset{t_3}{\overset{n_2 - \lg(C_1^-)+1, r_2, C_2}{\longrightarrow_P}} \quad \mu\rho.\mu R[n_2 - \lg(C_1^-) + 1 \leftarrow r_2(C_2^-)]$$

$$\mu\rho.\underbrace{\mu R[n_2 - \lg(C_1^-) + 1 \leftarrow r_2(C_2^-)]}_{R'} \quad \overset{t_4}{\overset{n_1, r_1, C_1}{\longrightarrow_P}} \quad \nu\mu\rho.\nu R'[n_1 \leftarrow r_1(C_1^-)]$$

Montrons que les transitions t_3 et t_4 satisfont le prédicat P^Γ.

- Condition provenant du prédicat P_{RS}
  - transition t_3 : Conséquence immédiate du Cut 1
  - transition t_4 : D'après le Cut 3, on a $\mu r_1 C_1 = r_1 C_1$ et le Cut 1 permet d'établir $\rho r_1 C_1 = r_1 C_1$. On a alors bien $\mu\rho r_1 C_1 = \mu r_1 C_1 = r_1 C_1$.

- Condition provenant du prédicat P_{RR}
  - transition t_3 : Si $x \in var(R)$, alors d'après le lemme 3.5, soit $x \in var(R_0)$, soit $x \in \vartheta(d)$. Dans les deux cas, x ne peut pas apparaître dans r_2C_2.
  - transition t_4 : Soit $x \in var(\mu R[n_2 - \lg(C_1^-) + 1 \leftarrow r_2(C_2^-)])$. Deux cas sont possibles : soit $x \in var(R[n_2 - \lg(C_1^-) + 1 \leftarrow r_2(C_2^-)])$, soit $x \in range(\mu)$. Dans les deux cas, par définition et d'après le Cut 2, soit $x \in var(R)$, soit $x \in var(r_2C_2)$. Dans le premier cas, le lemme 3.5, permet d'établir $x \in var(R_0) \vee x \in \vartheta(d)$ et dans tous les cas, x ne peut pas apparaître dans r_1C_1.

Lien entre les deux états dérivés Nous cherchons à montrer qu'il existe une substitution (assimilable à une substitution de renommage) r telle que :

$$r(\sigma R_2) = \nu R'[n_1 \leftarrow r_1(C_1^-)]$$

Pour cela, nous montrons tout d'abord les «Cut» suivants :

Cut 5 $\nu\mu \leqslant \sigma\theta$

PREUVE. D'après les Cut 2 et 4, $\varepsilon\mu = \sigma\theta$ et $\xi\nu = \varepsilon$. Il vient alors $\xi\nu\mu = \sigma\theta$, ce qui permet de conclure. ◀

Cut 6 $\theta \leqslant \nu\mu$ *(i.e. il existe une substitution ζ telle que $\zeta\theta = \nu\mu$).*

PREUVE. On a :

$$\begin{array}{ll} \nu\mu R[n_2 - \lg(C_1^-) + 1 \leftarrow r_2(C_2^-)]_{/n_1} = \nu r_1(C_1^+) & \text{(Cut 4)} \\ \nu\mu R[n_2 - \lg(C_1^-) + 1 \leftarrow r_2(C_2^-)]_{/n_1} = \nu\mu r_1(C_1^+) & \text{(Cut 3)} \\ R_{/n_1} = (R[n_2 - \lg(C_1^-) + 1 \leftarrow r_2(C_2^-)])_{/n_1} & (3.5) \end{array}$$

$\nu\mu$ est donc un unificateur de $R_{/n_1}$ et $r_1(C_1^+)$. Or, par hypothèse, θ est un unificateur principal de ces deux atomes (transition t_1) et donc $\theta \leqslant \nu\mu$. ◀

Cut 7 $\theta r_2C_2 = r_2C_2$

PREUVE. Montrons que si $x \in var(r_2(C_2))$, alors x n'appartient pas au domaine de θ. Par hypothèse, θ est supportée et couverte par $R_{/n_1}$ et $r_1(C_1^+)$, x est donc présent dans un de ces deux atomes. Si $x \in var(R_{/n_1})$, alors, d'après le lemme 3.5, soit $x \in var(R_0)$, soit $x \in \vartheta(d)$. Dans ces deux cas, x n'appartient pas à l'image de r_2 (i.e. $x \notin var(r_2(C_2))$). D'autre part, si $x \in var(r_1(C_1^+))$, alors $x \notin var(r_2(C_2))$. ◀

Cut 8 $\sigma \leqslant \zeta$ *(i.e. il existe une substitution δ telle que $\delta\sigma = \zeta$).*

PREUVE. On a :

$$\begin{array}{rcll} \mu R_{/n_2-\lg(C_1^-)+1} & = & \mu r_2(C_2^+) & (P^\Gamma(t_3)) \\ \nu\mu R_{/n_2-\lg(C_1^-)+1} & = & \nu\mu r_2(C_2^+) & \\ \zeta\theta R_{/n_2-\lg(C_1^-)+1} & = & \zeta\theta r_2(C_2^+) & (\text{Cut } 6) \\ \zeta\theta R_{/n_2-\lg(C_1^-)+1} & = & \zeta r_2(C_2^+) & (\text{Cut } 7) \end{array}$$

ζ unifie donc $\theta R_{/n_2-\lg(C_1^-)+1}$ et $r_2(C_2^+)$. D'après (3.5), on obtient alors :

$$\theta R_{/n_2-\lg(C_1^-)+1} = \theta R[n_1 \leftarrow r_1 C_1^-]_{n_2} = \theta R_{1_{/n_2}}$$

Or, par hypothèse (transition t_2), σ est un unificateur principal de ces deux atomes et donc $\sigma \leqslant \zeta$. ◀

Cut 9 $\sigma\theta \leqslant \nu\mu$

PREUVE. D'après le Cut 6, $\nu\mu = \zeta\theta$. Le Cut 8 permet alors d'établir $\nu\mu = \delta\sigma\theta$ ce qui permet de conclure. ◀

Les Cut 5 et 9 permettent de montrer que les substitutions $\nu\mu$ et $\sigma\theta$ sont $\leqslant$-équivalentes. Nous pouvons donc établir le lien entre les deux états dérivés. Tout d'abord, nous avons :

$$\begin{array}{rll} \sigma R_2 & = \sigma(\theta R_1)[n_2 \leftarrow r_2 C_2^-] & \\ & = \sigma\theta R_1[n_2 \leftarrow r_2 C_2^-] & (\text{Cut } 7) \\ & = \sigma(\theta R[n_1 \leftarrow r_1 C_1^-])[n_2 \leftarrow r_2 C_2^-] & \\ & = \sigma\theta(R[n_1 \leftarrow r_1 C_1^-])[n_2 \leftarrow \theta r_2 C_2^-] & \\ & = \sigma\theta(R[n_1 \leftarrow r_1 C_1^-])[n_2 \leftarrow r_2 C_2^-] & (\text{Cut } 7) \end{array}$$

D'autre part :

$$\begin{array}{rll} \nu R'[n_1 \leftarrow r_1 C_1^-] & = \nu(\mu R[n_2 + \lg(C_1^-) + 1 \leftarrow r_2 C_2^-])[n_1 \leftarrow r_1 C_1^-] & \\ & = \nu\mu R[n_2 + \lg(C_1^-) + 1 \leftarrow r_2 C_2^-][n_1 \leftarrow r_1 C_1^-] & (\text{Cut } 3) \\ & = \nu\mu(R[n_1 \leftarrow r_1 C_1^-])[n_2 \leftarrow r_2 C_2^-] & \end{array}$$

Le lemme 2.5 permet alors de construire une substitution (assimilable à une substitution de renommage) r, ce qui permet de conclure et termine la preuve du lemme de commutation.

La difficulté majeure du codage des preuves des lemmes de commutation et de généralisation provient essentiellement des hypothèses de renommage. En effet, ces lemmes sont généralement prouvés («à la main») en posant les hypothèses de séparation des variables au fur et à mesure que l'on construit les objets de la preuve (un même symbole désignant parfois durant la preuve différentes variantes d'une clause ne satisfaisant pas les mêmes propriétés de renommage). Leur formalisation montre bien à quel point le renommage joue un rôle important. Malgré tout, ces preuves se prêtent assez bien au codage dans le calcul des constructions inductives.

```
Lemma switching : (d:deriv)(t1:trans)(t2:trans)
  (Deriv_ok (deriv_cons (deriv_cons d t1) t2))
->(le (plus (n_trans t1) (Length_r (body_c (c_trans t1))))
      (n_trans t2))
->(Fst (state_init_d d))=tv
->(Ex [t3:trans] (Ex [t4:trans]
  ((Deriv_ok (deriv_cons (deriv_cons d t3) t4))
/\(n_trans t3)=(S (minus (n_trans t2)
                    (Length_r (body_c (c_trans t1)))))
/\(n_trans t4)=(n_trans t1)
/\(Ex [r:subst]
  ((Snd (state_end_d (deriv_cons (deriv_cons d t1) t2)))=
  (subst_req r
    (Snd (state_end_d (deriv_cons (deriv_cons d t3) t4))))
  /\((x:var) (IS_IN_LV x
            (var_req
   (Snd (state_end_d (deriv_cons (deriv_cons d t3) t4)))))
          -> (Ex [v:var] (r x)=(tv v)))))))).
```

TAB. 3.7 – *Lemme de commutation*

3.3 Aspects sémantiques

Nous présentons à présent une formalisation des aspects sémantiques de la SLD-résolution. Les schémas d'interprétation des objets syntaxiques, définis dans les paragraphes précédents, sont introduits et permettent la définition d'une sémantique déclarative des programmes définis. Les théorèmes de validité et de complétude de la SLD-résolution sont alors formalisés, reliant ainsi les deux aspects sémantiques (déclaratif et opérationnel) de la programmation logique. Nous utiliserons, durant ce développement, quelques résultats généraux sur les points fixes des applications monotones et/ou continues, rappelés dans le chapitre 1 et dont la formalisation est présentée dans l'annexe A.

3.3.1 Interprétations et modèles

Interprétations d'une signature

Nous présentons ici une formalisation de la notion d'interprétation des symboles de Σ et des symboles de Π. La définition des objets introduits dans ce paragraphe dans le système COQ est donnée dans le tableau 3.8.

Signature fonctionnelle Afin de pouvoir donner un schéma d'interprétation des termes, il est nécessaire d'«attacher un sens» aux symboles utilisés dans la construction des termes. Une interprétation, I, de la signature fonctionnelle Σ, est la donnée d'un ensemble non vide D, appelé le domaine de I, et, pour chaque $f \in \Sigma$, d'arité n, d'une application $f^I : D^n \to D$ (en particulier, chaque constante k, est interprétée par un élément k^I de D). Soit $\mathbb{L}_n[A]$ le type dépendant des listes de longueur n constituées à partir d'éléments d'un ensemble A, obtenu à partir du type `LIST` présenté en (1.10), et défini inductivement par :

- $||_0^A$ est une liste (c'est la liste vide de longueur nulle).
- si $a \in A$ et si l est une liste d'éléments de A de longueur n, alors $|a, l|_{n+1}^A$ est une liste d'éléments de A de longueur $n+1$.

Etant donné un entier n, $\mathbb{F}_n[A]$ est défini comme la collection des applications de $\mathbb{L}_n[A]$ vers A. Le type des interprétations, de domaine D, de la signature Σ, est alors défini par le type dépendant :

$$I_\Sigma[D] = \prod_{f \in \Sigma} \mathbb{F}_{ar(f)}[D]$$

Etant donné un symbole de fonction $f \in \Sigma$ et une interprétation I de domaine D, l'application f^I correspond donc à $I(f)$ qui définit une fonction de $\mathbb{L}_{ar(f)}[D]$ dans D.

Schéma d'interprétation des termes Il ne reste plus qu'à indiquer comment interpréter les symboles de variables pour pouvoir donner le schéma d'interprétation des termes (et des listes de termes) : les variables présentes dans un terme sont interprétées comme variant dans le domaine de l'interprétation de Σ considérée. L'ensemble A^X, des valuations sur A, est donc défini comme l'ensemble des applications de X vers A. Les schémas d'interprétation des termes et des listes de termes peuvent être à présent définis par les deux fonctions mutuellement récursives suivantes. Soit I une interprétation de Σ de domaine D (i.e. un élément de $I_\Sigma[D]$) :

- un terme t est interprété par (ou dénote) une application $t^I : D^X \to D$ définie par :

$$\forall v \in D^X \quad t^I(v) = \begin{cases} v(x) & \text{si } t = \mathsf{tv}(x) \\ f^I(l^I(v)) & \text{si } t = \mathsf{tf}(f, l) \end{cases}$$

- une liste de termes l de longueur n est interprétée par (ou dénote) une application $l^I : D^X \to \mathbb{L}_n[D]$ définie par :

$$\forall v \in D^X \quad l^I(v) = \begin{cases} ||_0^D & \text{si } n = 0 \\ |t_0^I(v), l_0^I(v)|_{n_0+1}^D & \text{si } l = \langle t_0, l_0 \rangle_{n_0+1} \end{cases}$$

Signature relationnelle Une interprétation, I, de la signature relationnelle Π, est la donnée d'un ensemble non vide D, appelé le domaine de I, et, pour chaque symbole $p \in \Pi$, d'arité n, d'une application $p^I : D^n \to \mathbb{B}$ où $\mathbb{B}$ désigne l'ensemble (défini inductivement) des booléens ($\{\mathsf{true}, \mathsf{false}\}$). $\mathbb{B}$ est de type Set et est muni des opérateurs classiques, que l'on notera $\neg_b$, $\wedge_b$ et $\vee_b$ afin de les distinguer des opérateurs utilisés sur les propositions (Prop) que nous manipulons ($\neg$, $\wedge$, $\vee$). Nous procédons de la même manière que pour Σ pour définir formellement les interprétations de la signature Π. Tout d'abord, étant donné un entier n, $\mathbb{P}_n[A]$ désigne la collection des applications de $\mathbb{L}_n[A]$ vers $\mathbb{B}$. Le type des interprétations, de domaine D, de la signature Π, est alors défini par le type dépendant :

$$I_\Pi[D] = \prod_{p \in \Pi} \mathbb{P}_{ar(p)}[D]$$

Etant donné un symbole de prédicat $p \in \Pi$ et une interprétation I de domaine D, l'application p^I correspond donc à $I(p)$ qui définit une fonction de $\mathbb{L}_{ar(f)}[D]$ dans $\mathbb{B}$.

Interprétations des clauses de Horn

Nous pouvons à présent définir les schémas d'interprétation des objets utilisés en programmation logique (atomes, requêtes, clauses, ...). Le tableau 3.9 contient les définitions qui suivent dans le système Coq. Afin d'alléger les notations, I désignera dans ce qui suit une interprétation de $\Sigma \cup \Pi$ de domaine D.

Atomes Un atome $a = \mathsf{pl}(p, l)$ est interprété par (ou dénote) une application $a^I : D^X \to \mathbb{B}$ défine par : $\forall v \in D^X \quad a^I(v) = p^I(l^I(v))$.

Requêtes Deux schémas d'interprétation sont définis pour les requêtes : une requête r est interprétée par une application :

- $r^I : D^X \to \mathbb{B}$ définie par :

$$\forall v \in D^X \quad r^I(v) = \begin{cases} \mathsf{false} & \text{si } r = r_\varnothing \\ (\neg_b a_0^I(v)) \vee_b r_0^I(v) & \text{si } r = c_r(a_0, r_0) \end{cases}$$

- $\tilde{r}^I : D^X \to \mathbb{B}$ définie par

$$\forall v \in D^X \quad \tilde{r}^I(v) = \begin{cases} \mathsf{true} & \text{si } r = r_\varnothing \\ a_0^I(v) \wedge_b \widetilde{r_0}^I(v) & \text{si } r = c_r(a_0, r_0) \end{cases}$$

Tandis que le premier schéma d'interprétation donné correspond à l'interprétation d'une «requête» $a_1, \cdots, a_q$ vue comme un but (i.e. une clause négative) $\forall \vec{x}(\neg_b a_1 \vee_b \cdots \vee_b \neg_b a_q)$, le second ne correspond pas à l'interprétation d'une

<table>
<tr><td>$\mathbb{F}_n[A]$</td><td><pre>Definition f_n:Set->nat->Set:=
              [D:Set][n:nat](LIST D n)->D.</pre></td></tr>
<tr><td>$I_\Sigma[D]$</td><td><pre>Definition f_i:Set->Set:=
              [D:Set](f:fun)(f_n D (arity ar f)).</pre></td></tr>
<tr><td>A^X</td><td><pre>Definition valuation:Set->Set:=[A:Set](var->A).</pre></td></tr>
<tr><td>t^I et l^I</td><td><pre>Fixpoint Interp_t [D:Set;I:(f_i D);t:Term]:
 (valuation D)->D:=
 [v:(valuation D)]<D>Case t of
   [x:var](v x)
   [f:fun][l:(list_term (arity ar f))]
          ((I f) (apply_l D I (arity ar f) l v))
                    end
with apply_l [D:Set;I:(f_i D);n:nat;l:(list_term n)]:
             (valuation D)->(LIST D n):=
 [v:(valuation D)]<[n:nat](LIST D n)>Case l of
  (NIL D)
  [n0:nat][t0:Term][l0:(list_term n0)]
  (CONS D n0 (Interp_t D I t0 v) (apply_l D I n0 l0 v))
                    end.</pre></td></tr>
<tr><td>$\mathbb{B}$</td><td><pre>Inductive bool:Set:=true:bool | false:bool.</pre></td></tr>
<tr><td>$\mathbb{P}_n[D]$</td><td><pre>Definition p_n:Set->nat->Set:=
              [D:Set][n:nat](LIST D n)->bool.</pre></td></tr>
<tr><td>$I_\Pi[D]$</td><td><pre>Definition p_i:Set->Set:=
              [D:Set](p:predic)(p_n D (arity_p p)).</pre></td></tr>
</table>

TAB. 3.8 – *Interprétations d'une signature*

requête par la formule existentielle $\exists\vec{x}(a_1 \wedge_b \cdots \wedge_b a_q)$. En effet, nous caractériserons une substitution solution θ d'une requête $a_1, \cdots, a_q$, à partir d'un programme défini P, par $P \models \forall\vec{x}\theta(a_1 \wedge_b \cdots \wedge_b a_q)$ ce qui correspond au deuxième schéma d'interprétation.

Lemme 3.9 *Soit I une interprétation de $\Sigma \cup \Pi$ de domaine D et v une valuation sur D. Pour toute requête R, si $R^I(v) = \mathsf{true}$, alors $\tilde{R}^I(v) = \mathsf{false}$.*

PREUVE. Induction sur R. ◀

Clauses définies Une clause $c = \langle a, r\rangle \in C_{\Sigma,\Pi}[X]$ est interprétée par une application $c^I : D^X \to \mathbb{B}$ définie par : $\forall v \in D^X \quad c^I(v) = a^I(v) \vee_b r^I(v)$.

Programmes définis Un programme $P \in P_{\Sigma,\Pi}[X]$ est interprété par une application $P^I : D^X \to \mathbb{B}$ définie par :

$$\forall v \in D^X \quad P^I(v) = \begin{cases} \mathsf{true} & \text{si } P = P_\varnothing \\ c^I(v) \wedge_b P_0^I(v) & \text{si } P = c_p(c, P_0) \end{cases}$$

Clauses de Horn Un «ensemble» de clauses de Horn $h \in H_{\Sigma,\Pi}[X]$ est interprété par une application $h^I : D^X \to \mathbb{B}$ définie par :

$$\forall v \in D^X \quad h^I(v) = \begin{cases} P^I(v) & \text{si } h = h_p(P) \\ r^I(v) \wedge_b h_0^I(v) & \text{si } h = h_r(r, h_0) \end{cases}$$

Modèles

Nous définissons à présent les notions de modèle et de conséquence sémantique (introduites dans le système Coq par les définitions du tableau 3.9). Une interprétation I de $\Sigma \cup \Pi$ de domaine D est un modèle d'un objet o (atome, requête, clause, ...), ce que l'on notera $\models_I o$, si et seulement si pour toute valuation v sur D, $o^I(v) = \mathsf{true}$.

$$\models_I o \Leftrightarrow (\forall v \in D^X \quad o^I(v) = \mathsf{true})$$

Par abus de notation, nous écrirons simplement $\models_I r$ si $\forall v \in D^X\ r^I(v) = \mathsf{true}$ et $\models_I \tilde{r}$ si $\forall v \in D^X\ \tilde{r}^I(v) = \mathsf{true}$. Une requête R est conséquence sémantique d'un ensemble de clauses de Horn H, ce qui sera noté $H \models R$, si et seulement si tout modèle I de H vérifie $\models_I \tilde{R}$.

$$H \models R \Leftrightarrow \forall I \in I_{\Sigma,\Pi}[D] \quad (\models_I H \Rightarrow \models_I \tilde{R})$$

Signalons qu'étant donnés une interprétation I de $\Sigma \cup \Pi$ de domaine D et un objet o (atome, requête, clause, ...), pour toute valuation v sur D, la proposition $o^I(v) = \mathsf{true}$ est toujours décidable, tandis que la proposition $\models_I o$ ne l'est pas forcément (c'est le cas si D est infini). Certaines des difficultés rencontrées plus loin proviennent, en partie, de cette remarque. Enfin, puisque valuations et substitutions sont des applications sur X, nous avons le lemme suivant.

Lemme 3.10 *Soit I une interprétation de $\Sigma \cup \Pi$ de domaine D et o un objet (atome, requête, clause, ...). Si $\models_I o$, alors, pour toute substitution θ, on a $\models_I \theta(o)$.*

Preuve. On montre, par induction, que pour tout terme t, on a :

$$\forall v \in D^X \quad (\theta t)^I(v) = t^I(\lambda x.((\theta x)^I(v)))$$

Schéma général d'interprétation de $o : O$	

`Definition Interp_`O`:`
`(D:Set)(f_i D)->(p_i D)->`O`-> (valuation D)->bool:=`
`[D:Set][If:(f_i D)][Ip:(p_i D)][`$o : O$` ] [v:(valuation D)]`
`<bool> Case `o` of `I` end.`	

O	I
`Interp_p atom`	`[p0:predic][l0:(list_term (arity_p p0))]` `((Ip p0) (apply_l D If (arity_p p0) l0 v))`

`Interp_r request`	`false`
	`[l0:atom][r0:request]`
	`(orb (neg (Interp_p D If Ip l0 v))`
	`(Interp_r D If Ip r0 v))`

`Interp_rn request`	`true`
	`[l0:atom][r0:request]`
	`(andb (Interp_p D If Ip l0 v))`
	`(Interp_rn D If Ip r0 v))`

`Interp_c clause`	`[a:atom][r:request]`
	`(orb (Interp_r D If Ip r v)`
	`(Interp_p D If Ip a v))`

`Interp_P program`	`true`
	`[c0:clause][p0:program]`
	`(andb (Interp_c D If Ip c0 v)`
	`(Interp_P D If Ip p0 v))`

`Interp_h horn`	`[p:program](Interp_P D If Ip p v)`
	`[r0:request][h0:horn]`
	`(andb (Interp_r D If Ip r0 v)`
	`(Interp_h D If Ip h0 v))`

$\models_I$	`Definition `O`_valid:(D:Set)(f_i D)->(p_i D)->`O`->Prop:=`
	`[D:Set][If:(f_i D)][Ip:(p_i D)][`$o : O$`]`
	`((v:(valuation D))(Interp_`O` D If Ip l v)=true).`

$\models$	`Definition semantic_csq : horn -> request -> Prop :=`
	`[h:horn][r:request]`
	`( (D:Set)(If:(f_i D))(Ip:(p_i D))`
	`(h_valid D If Ip h)->(reqn_valid D If Ip r) ).`

TAB. 3.9 – *Interprétations et Modèles*

Par hypothèse, $\models_I o$, et donc $o^I(v) = \mathsf{true}$ pour toute valuation v sur D. Il vient alors :

$$\forall v \in D^X \quad (\theta o)^I(v) = o^I(\lambda x.((\theta x)^I(v))) = \mathsf{true}$$

ce qui permet de conclure. ◀

3.3.2 Validité de la SLD-Résolution

Solutions et réponses

Tandis que la sémantique opérationnelle des programmes définis, que nous avons formalisée dans la première partie de ce chapitre, détermine comment un calcul est effectué, la sémantique déclarative des programmes définis décrit ce qui peut être calculé. Ces deux lectures d'un programme défini sont associées à la notion de réponse et de solution d'une requête à partir d'un programme.

Définition 3.4 (Solutions) *Les solutions d'une requête R, relativement à un programme défini P, sont les substitutions θ telles que $h_p(P) \models \theta R$.*

Cette définition établit une description déclarative du résultat que l'on attend d'un programme à partir d'une requête. La contre-partie opérationnelle des solutions est la notion de réponse (voir tableau 3.10).

Définition 3.5 (Réponses) *Si $s_{id}.R \stackrel{*}{\rightarrow}_P \rho.r_\varnothing$ est une dérivation satisfaisant le prédicat $P^{\mathbb{D}}$, alors ρ est une réponse pour R, relativement au programme défini P.*

Théorème de validité

Il s'agit de prouver que si une requête admet une réponse à partir d'un programme défini, alors elle admet une solution. Plus précisément, nous montrons que toute réponse est solution. La démonstration de ce théorème est classique et requiert la notion de transition «libre» (correspondant à la notion de dérivation «sans restriction» dans [67]) : une transition libre est une transition satisfaisant la règle de résolution, mais où l'unificateur utilisé n'est pas forcément un unificateur principal (la condition $\rho R = R$ sur les états de résolution $\rho.R$ n'est alors plus vérifiée). Cette preuve est obtenue en appliquant un schéma d'induction «droit» sur la dérivation. Nous montrons tout d'abord les trois lemmes suivants.

Lemme 3.11 *Si $\rho_i.R_i \stackrel{n,r,C}{\rightarrow}_P \rho_f.R_f$ est une transition libre (ou satisfaisant P^Γ), alors, pour toute substitution σ, la transition $\rho_i.R_i \stackrel{n,r,C}{\rightarrow}_P \sigma\rho_f.\sigma R_f$ est une transition libre.*

PREUVE. Conséquence immédiate des définitions. ◀

Lemme 3.12 *Si* $\rho.R \overset{n,r,C}{\rightarrow}_P \theta\rho.\underbrace{\theta R[n \leftarrow r(C^-)]}_{R'}$ *est une transition libre, alors :*

$$h_p(c_p(C, P_\varnothing)) \models R' \Rightarrow h_p(c_p(C, P_\varnothing)) \models \theta R$$

PREUVE. Soit I une interprétation de $\Sigma \cup \Pi$ de domaine D telle que $\models_I C$, et v une valuation sur D. Soit q la longueur de la requête R. Par hypothèse :

$$\widetilde{\theta R}^I(v) = \bigwedge_{i=0\ b}^{n-1} \theta R^I_{/i}(v) \wedge_b \widetilde{\theta r(C^-)}^I(v) \wedge_b \bigwedge_{i=n+1\ b}^{q-1} \theta R^I_{/i}(v) = \mathsf{true}$$

et il suffit de montrer que $\theta R^I_{/n}(v) = \mathsf{true}$. Si $\theta R^I_{/n}(v) = \mathsf{false}$, alors puisque par hypothèse on a $\theta R_{/n} = \theta r(C^+)$, il vient $\theta r(C^+)^I(v) = \mathsf{false}$. Cependant, I est un modèle de C et, d'après le lemme 3.10, I est aussi un modèle de $\theta r(C)$. On obtient alors :

$$\theta r(C)^I(v) = \theta r(C^+)^I(v) \vee_b \theta r(C^-)^I(v) = \mathsf{true}$$

On a donc $\theta r(C^-)^I(v) = \mathsf{true}$ et le lemme 3.9 permet d'établir $\widetilde{\theta r(C^-)}^I(v) = \mathsf{false}$ ce qui est contradictoire. ◀

Ce lemme ne dépend pas du choix de l'unificateur utilisé : tout unificateur est satisfaisant.

Lemme 3.13 *Si, à partir d'un programme défini P et d'une requête R_0, on obtient la dérivation de $\mathbb{D}_d$ satisfaisant $P^{\mathbb{D}_d}$: $\rho.R_0 \overset{*}{\rightarrow}_P \theta\rho.R$, alors on a $h_p(P) \models R \Rightarrow h_p(P) \models \theta R_0$.*

PREUVE. Nous utilisons ici une schéma d'induction «droit».

- Pour $\rho.R_0 \overset{n,r,C}{\rightarrow}_P \theta\rho.\underbrace{\theta R_0[n \leftarrow r(C^-)]}_{R}$

Par définition, cette transition est libre et le lemme 3.12 permet de conclure.

- Pour $d^d_c(t,d)$: $\rho.R_0 \overset{n,r,C}{\rightarrow}_P \theta\rho.\underbrace{\theta R_0[n \leftarrow r(C^-)]}_{R_1} \overset{*}{\rightarrow}_P \sigma\theta\rho.\sigma\theta R_2$

Soit I une interprétation de $\Sigma \cup \Pi$ de domaine D, telle que $\models_I P$, et v une valuation sur D. Par hypothèse, I est un modèle de $\widetilde{\sigma\theta R_2}$ et l'hypothèse d'induction permet de montrer que I est aussi un modèle de $\sigma\theta R_0[\widetilde{n \leftarrow r}(C^-)]$. Puisque $P^\Gamma(t)$, d'après le lemme 3.11, la transition :

$$\rho.R_0 \overset{n,r,C}{\rightarrow}_P \sigma\theta\rho.\underbrace{\sigma\theta R_0[n \leftarrow r(C^-)]}_{\sigma R_1}$$

```
Definition u_state_t : trans -> subst :=
 [t:trans]<subst>Case t of
  [ei:state][p:program][n:nat][c:clause][r:rename]
  [s:subst][a:atom][ef:state]
   s end.
```

```
Fixpoint answer [d:deriv] : subst :=
<subst>Case d of
 [t0:trans](u_state_t t0)
 [d1:deriv][t1:trans]
  ([w:var](Subst_t (u_state_t t1) ((answer d1) w)))
      end.
```

```
Lemma soundness : (d:deriv)
 (Deriv_ok d)                           ->
 (Fst (state_init_d d))=tv              ->
 (Snd (state_end_d d))=true_req         ->
 (semantic_csq
    (hp (p_deriv d))
    (subst_req (answer d) (Snd (state_init_t (head_d d))))).
```

TAB. 3.10 – *Validité de la SLD-résolution*

est une transition libre. D'après le lemme 3.12, $h_p(P) \models \sigma\theta R_0$ et on peut conclure. ◀

Le théorème de validité, dont l'énoncé formel dans le système COQ est présenté dans le tableau 3.10, peut être vu comme un corollaire de ce lemme ($R = r_\varnothing$).

Théorème 3.1 (Soundness) *Si, à partir d'un programme défini P et d'une requête R, on obtient la dérivation satisfaisant $P^{\mathbb{D}}$: $s_{id}.R \overset{*}{\to}_P \theta.r_\varnothing$, alors $h_p(P) \models \theta R$.*

PREUVE. Si $d \in \mathbb{D}_g$ vérifie $P^{\mathbb{D}}(d)$, alors on a $P^{\mathbb{D}_d}(\tau_g^d(d))$ et le lemme 3.13 permet de conclure. ◀

3.3.3 Théorème de complétude

Le théorème de complétude exprime le fait que si une requête R admet une solution à partir d'un programme défini, alors elle admet une réponse à partir de ce programme. Sous cette forme, le théorème de complétude est plus «faible» que le théorème de validité formalisé (et aussi plus faible que le théorème de complétude généralement énoncé) : nous ne montrons pas que pour toute solution θ, il existe une réponse σ, telle que $\sigma R \leqslant \theta R$.

Modèles de Herbrand

Les formules logiques du calcul des prédicats s'interprètent dans des structures quelconques formées d'un domaine, d'opérateurs (fonctions) et de relations. L'intérêt des formes clausales provient du fait que l'étude de l'existence de modèles, pour un ensemble de clauses de Horn, peut être restreinte à une classe d'interprétations associée, de manière syntaxique, au langage : les interprétations de Herbrand (voir tableau 3.11). Ces interprétations permettent d'interpréter les symboles par eux-mêmes. Tout d'abord, nous définissons inductivement l'ensemble $T_\Sigma[\varnothing]$ des termes fermés (termes ne contenant pas de variables), aussi appelé univers de Herbrand de Σ. A cette occasion, la famille des listes de termes fermés de longueur finie $\sum_{n\in\mathbb{N}} L_{n,\Sigma}[\varnothing]$ est aussi définie. Similairement, l'ensemble $At_{\Sigma,\Pi}[\varnothing]$ des atomes fermés est défini inductivement comme l'ensemble des atomes obtenus à partir des éléments de $\sum_{n\in\mathbb{N}} L_{n,\Sigma}[\varnothing]$, et est aussi appelé base de Herbrand de $\Sigma\cup\Pi$. Enfin, l'ensemble des substitutions fermées $S_\varnothing[X]$ est défini comme l'ensemble des applications de X vers $T_\Sigma[\varnothing]$ (cet ensemble correspond exactement à $T_\Sigma[\varnothing]^X$). Nous supposons ici que la signature Σ contient au moins un symbole fonctionnel d'arité nulle, noté f_h^0 (il s'agit de la constante de Herbrand, qui est posée en axiome) afin que les ensembles que nous venons de définir soient non vides[2]. L'interprétation de Herbrand des symboles fonctionnels de Σ est alors définie par l'interprétation, notée $\mathcal{H}$, de domaine $T_\Sigma[\varnothing]$, qui pour tout $f\in\Sigma$ associe l'application $f^{\mathcal{H}} : \mathbb{L}_{ar(f)}[T_\Sigma[\varnothing]] \to T_\Sigma[\varnothing]$ définie par :

$$\forall f\in\Sigma \quad \forall l\in\mathbb{L}_{ar(f)}[T_\Sigma[\varnothing]] \quad f^{\mathcal{H}}(l) = \mathsf{tfg}(f,l')$$

où tfg est l'unique constructeur de $T_\Sigma[\varnothing]$ et où l' est la «transposition» de l dans $L_{ar(f),\Sigma}[\varnothing]$. Moins formellement, $\mathcal{H}$ permet d'interpréter les symboles de fonction comme des constructeurs de termes. Toute interprétation de Π de domaine $T_\Sigma[\varnothing]$, «basée sur» $\mathcal{H}$, est une interprétation de Herbrand de Π. Un modèle de Herbrand d'un objet o (atome, requête, clause, ...) est une interprétation de Herbrand de $\Sigma\cup\Pi$ qui est un modèle de o.

Prédicats sur la base de Herbrand

L'interprétation de Herbrand des symboles fonctionnels étant fixée, il semble possible de caractériser une interprétation de Herbrand, I, de $\Sigma\cup\Pi$ (objet fonctionnel), par un sous-ensemble de la base de Herbrand, défini par un prédicat $\pi(I)$ sur $At_{\Sigma,\Pi}[\varnothing]$, constitué de tous les atomes fermés dont I est un modèle :

$$\forall a\in At_{\Sigma,\Pi}[\varnothing] \quad (\models_I a \Rightarrow \pi(I,a))$$

2. Dans [34], J.H. Gallier présente une discussion sur la nécessité de cette constante dans la preuve du théorème de Herbrand.

Cependant, les spécifications de I et $\pi(I)$ sont de nature différente :

$$I : \underbrace{\prod_{q\in\Pi}(\mathbb{L}_{ar(q)}[T_\Sigma[\varnothing]] \to \mathbb{B})}_{\textsf{Set}} \qquad \pi(I) \colon \underbrace{At_{\Sigma,\Pi}[\varnothing] \to \textsf{Prop}}_{\textsf{Type}}$$

Tandis que pour tout atome fermé a, et toute interprétation (de Herbrand) I, la proposition «*I est un modèle de a*» est décidable :

$$\forall a \in At_{\Sigma,\Pi}[\varnothing] \quad (\models_I a \vee \neg \models_I a)$$

si l'on ne connaît pas l'interprétation dont est issu un prédicat p quelconque sur la base de Herbrand, la proposition $p(a)$ n'est pas décidable ($p(a) \vee \neg p(a)$ n'est pas prouvable). Pour contourner ce problème, il faudra poser en hypothèse de certains lemmes la décidabilité des prédicats sur la base de Herbrand, ce qui revient à supposer que l'on sait construire une interprétation de Herbrand à partir d'un prédicat sur la base de Herbrand (nous limiterons évidemment l'emploi de cette hypothèse à cet usage). Le théorème de complétude de la SLD-résolution s'obtiendra au prix de cette hypothèse (forte), implicitement suggérée par l'identification Set/Type. Cette hypothèse est indispensable. En effet, nous allons voir que l'identification d'une interprétation de Herbrand avec un sous-ensemble de la base de Herbrand sert à établir des propriétés sur l'opérateur T_P associé à un programme P, et permettant de définir une sémantique par point fixe. Cet opérateur est généralement défini par une application de type :

$$(At_{\Sigma,\Pi}[\varnothing] \to \textsf{Prop}) \to (At_{\Sigma,\Pi}[\varnothing] \to \textsf{Prop})$$

Si P est un programme défini et p une interprétation de Herbrand (i.e. un prédicat sur la base de Herbrand), alors $T_P(p)$ est un prédicat sur $At_{\Sigma,\Pi}[\varnothing]$ caractérisant les atomes fermés a tels que pour une clause $c \in P$ et pour une substitution fermée σ telle que $a = \sigma c^+$, on ait :

$$\forall a_0 \in At_{\Sigma,\Pi}[X] \quad a_0 \in c^- \Rightarrow p(\sigma a_0)$$

$T_P(p)(a)$ est vrai pour ces atomes. Toutefois, même si p est initialement décidable, $T_P(p)$ n'est pas décidable. En effet, décider si un atome a est satisfait par $T_P(p)$ revient à chercher une substitution fermée θ et une clause $c^+ \leftarrow c^-$ telle que les atomes de θc^- soient satisfaits par p. Or, en considérant les substitutions fermées comme des fonctions partielles, on sait seulement décider si une substitution θ^+ telle que $\theta^+ c^+ = a$ et $dom(\theta^+) = var(c^+)$ existe et θ^+ correspond alors à la restriction de θ aux variables de c^+. Une fois θ^+ déterminée, il reste à décider s'il existe une substitution fermée θ^- telle que $dom(\theta^-) = var(\theta^+ c^-)$ et telle que les atomes de $\theta^- \theta^+ c^-$ soient satisfaits par $T_P(p)$. C'est précisément ce problème qui n'est pas décidable. En effet, si l'on considère par exemple la signature $\Sigma_{\mathbb{N}} = \{0, S\}$, alors il se peut très

Termes fermés ($T_\Sigma[\varnothing]$)

```
Mutual Inductive Ground_Term : Set :=
 tfg : (f:fun)(glt (arity ar f)) -> Ground_Term
with glt : nat -> Set :=
 nilg  : (glt O) |
 consg :(n:nat)Ground_Term -> (glt n) -> (glt (S n)).
```

Atomes fermés ($At_{\Sigma,\Pi}[\varnothing]$)

```
Inductive Ground_Atom : Set :=
plg : (p:predic)(glt (arity_p p)) -> Ground_Atom.
```

Substitutions fermées ($S_\varnothing[X]$)

```
Definition Ground_subst : Set := var -> Ground_Term.
```

Constante de Herbrand (f_h^0)

```
Parameter fh : fun.
Axiom ar_fh : (arity ar fh)=O.
```

Interprétation de Herbrand de Σ ($\mathcal{H}$)

```
Definition f_hi : (f_i Ground_Term) :=
  [f:fun][l:(LIST Ground_Term (arity ar f))]
  (tfg f (LIST_to_glt (arity ar f) l)).
```

Interprétation de Herbrand de $\Sigma \cup \Pi$

```
⟨f_hi:(f_i Ground_Term),Ip:(p_i Ground_Term)⟩
```

Prédicat sur $At_{\Sigma,\Pi}[\varnothing]$ associé à I ($\pi(I)$)

```
Definition MH:(p_i Ground_Term)->Ground_Atom->Prop :=
  [Ip:(p_i Ground_Term)][a:Ground_Atom]
    (at_valid Ground_Term f_hi Ip (Ground_to_atom a)).
```

Nature des spécifications

```
 I:(p_i Ground_Term):Set
 (MH I):(Ground_Atom->Prop):Type
```

TAB. 3.11 – *Interprétations de Herbrand*

bien que pour un prédicat p, une interprétation $I = \{p(m,n),\ (m,n) \in D\}$, où D est une partie récursive de $T_{\Sigma_{\mathbb{N}}}[\varnothing]^2$, soit calculable mais que l'ensemble $\{m \in T_{\Sigma_{\mathbb{N}}}[\varnothing], \exists n \in T_{\Sigma_{\mathbb{N}}}[\varnothing],\ p(m,n)\}$ ne soit pas calculable. C'est le cas, par exemple, si $p(m,n)$ est interprété par «*le m-ième programme termine en n étapes*». Enfin, signalons que lorsque le programme est tel que chaque clause c vérifie $var(c^-) \subseteq var(c^+)$, $T_P(p)$ est décidable si p est décidable : il suffit de vérifier que chacun des atomes de θc^- vérifie p.

Existence d'un modèle pour un ensemble fini de clauses de Horn

A toute interprétation, I, de $\Sigma \cup \Pi$, est associée une interprétation de Herbrand, $H(I)$, définie par :

- $\forall f \in \Sigma \quad f^{H(I)} = f^{\mathcal{H}}$

- $\forall p \in \Pi \quad \forall l \in \mathbb{L}_{ar(p)}[T_\Sigma[\varnothing]] \quad p^{H(I)}(l^{\mathcal{H}}) = p^I(l^I)$ (où l^I correspond à l'interprétation de la liste de termes fermés l)

On peut alors obtenir une preuve formelle du célèbre théorème :

Théorème 3.2 (Herbrand) *Un ensemble fini de clauses de Horn admet un modèle si et seulement si il admet un modèle de Herbrand.*

L'étude de la satisfiabilité (i.e. l'existence de modèles) d'un ensemble de clauses de Horn peut donc être restreinte aux seules interprétations «syntaxiques» que constituent les interprétations de Herbrand. Cette restriction permet de considérer des propriétés sémantiques avec des moyens syntaxiques mais cette propriété n'est plus vérifiée si l'on considère des formules logiques quelconques (par exemple, la formule $p(a) \wedge \exists x \neg p(x)$ est satisfiable mais n'admet pas de modèle de Herbrand). D'autre part, toutes les formules ne peuvent s'écrire sous la forme d'une clause de Horn (c'est le cas, par exemple, pour $A \vee B$). D'un point de vue sémantique, l'intérêt de la restriction aux clauses de Horn provient du fait que tout ensemble de clauses de Horn possède un plus petit modèle et vérifie la propriété de fermeture pour l'intersection des modèles.

Lemme 3.14 (Model intersection property [67]) *Si S est un ensemble de clauses de Horn et $\{M_i\}_i$ un ensemble non vide de modèles de Herbrand de S, alors $\cap_i M_i$ est un modèle de Herbrand de S.*

Dans [69], J.A. Makowsky montre que le fragment Hornien est le plus grand sous-ensemble de la logique du premier ordre satisfaisant cette propriété. Cette propriété est donc caractéristique des formules logiques qui peuvent s'écrire sous forme de clauses de Horn. Par exemple, $A \vee B$ ne vérifie pas cette propriété puisque $\{\{A\}, \{B\}\}$ est bien une paire de modèles de $A \vee B$ dont l'intersection ($\varnothing$) n'est pas un modèle.

Plus petit modèle de Herbrand d'un programme défini

La base de Herbrand de la signature $\Sigma \cup \Pi$ est un modèle de tout programme défini construit sur cette signature, aussi, la propriété d'intersection des modèles établit l'existence d'un plus petit modèle de Herbrand d'un programme. Pour tout programme défini P, le plus petit modèle de Herbrand de P, noté $\mathcal{M}_P$, est défini, par un prédicat sur la base de Herbrand, caractérisant les atomes fermés présents dans l'intersection de tous les modèles de Herbrand de P (voir tableau 3.12) :

$$\begin{array}{l} \forall P \in P_{\Sigma,\Pi}[X] \, \forall a \in At_{\Sigma,\Pi}[\varnothing] \\ \qquad ((\forall I \in I_{\Sigma,\Pi}[T_\Sigma[\varnothing]] \models_I P \Rightarrow \pi(I,a)) \Rightarrow \mathcal{M}_P(a)) \end{array}$$

On montre que les atomes caractérisés par $\mathcal{M}_P$ correspondent exactement à ceux qui sont conséquences sémantiques du programme défini P.

Lemme 3.15 ([96]) $\forall a \in At_{\Sigma,\Pi}[\varnothing] \quad \mathcal{M}_P(a) \Leftrightarrow h_p(P) \models c_r(a, r_\varnothing)$

PREUVE. Soit a un atome fermé et I un modèle de P. Puisque $\mathcal{M}_P(a)$, pour toute interprétation de Herbrand I_H, $\models_{I_H} P$ implique $\pi(I_H, a)$. De plus, I est un modèle de P implique $\models_{H(I)} P$, et on a donc $\pi(H(I), a)$. $H(I)$ est par conséquent un modèle de a. De plus, puisque a ne contient pas de variables, on montre que I est aussi un modèle de a, ce qui prouve $h_p(P) \models c_r(a, r_\varnothing)$. Réciproquement, si $h_p(P) \models c_r(a, r_\varnothing)$, alors I est un modèle de a et on a $\pi(H(I), a)$ ce qui permet de conclure à $\mathcal{M}_P(a)$. ◀

Opérateur de «conséquence immédiate»

Nous présentons à présent une caractérisation de cet ensemble en utilisant la notion de point fixe d'un opérateur : l'opérateur de «conséquence immédiate» associé à un programme défini P, noté T_P. Nous verrons que cet opérateur, introduit par M.H. van Emden et R.A. Kowalski, fournit un lien entre sémantique déclarative et sémantique opérationnelle des programmes définis. En utilisant l'identification d'une interprétation de Herbrand à un sous-ensemble de la base de Herbrand, cet opérateur est généralement défini inductivement par :

$$(T_P) : \frac{\begin{array}{ll} a \in At_{\Sigma,\Pi}[\varnothing] & p : At_{\Sigma,\Pi}[\varnothing] \to \mathsf{Prop} \\ \sigma \in S_\varnothing[X] & c \in P \\ \sigma c^+ = a & \forall a_0 \in c^- \ \ p(\sigma a_0) \end{array}}{T_P(p, a)}$$

Le lemme que nous formalisons à présent établit un lien entre les modèles de Herbrand d'un programme défini P et son opérateur T_P :

Lemme 3.16 *Soit P un programme défini et I une interprétation de Herbrand. I est un modèle de P si et seulement si $T_P(\pi(I)) \subseteq \pi(I)$. Plus formellement :*

$$\models_I P \Leftrightarrow \forall a \in At_{\Sigma,\Pi}[\varnothing] \quad (T_P(\pi(I), a) \Rightarrow \pi(I, a))$$

PREUVE. ($\Rightarrow$). Soit I un modèle de Herbrand de P et a un atome fermé. Si $T_P(\pi(I), a)$, alors pour une clause $c \in P$ et pour une substitution fermée σ telle que $a = \sigma c^+$:

$$\forall a_0 \in At_{\Sigma,\Pi}[X] \quad (a_0 \in c^- \Rightarrow \pi(I, \sigma a_0))$$

Puisque $\models_I P$, on a $\models_I c$. I est donc un modèle de Herbrand de σc. Soit v une valuation sur $T_\Sigma[\varnothing]$, deux cas se présentent. Si $(\sigma c^+)^I(v) = a^I(v) = \mathsf{true}$, alors $\pi(I, a)$ et on peut conclure. Sinon, $(\sigma c^-)^I(v) = \mathsf{true}$ et alors on a :

$$\widetilde{(\sigma c^-)}^I(v) = \mathsf{false}$$

ce qui contredit l'hypothèse $T_P(\pi(I), a)$.
$(\Leftarrow)$. Soit I une interprétation de Herbrand telle que $T_p(\pi(I)) \subseteq \pi(I)$. I est un modèle de P si, pour chaque clause c de P, $\models_I c$, ce qui peut s'écrire :

$$\forall v \in T_\Sigma[\varnothing]^X \quad (\widetilde{c^-}^I(v) = \mathsf{true} \Rightarrow (c^+)^I(v) = \mathsf{true})$$

autrement dit :

$$\forall v \in T_\Sigma[\varnothing]^X \; ((\forall a \in At_{\Sigma,\Pi}[X] \quad a \in c^- \Rightarrow a^I(v) = \mathsf{true}) \Rightarrow (c^+)^I(v) = \mathsf{true})$$

c'est à dire :

$$\forall v \in T_\Sigma[\varnothing]^X \; ((\forall a \in At_{\Sigma,\Pi}[X] \quad a \in c^- \Rightarrow \pi(I, v(a))) \Rightarrow (c^+)^I(v) = \mathsf{true})$$

L'hypothèse $T_p(\pi(I)) \subseteq \pi(I)$ permet alors de conclure. ◀

Afin d'alléger les notations, étant donnés deux prédicats p_1 et p_2 sur un ensemble quelconque A, nous écrirons par la suite $p_1 \subseteq p_2$ pour exprimer $\forall x \in A \; p_1(x) \Rightarrow p_2(x)$.

Points fixes de T_P

La sémantique formelle d'un langage de programmation peut être définie en terme de plus petite solution d'une équation de point fixe. En programmation logique, la sémantique de point fixe est identifiée à la plus petite solution de l'équation de point fixe $I = T_P(I)$. Lorsque $T_P(p) \subseteq p$, p est appelé un «pré-point fixe» de T_P. Aussi, pour caractériser les modèles de Herbrand d'un programme défini P, nous allons utiliser les propriétés sur les points fixes de son opérateur associé T_P. En effet, «$\{At_{\Sigma,\Pi}[\varnothing] \to \mathsf{Prop}\}$» ordonné par l'«inclusion ensembliste» formant un treillis complet sur lequel T_P est un opérateur continu, $T_P^{\uparrow\omega}$ correspond au plus petit point fixe de T_P et peut donc être obtenu par approximations successives (théorèmes de Kleene, Knaster et Tarski [95] présentés dans le chapitre 1 et formalisés en annexe A) en ω étapes. Nous montrons donc les lemmes suivants.

Lemme 3.17 *T_P est un opérateur continu.*

Preuve. D'après le lemme 1.5, il suffit de montrer que T_P est : (i) monotone et (ii) finitaire. (i) Soit p_1 et p_2 deux prédicats sur la base de Herbrand, tels que $p_1 \subseteq p_2$ et σc^+ un atome fermé caractérisé par $T_P(p_1)$. Par définition, les atomes a de c^- vérifient $p_1(\sigma a)$. Aussi, $p_2(\sigma a)$ est vérifié pour ces atomes et il vient $T_P(p_2, \sigma c^+)$. On a donc $T_P(p_1) \subseteq T_P(p_2)$ et T_P est donc bien un opérateur monotone. (ii) Soit $(p_n)_{n\geq 0}$ une «suite croissante de parties de la base de Herbrand» et σc^+ un atome fermé tel que $T_P(\bigcup_{n\geq 0} p_n, \sigma c^+)$. Par définition, on a :

$$\forall a \in At_{\Sigma,\Pi}[X] \quad (a \in c^- \Rightarrow (\bigcup_{n\geq 0} p_n)(\sigma a))$$

Puisque $(p_n)_{n\geq 0}$ est une suite croissante, c'est que pour un naturel k, on a :

$$\forall a \in At_{\Sigma,\Pi}[X] \quad (a \in c^- \Rightarrow p_k(\sigma a))$$

et on a alors $T_P(p_k, \sigma c^+)$ ce qui permet de conclure à :

$$(\bigcup_{n\geq 0} T_P(p_n))(\sigma c^+)$$

T_P est donc bien un opérateur finitaire. ◀

Puisque T_P est un opérateur continu et puisque $\mathcal{M}_P$ est défini comme l'intersection de tous les modèles de Herbrand de P, qui sont, d'après le lemme 3.16, les parties closes par T_P, nous souhaiterions établir l'égalité $\mathcal{M}_P = \mathsf{Ind}(T_P) = T_P^{\uparrow\omega}$. Cependant, alors que l'identification d'une interprétation de Herbrand à un sous-ensemble de la base de Herbrand caractérisé par un prédicat, permet facilement de formaliser le prédicat $\mathcal{M}_P$, l'opérateur T_P et ses points fixes, cette identification «rend» indécidable la proposition «*cette interprétation est un modèle de cet atome fermé*» nécessaire dans certaines preuves et alors posée en hypothèse. Le théorème de caractérisation qui suit fait appel à une telle hypothèse.

Théorème 3.3 ([96]) *Sous l'hypothèse :*

$$\forall p \in At_{\Sigma,\Pi}[\varnothing] \rightarrow \mathsf{Prop} \quad \forall a \in At_{\Sigma,\Pi}[\varnothing] \quad p(a) \vee \neg p(a)$$

on a $\mathcal{M}_P = \mathsf{Ind}(T_P) = T_P^{\uparrow\omega}$.

Preuve. Soit a un atome fermé tel que $\mathcal{M}_P(a)$. Par définition, tout modèle I de P est aussi un modèle de a. Soit p un prédicat sur $At_{\Sigma,\Pi}[\varnothing]$ caractérisant une partie close par T_P de la base de Herbrand et I l'interprétation dont p est issue, c'est à dire l'interprétation I telle que $\pi(I) = p$ (l'hypothèse de décidabilité est utilisée pour construire I). D'après le lemme 3.16, on a $\pi(I, a)$. Réciproquement, soit I un modèle de Herbrand de P. $\pi(I)$ caractérise une partie close par T_P et toujours d'après le lemme 3.16, nous pouvons conclure. La partie $\mathsf{Ind}(T_P) = T_P^{\uparrow\omega}$ de l'égalité est prouvée en utilisant le théorème 1.2. ◀

Ensemble des succès d'un programme défini

L'ensemble des succès d'un programme constitue la contre-partie opérationnelle du plus petit modèle de Herbrand de ce programme (voir tableau 3.12) et est défini comme l'ensemble des atomes fermés admettant une réfutation (i.e. l'ensemble des atomes à partir desquels on peut construire une dérivation se terminant par la requête vide). Les atomes de cet ensemble sont caractérisés par un prédicat, noté S_P, sur $At_{\Sigma,\Pi}[\varnothing]$. Nous cherchons à

$\mathcal{M}_P$
`Definition Mp : program -> Ground_Atom -> Prop :=`
` [p:program][a:Ground_Atom]`
` ((Ip:(p_i Ground_Term))`
` (P_valid Ground_Term f_hi Ip p) -> (MH Ip a)).`

Opérateur T_P
`Inductive Tp [p:program;Ip:(Ground_Atom->Prop);a:Ground_Atom]`
`          :Prop := Tpi: (c:clause)(v:Ground_subst)`
`    (IS_IN_P c p)->(a=(Gsubst_at v (head_c c)))->`
`    ((a0:atom)(IS_IN_R a0 (body_c c))->(Ip (Gsubst_at v a0)))`
`    ->(Tp p Ip a).`

S_P
`Inductive Success [p:program;a:Ground_Atom]:Prop :=`
`scs:(d:deriv)`
` (p_deriv d)=p->(Deriv_ok d)->`
` (Fst (state_init_d d))=tv->(Snd (state_init_d d))=`
` (cons_req (Ground_to_atom a) true_req)->`
` (Snd (state_end_d d))=true_req-> (Success p a).`

TAB. 3.12 – $\mathcal{M}_P$, T_P *et* S_P

établir la correspondance entre $\mathcal{M}_P$ et S_P. Ce résultat est classique [7], et est prouvé par induction. Cependant, une partie de cette preuve est usuellement omise : il s'agit de l'étape de «combinaison de dérivations». Cette opération est délicate et la manière dont elle est effectuée est toujours passée sous silence. Il s'agit pourtant d'un «véritable calcul», puisque plusieurs renommages (des variables mises en jeu dans les dérivations) sont nécessaires afin de garantir les conditions de séparation des variables dans la dérivation finale. La formalisation de cette opération requiert les deux lemmes présentés dans le paragraphe suivant.

Renommages et combinaisons de dérivations

Afin de pouvoir combiner des dérivations, tout en conservant les conditions de séparation des variables, il est nécessaire de pouvoir renommer les variables utilisées lors d'une dérivation. C'est ce que permet le lemme suivant, inspiré d'un lemme similaire présenté dans [5] et dû à J.W. Lloyd et J.C. Shepherdson [68].

Lemme 3.18 (Variant lemma) *Soit ℓ une liste finie de symboles de variable et d_1 la dérivation $s_{id}.R \xrightarrow{*}_P \rho_1.R_1$ satisfaisant $P^{\mathbb{D}}$. Il existe une dérivation d_2: $s_{id}.R \xrightarrow{*}_P \rho_2.R_2$, satisfaisant $P^{\mathbb{D}}$, telle que $\vartheta(d_2) \cap \ell = \varnothing$. De plus, pour une substitution s_r telle que $dom(s_r) \subseteq var(R_2)$, on a $R_1 = s_r R_2$.*

PREUVE. Induction sur d_1.

• Pour $s_{id}.R \overset{n,r_1,C}{\to_P} \rho_1.\rho_1 R[n \leftarrow r_1 C^-]$

Soit r_2 une substitution de renommage satisfaisant le prédicat P^R et telle que $(var(R) \cup \ell) \cap range(r_2) = \varnothing$ (cette substitution est obtenue facilement puisque X est indexé sur $\mathbb{N}$). Nous pouvons construire une substitution θ vérifiant :

$$\forall x \in C \quad \theta r_2 x = \rho_1 r_1 x \quad (\text{i.e. } \theta r_2 C = \rho_1 r_1 C)$$

Par hypothèse, on a :

$$\rho_1 R_{/n} = \rho_1 r_1 C^+ \tag{3.6}$$

Aussi, la substitution :

$$\lambda x.(\text{ si } x \in R \text{ alors } \rho_1(x) \text{ sinon } \theta(x))$$

est un unificateur de $R_{/n}$ et $r_2 C^+$ qui admettent alors un unificateur principal ρ_2. Nous pouvons donc construire la dérivation :

$$s_{id}.R \overset{n,r_2,C}{\to_P} \rho_2.\rho_2 R[n \leftarrow r_2 C^-]$$

Nous cherchons à présent à construire une substitution s_r telle que :

$$\rho_1 R[n \leftarrow r_1 C^-] = s_r \rho_2 R[n \leftarrow r_2 C^-]$$

Tout d'abord, on montre qu'il existe une substitution de renommage r_C satisfaisant :

$$r_1 C = r_C r_2 C \quad \text{et} \quad dom(r_C) \subseteq var(r_2 C) \tag{3.7}$$

Par conséquent, par définition des conditions de renommage imposées par le prédicat $P^{\mathbb{D}}$, il vient :

$$r_C R = R \tag{3.8}$$

L'équation (3.6) permet alors d'établir $\rho_1 r_C R_{/n} = \rho_1 r_1 C^+$, et d'après (3.7), nous avons $\rho_1 r_C R_{/n} = \rho_1 r_C r_2 C^+$, ce qui permet de considérer $\rho_1 r_C$ comme un unificateur de $R_{/n}$ et $r_2 C^+$. ρ_2 étant l'unificateur principal de ces deux atomes, il vient $\rho_2 \leqslant \rho_1 r_C$. Il existe donc une substitution s_r telle que :

$$s_r \rho_2 = \rho_1 r_C \tag{3.9}$$

En considérant la restriction de s_r aux variables présentes dans la requête $\rho_2 R[n \leftarrow r_2 C^-]$, nous pouvons conclure puisque :

$$\begin{aligned} \rho_1 R[n \leftarrow r_1 C^-] &= \rho_1 R[n \leftarrow r_C r_2 C^-] && (3.7)\\ &= \rho_1 r_C R[n \leftarrow r_2 C^-] && (3.8)\\ &= s_r \rho_2 R[n \leftarrow r_2 C^-] && (3.9) \end{aligned}$$

• Pour $s_{id}.R \xrightarrow{*}_P \rho.R_1 \overset{n,r_1,C}{\rightarrow}_P \theta_1\rho.\theta_1 R_1[n \leftarrow r_1 C^-]$
Par hypothèse d'induction, il existe une dérivation $d_I\colon s_{id}.R \xrightarrow{*}_P \eta.R_2$ et pour une substitution σ_I telle que :

$$dom(\sigma_I) \subseteq var(R_2) \tag{3.10}$$

on a :

$$R_1 = \sigma_I R_2 \tag{3.11}$$

Soit r_2 une substitution de renommage satisfaisant $P^R(r_2)$ et telle que :

$$(var(R) \cup \vartheta(d_I) \cup \ell) \cap range(r_2) = \varnothing$$

Nous pouvons construire une substitution de renommage r_C vérifiant :

$$dom(r_C) \subseteq var(r_2 C) \tag{3.12}$$

$$r_1 C = r_C r_2 C \tag{3.13}$$

D'après (3.10), toutes les variables du domaine de σ_I apparaissent dans R_2 et, d'après le lemme 3.5 et la définition de r_2, n'apparaissent pas dans $r_2 C$ (i.e. dans le domaine de r_C). Aussi, nous avons :

$$dom(\sigma_I) \cap dom(r_C) = \varnothing$$

Toujours d'après (3.10), il vient :

$$dom(\sigma_I) \cap var(r_2 C) = \varnothing \tag{3.14}$$

et d'après (3.12), on obtient :

$$dom(r_C) \cap var(R_2) = \varnothing$$

On peut alors définir la substitution :

$$r_X \colon = \lambda x. \begin{pmatrix} \text{si } x \in dom(\sigma_I) & \\ \text{alors } \sigma_I(x) & \\ \text{sinon} & \begin{pmatrix} \text{si } x \in dom(r_C) \\ \text{alors } \mathsf{tv}(r_C(x)) \\ \text{sinon } \mathsf{tv}(x) \end{pmatrix} \end{pmatrix}$$

D'après (3.14), il vient :

$$r_X r_2 C = r_C r_2 C \tag{3.15}$$

De plus, d'après (3.12), on a :

$$r_X R_2 = \sigma_I R_2 \tag{3.16}$$

Enfin, par hypothèse, $\theta_1 R_{1_{/n}} = \theta_1 r_1 C^+$ et les équations (3.11) et (3.13) permettent d'établir $\theta_1 r_X R_{2_{/n}} = \theta_1 r_C r_2 C^+$. D'après l'égalité (3.15), il vient alors $\theta_1 r_X R_{2_{/n}} = \theta_1 r_X r_2 C^+$. $R_{2_{/n}}$ et $r_2 C^+$ sont donc unifiables et admettent alors un unificateur principal θ_2 tel que $\theta_2 \leqslant \theta_1 r_X$. Il existe donc une substitution σ telle que :

$$\sigma\theta_2 = \theta_1 r_X \tag{3.17}$$

Nous sommes maintenant en mesure de construire la dérivation :

$$s_{id}.R \xrightarrow{*}_P \eta.R_2 \overset{n,r_2,C}{\rightarrow_P} \theta_2\eta.\theta_2 R_2[n \leftarrow r_2 C^-]$$

Enfin, les deux états dérivés sont reliés comme suit :

$$\begin{array}{lll} \theta_1 R_1[n \leftarrow r_1 C^-] & = \theta_1 R_1[n \leftarrow r_C r_2 C^-] & (3.13) \\ & = \theta_1 R_1[n \leftarrow r_X r_2 C^-] & (3.15) \\ & = \theta_1(\sigma_I R_2)[n \leftarrow r_X r_2 C^-] & (3.11) \\ & = \theta_1(r_X R_2)[n \leftarrow r_X r_2 C^-] & (3.16) \\ & = \theta_1 r_X R_2[n \leftarrow r_2 C^-] & \\ & = \sigma\theta_2 R_2[n \leftarrow r_2 C^-] & (3.17) \end{array}$$

ce qui permet de conclure. ◀

Ce lemme permet alors de prouver le lemme suivant, énoncé dans [7].

Lemme 3.19 (Combinaison de dérivations) *Soit P un programme, R une requête non vide et σ une substitution fermée. Si tous les atomes présents dans σR appartiennent à l'ensemble des succès de P, alors il existe une dérivation $s_{id}.\sigma R \xrightarrow{*}_P \mu.r_\varnothing$ satisfaisant $P^{\mathbb{D}}$.*

Preuve. La preuve est obtenue par induction sur la requête R. Le cas $R = r_\varnothing$ contredit l'hypothèse. Pour $R = c_r(a, r)$, par hypothèse d'induction, si $r \neq r_\varnothing$ et si tous les atomes présents dans σr appartiennent à S_P, alors il existe une dérivation $s_{id}.\sigma r \xrightarrow{*}_P \mu_I.r_\varnothing$. De plus, par hypothèse, on a :

$$\forall a_0 \in At_{\Sigma,\Pi}[X] \quad (a_0 \in c_r(a, r) \Rightarrow S_P(\sigma a_0)) \tag{3.18}$$

Deux cas sont possibles. Si $r = r_\varnothing$, alors puisque $a \in c_r(a, r)$, d'après (3.18), il vient $S_P(\sigma a)$ et par définition, il existe une dérivation $s_{id}.\sigma a \xrightarrow{*}_P \mu.r_\varnothing$. Sinon ($r \neq r_\varnothing$), par hypothèse d'induction, il existe une dérivation :

$$d_1 : s_{id}.\sigma r \xrightarrow{*}_P \mu_I.r_\varnothing$$

De plus, d'après (3.18), on a $S_P(\sigma a)$ et il existe alors une dérivation :

$$d_2 : s_{id}.\sigma a \xrightarrow{*}_P \mu_H.r_\varnothing$$

Puisque σ est une substitution fermée, on peut prouver par induction sur d_1 que :

$$d_3 : s_{id}.\sigma c_r(a, r) \xrightarrow{*}_P \mu_I.\sigma a$$

est une dérivation satisfaisant $P^{\mathbb{ID}}$. Par ailleurs, à partir de d_2, et en appliquant le lemme 3.18, on peut construire une dérivation :

$$d_4 : \mu_I.\sigma a \xrightarrow{*}_P \mu_R.r_\varnothing$$

telle que $\vartheta(d_4) \cap \vartheta(d_3) = \varnothing$. On montre alors, par induction sur d_4, que :

$$d_5 : \mu_I.\sigma a \xrightarrow{*}_P \mu_R\mu_I.r_\varnothing$$

est une dérivation satisfaisant $P^{\mathbb{ID}}$. Les hypothèses de séparation des variables étant à présent satisfaites, on peut maintenant «concaténer» les deux dérivations d_3 et d_5 pour obtenir la dérivation $s_{id}.\sigma c_r(a, r) \xrightarrow{*} \mu.r_\varnothing$ ce qui permet de conclure. ◀

Complétude de la SLD-résolution

Nous pouvons à présent formaliser la preuve du théorème de complétude de la SLD-résolution. En effet, le lemme 3.19 nous permet de relier les prédicats $\mathcal{M}_P$ et S_P.

Lemme 3.20

1. $\forall a \in At_{\Sigma,\Pi}[\varnothing] \quad S_P(a) \Rightarrow \mathcal{M}_P(a) \quad$ (*i.e* $S_P \subseteq \mathcal{M}_P$)

2. Sous l'hypothèse

$$\forall p \in At_{\Sigma,\Pi}[\varnothing] \to \mathsf{Prop} \quad \forall a \in At_{\Sigma,\Pi}[\varnothing] \quad p(a) \vee \neg p(a)$$

on a $\forall a \in At_{\Sigma,\Pi}[\varnothing] \quad \mathcal{M}_P(a) \Rightarrow S_P(a) \quad$ (*i.e.* $\mathcal{M}_P \subseteq S_P$).

Preuve. (1). Soit a un atome fermé tel que $S_P(a)$. Par définition, il existe une dérivation $s_{id}.a \xrightarrow{*}_P \theta.r_\varnothing$. D'après le théorème de validité, on a $h_p(P) \models c_r(\theta a, r_\varnothing)$ et puisque $\theta a = a$ (a étant fermé), il vient $h_p(P) \models c_r(a, r_\varnothing)$. Le lemme 3.15 permet alors de conclure à $\mathcal{M}_P(a)$.
(2). Soit a un atome fermé tel que $\mathcal{M}_P(a)$. D'après le théorème 3.3, on a $T_P^{\uparrow\omega}(a)$. Par définition, pour un naturel k, on a $T_P^{\uparrow k}(a)$ et il suffit alors de prouver :

$$\forall k \in \mathbb{N} \quad \forall a \in At_{\Sigma,\Pi}[\varnothing] \quad (T_P^{\uparrow k}(a) \Rightarrow S_P(a))$$

Nous procédons par induction sur k.
Pour $k = 0$, puisque $T_P^{\uparrow 0} = \varnothing$, l'hypothèse $T_P^{\uparrow 0}(a)$ induit une contradiction.
Pour $k = n + 1$, par hypothèse d'induction, on a :

$$\forall a \in At_{\Sigma,\Pi}[\varnothing] \quad (T_P^{\uparrow n}(a) \Rightarrow S_P(a))$$

Par hypothèse, $T_P^{\uparrow k}(a)$ et donc $T_P(T_P^{\uparrow n}, a)$. Par conséquent, pour une clause $c \in P$ et pour une substitution fermée σ telle que $a = \sigma c^+$, on a :

$$\forall a_0 \in At_{\Sigma,\Pi}[X] \quad (a_0 \in c^- \Rightarrow T_P^{\uparrow n}(\sigma a_0))$$

Par hypothèse d'induction, il vient :

$$\forall a_0 \in At_{\Sigma,\Pi}[X] \quad (a_0 \in c^- \Rightarrow S_P(\sigma a_0))$$

Puisque σ est fermée, il existe une substitution fermée θ et une substitution de renommage r, telle que $P^R(r)$, qui renomme toutes les variables de c et seulement ces variables et satisfaisant :

$$\forall x \in c \quad \sigma x = \theta r x \quad (i.e. \quad \sigma c = \theta r c) \tag{3.19}$$

Considérons maintenant, la restriction de θ aux variables de rc^+, notée $\theta_{\restriction +}$. Puisque, par définition, $a = \sigma c^+$, nous pouvons construire la dérivation :

$$s_{id}.a \overset{0,r,c}{\to}_P \theta_{\restriction +}.\theta_{\restriction +} r c^- \tag{3.20}$$

Deux cas sont possibles. Si $c^- = r_\varnothing$, alors d'après (3.20), on a $S_P(a)$ ce qui permet de conclure. Sinon, si $c^- \neq r_\varnothing$, alors le lemme 3.19 permet de construire la dérivation $s_{id}.\sigma c^- \overset{*}{\to}_P \mu.r_\varnothing$. D'autre part, d'après (3.19), il existe une dérivation $s_{id}.\theta r c^- \overset{*}{\to}_P \mu.r_\varnothing$. Le lemme 3.18 permet alors de construire la dérivation :

$$s_{id}.\theta r c^- \overset{*}{\to}_P \mu_0.r_\varnothing \tag{3.21}$$

telle qu'aucune variable présente dans rc n'apparaît dans l'image d'une substitution de renommage utilisée lors de la dérivation (3.21). Considérons maintenant la restriction de θ aux variables de rc^-, notée $\theta_{\restriction -}$, il vient $\theta r c = \theta_{\restriction -}\theta_{\restriction +} r c$. A partir de (3.21), on peut donc construire la dérivation $s_{id}.\theta_{\restriction -}\theta_{\restriction +} r c^- \overset{*}{\to}_P \mu_0.r_\varnothing$ et en appliquant le lemme de généralisation, on obtient $s_{id}.\theta_{\restriction +} r c^- \overset{*}{\to}_P \mu_1.r_\varnothing$ ce qui permet de construire la dérivation :

$$\theta_{\restriction +}.\theta_{\restriction +} r c^- \overset{*}{\to}_P \mu_1 \theta_{\restriction +}.r_\varnothing$$

En combinant (3.20) avec cette dérivation, il vient :

$$s_{id}.a \overset{*}{\to}_P \mu_1 \theta_{\restriction +}.r_\varnothing$$

ce qui permet de conclure à $S_P(a)$. ◀

Grâce au lemme de généralisation, nous sommes enfin en mesure de formaliser la preuve du théorème de complétude (voir tableau 3.13) :

Théorème 3.4 (Completeness) *Soit P un programme défini, R une requête non vide et θ une solution pour R. Sous l'hypothèse*

$$\forall p \in At_{\Sigma,\Pi}[\varnothing] \to \mathsf{Prop} \quad \forall a \in At_{\Sigma,\Pi}[\varnothing] \quad p(a) \vee \neg p(a)$$

il existe une dérivation $s_{id}.R \overset{*}{\to}_P \rho.r_\varnothing$ satisfaisant $P^{\mathbb{D}}$.

```
Lemma completeness : (r:request)(s:subst)(p:program)
 ((pi:Ground_Atom -> Prop)(a:Ground_Atom){(pi a)}+{~(pi a)}) ->
 ~(r=true_req)->
 ->(semantic_csq (hp p) (subst_req s r))->
 (Ex [d:deriv]((Deriv_ok d) /\ (p_deriv d)=p /\
  (Fst (state_init_d d)) = tv /\ (Snd (state_init_d d)) = r  /\
  (Snd (state_end_d d)) = true_req )).
```

TAB. 3.13 – *Complétude de la SLD-résolution*

PREUVE. Par définition, puisque la substitution θ est une solution pour la requête R, on a $h_p(P) \models \theta R$. Soit σ_H la substitution $\lambda x.f_h^0$ (constante de Herbrand). D'après le lemme 3.10, on a $h_p(P) \models \sigma_H \theta R$, et par définition, il vient :

$$\forall a \in R \quad h_p(P) \models c_r(\sigma_H \theta a, r_\varnothing)$$

Aussi, d'après le lemme 3.15, on a :

$$\forall a \in R \quad \mathcal{M}_P(\sigma_H \theta a)$$

A partir du lemme 3.20, on obtient :

$$\forall a \in R \quad S_P(\sigma_H \theta a)$$

et le lemme 3.19 permet de construire la dérivation :

$$s_{id}.\sigma_H \theta R \overset{*}{\rightarrow}_P \mu.r_\varnothing$$

Le lemme 3.18 permet de renommer cette dérivation et d'obtenir :

$$s_{id}.\sigma_H \theta R \overset{*}{\rightarrow}_P \mu_0.r_\varnothing \tag{3.22}$$

où aucune variable apparaissant dans l'image d'une substitution de renommage utilisée lors de (3.22) n'est présente dans R. Nous pouvons alors appliquer le lemme de généralisation en considérant la restriction de $\sigma_H \theta$ aux variables de R, et obtenir la dérivation :

$$s_{id}.R \overset{*}{\rightarrow}_P \rho.r_\varnothing$$

qui permet de conclure. ◀

A propos du lien entre solution et réponse

Le théorème de complétude formalisé ici ne relie pas la réponse construite avec la solution. En effet, ce lien est généralement obtenu en menant le «tour de passe-passe» suivant :

> [5] (...) *Let $x_1, \ldots, x_n$ be the variables of θN. Enrich the language of P by adding new constants $a_1, \ldots, a_n$ and let γ be the substitution $\{x_1/a_1, \ldots, x_n/a_n\}$* (...) *there exists an SLD-refutation of*

> $P \cup \{\gamma\theta N\}$ (...). *By textually replacing in this refutation* a_i *by* x_i, *for* $i = 1, \ldots, n$ (...).

Ces opérations sont difficilement formalisables à partir de nos définitions (qui ne permettent ni l'ajout de constantes à la signature Σ, ni la construction de substitutions de constantes par des variables). Il semblerait que ce problème puisse être contourné en formalisant l'approche développée dans [13, 29, 30, 31] pour la sémantique déclarative des programmes définis, correspondant à la $\mathcal{S}$-sémantique et autorisant la présence de variables dans la base de Herbrand, capturant ainsi de manière plus fine la sémantique opérationnelle des programmes définis.

Chapitre 4

SLD preuves infinies

Tandis que les résultats classiques de la programmation logique, formalisés dans le chapitre précédent, concernent principalement la sémantique des calculs finis (i.e. fournissent une caractérisation des objets que l'on peut obtenir avec des dérivations finies), certains programmes logiques peuvent donner lieu à des dérivations infinies pour lesquelles ces résultats ne s'appliquent pas. Toutefois, certains objets sont, par nature, infinis et les programmes définis qui permettent de les construire, même s'ils ne terminent pas, effectuent bien un calcul «utile en un certain sens». Par exemple, le programme permettant de construire la liste (infinie) des entiers naturels consécutifs à partir d'un certain rang s'écrit :

$$P = \{\mathsf{LN}(x, [x|l]) \leftarrow \mathsf{LN}(S(x), l)\} \tag{4.1}$$

En effet, pour tout entier naturel k, on peut construire une dérivation infinie à partir de la requête $\mathsf{LN}(k, l)$ qui fournit à chaque étape i une approximation $[k, S(k), \cdots, S^{i-1}(k)|l_i]$ de la liste des entiers naturels consécutifs à partir de k (voir tableau 4.1). Cependant, toutes les dérivations infinies ne correspondent pas nécessairement à la construction d'un tel objet : certaines dérivations «fuient». C'est typiquement le cas avec le programme :

$$P = \{p(x) \leftarrow p(x) \ ; \ q(f(x)) \leftarrow\} \tag{4.2}$$

qui, à partir de la requête $p(z)$, donne lieu à la dérivation infinie :

$$p(z) \xrightarrow{\theta_1 = \begin{bmatrix} x_1 \\ z \end{bmatrix}}_P p(z) \xrightarrow{\theta_2 = \begin{bmatrix} x_2 \\ z \end{bmatrix}}_P \cdots \xrightarrow{\theta_i = \begin{bmatrix} x_i \\ z \end{bmatrix}}_P p(z) \xrightarrow{\theta_{i+1} = \begin{bmatrix} x_{i+1} \\ z \end{bmatrix}}_P \cdots \tag{4.3}$$

Si, à première vue, cette dérivation semble n'effectuer aucun calcul «utile» (elle ne construit aucun objet), l'identification «dérivation $\equiv$ preuve» peut conduire à interpréter la dérivation (4.3) comme une preuve infinie portant sur un objet fini (une preuve de $\forall x\, p(x)$ par exemple).

$$
\begin{array}{ll}
\mathsf{LN}(k,l) & \\
\quad\downarrow & \theta_1 = \begin{bmatrix} x_1 & l \\ k & [k|l_1] \end{bmatrix} \\
\mathsf{LN}(S(k),l_1) & \\
\quad\downarrow & \theta_2 = \begin{bmatrix} x_2 & l_1 \\ S(k) & [S(k)|l_2] \end{bmatrix} \\
\mathsf{LN}(S^2(k),l_2) & \\
\quad\downarrow & \\
\quad\vdots & \\
\quad\downarrow & \\
\mathsf{LN}(S^{i-1}(k),l_{i-1}) & \\
\quad\downarrow & \theta_i = \begin{bmatrix} x_i & l_{i-1} \\ S^{i-1}(k) & [S^{i-1}(k)|l_i] \end{bmatrix} \\
\mathsf{LN}(S^i(k),l_i) & \\
\quad\downarrow & \\
\quad\vdots &
\end{array}
$$

TAB. 4.1 – *Construction de la liste des entiers consécutifs $\geq k$*

D'un point de vue plus théorique, l'étude des dérivations infinies est nécessaire si l'on veut donner un sens à tous les atomes de la base de Herbrand $At_{\Sigma,\Pi}[\varnothing]$. En effet, nous avons vu que le plus petit modèle de Herbrand d'un programme P, noté $\mathcal{M}_P$, coïncide avec l'ensemble des succès de P, noté S_P, et correspond exactement au plus petit point fixe de l'opérateur T_P de $2^{At_{\Sigma,\Pi}[\varnothing]}$ dans $2^{At_{\Sigma,\Pi}[\varnothing]}$, associé à P.

$$\mathcal{M}_P = T_P^{\uparrow\omega} = \bigcup_{n\geq 0} T_P^{\uparrow n} = \mathsf{lfp}(T_P) = S_P$$

D'autre part, le complément de $T_P^{\downarrow\omega} = \cap_{n\geq 0} T_P^n(At_{\Sigma,\Pi}[\varnothing])$ dans la base de Herbrand correspond exactement à l'ensemble des échecs finis de P. L'étude des dérivations infinies semble donc nécessaire pour décrire les atomes de $T_P^{\downarrow\omega}\backslash T_P^{\uparrow\omega}$. Aussi, les dérivations infinies apparaissent naturellement dans la littérature consacrée à la programmation logique lors de l'étude de la sémantique de la négation. Une propriété classique, introduite par J.L. Lassez et M.J. Maher [65], concerne l'équité d'une dérivation. Dans sa forme la plus simple, cette propriété est définie comme suit.

Définition 4.1 (Equité [67]) *Une dérivation est équitable si elle échoue ou si pour tout atome A y apparaissant, A ou l'un de ses résidus (i.e. une version instanciée de A) est sélectionné en un nombre fini d'étapes.*

Cette propriété est qualifiée de propriété infinitaire puisque l'équité d'une dérivation ne peut être établie en temps fini. Il est possible de montrer qu'un atome à partir duquel on peut construire une dérivation équitable qui n'échoue pas ne peut être la racine d'un d'arbre SLD d'échec. L'équité sert, par exemple, à montrer le lemme suivant ($[A]$ désigne l'ensemble des instances fermées de A dans $At_{\Sigma,\Pi}[\varnothing]$) :

Lemme 4.1 ([5]) *Soit P un programme défini et G le but $\leftarrow A_1, \cdots, A_m$. Si il existe une (SLD-)dérivation infinie équitable à partir de G dont la séquence d'unificateurs est $\theta_0, \theta_1, \cdots$, alors :*

$$\forall k \geq 0 \quad \exists n \geq 0 \quad \bigcup_{1 \leqslant i \leqslant m} [\theta_n \cdots \theta_0 A_i] \subseteq T_P^{\downarrow k}$$

4.1 Calculabilité à l'infini

Plusieurs approches ont été proposées pour prendre en compte les dérivations infinies engendrées par un programme à partir d'une requête. Les plus connues reposent sur la notion de plus grand point fixe d'un opérateur T_P associé au programme «sur et dans» une base de Herbrand «complétée» comportant des éléments éventuellement infinis. La présence d'éléments infinis dans l'univers du discours est généralement motivée par le fait qu'une dérivation infinie doit «calculer» un objet infini (si tel n'est pas le cas, la dérivation infinie n'est pas considérée et résulte d'un «mauvais programme»), il s'agit alors seulement de donner une caractérisation déclarative des objets infinis calculables à l'infini par les SLD-dérivations. D'autre part, les complétions proposées pour la base de Herbrand visent à obtenir l'égalité $\mathsf{gfp}(T_P) = T_P^{\downarrow\omega}$. Mais si le domaine de l'opérateur T_P ne contient pas d'éléments infinis, cette égalité n'est pas vérifiée pour tous les programmes. Considérons, par exemple, le programme :

$$P = \{p(0) \leftarrow q(x) \ ; \ q(S(x)) \leftarrow q(x)\} \tag{4.4}$$

Si le domaine de l'opérateur T_P est l'ensemble des parties de la base de Herbrand $At_{\Sigma,\Pi}[\varnothing]$, alors on a :

$$T_P^{\downarrow\omega} = \bigcap_{n \geq 0} T_P^{\downarrow n} = \bigcap_{n \geq 0} \left(\bigcup_{i \geq n} \{q(S^i(0))\} \cup \{p(0)\} \right) = \{p(0)\}$$

et il vient $T_P(T_P^{\downarrow\omega}) = \varnothing = \mathsf{gfp}(T_P)$. Il existe toutefois certains programmes pour lesquels l'égalité $\mathsf{gfp}(T_P) = T_P^{\downarrow\omega}$ est vérifiée lorsque le domaine de T_P est $2^{At_{\Sigma,\Pi}[\varnothing]}$. Cette classe de programmes, appelés programmes canoniques, a été étudiée par J. Jaffar et P.J. Stuckey [48, 49]. Si, par contre, on considère un domaine «complété» satisfaisant de «bonnes» propriétés et pouvant contenir des éléments infinis, alors il vient :

$$\begin{aligned} T_P^{\downarrow\omega} &= \bigcap_{n \geq 0} T_P^{\downarrow n} = \bigcap_{n \geq 0} \left(\bigcup_{i \geq n} \{q(S^i(0))\} \cup \{p(0), q(S^\omega)\} \right) \\ &= \{p(0), q(S^\omega)\} = \mathsf{gfp}(T_P) \end{aligned}$$

Aussi, les caractérisations des objets calculés lors d'une dérivation infinie, utilisant la notion de plus grand point fixe, s'obtiendront en complétant la base de Herbrand de manière à disposer d'atomes infinis. La nouvelle base de Herbrand complétée, notée $At^{C}_{\Sigma,\Pi}[\varnothing]$, peut être obtenue en utilisant différentes techniques de complétions : la plus classique, proposée par M.A. Nait Abdallah [2], correspond à la complétion métrique, tandis que W.G. Golson [38] utilise une complétion par idéaux qui permet de munir la base de Herbrand complétée d'une structure de CPO (*complete partial order*). Les ensembles ainsi complétés satisfont de «bonnes» propriétés qui permettent d'établir la $\downarrow$-continuité d'un opérateur T_P. Le théorème 1.4 permet alors d'établir $\mathsf{gfp}(T_P) = T_P^{\downarrow\omega}$. Comme nous le verrons, une des critiques adressées à l'approche métrique provient de l'incomplétude de la sémantique proposée. Aussi, nous présenterons dans ce paragraphe les travaux de J. Hein qui tente de contourner ce problème d'incomplétude. Nous évoquerons aussi la programmation logique avec contraintes sur le domaine des arbres infinis. Toutes les approches présentées dans ce paragraphe pour caractériser les atomes infinis calculés par une dérivation, en utilisant une notion de plus grand point fixe d'un opérateur associé au programme, sont incomplètes : il existe des atomes infinis présents dans l'ensemble dénoté par le programme qui ne sont pas constructibles par une dérivation. Pour finir, nous présenterons donc une approche basée sur la notion de fermeture topologique du plus petit point fixe d'un opérateur T_P qui, de manière intuitive, semble mieux capturer la notion de calculabilité à l'infini, en considérant les objets infinis effectivement construits comme la limite d'une suite d'approximations. Hélas, nous verrons que cette approche est aussi incomplète.

4.1.1 Approche métrique

L'approche métrique a été introduite par M.A. Nait Abdallah [2, 3] en 1984 et reprise ensuite par J.W. Lloyd dans [67]. Il s'agit d'une généralisation de la sémantique opérationnelle des programmes définis (i.e. de l'ensemble des succès) aux atomes infinis basée sur une notion de distance.

Définition d'un espace métrique

Un terme peut être représenté par un arbre en numérotant les arcs issus de chaque noeud de gauche à droite à partir de 1. On accède alors à un noeud de l'arbre en parcourant un chemin qui définit un mot de $\mathbb{N}^\star$. Un terme t est donc défini par un ensemble de mots, l'ensemble $\mathcal{O}(t)$ des occurrences de t (parfois appelé domaine de l'arbre t), et peut ainsi être vu comme une fonction partielle $t : \mathcal{O}(t) \subseteq \mathbb{N}^\star \rightarrow \Sigma \cup X$.

Définition 4.2 (Termes) *Un terme t est une fonction partielle de $\mathbb{N}^\star$ dans $\Sigma \cup X$ telle que :*

1. $\forall u, v \in \mathbb{N}^\star \quad uv \in \mathcal{O}(t) \Rightarrow u \in \mathcal{O}(t)$ *(fermeture préfixe)*

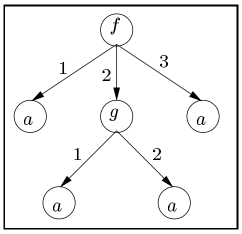


FIG. 4.1 – *Représentation du terme $f(a, g(a, a), a)$*

2. *$\forall u \in \mathbb{N}^\star$, si $u \in \mathcal{O}(t)$ et $ar(t(u)) = m$, alors $\forall i \in \mathbb{N}\backslash\{0\}$, si $i \leqslant m$ alors $ui \in \mathcal{O}(t)$, sinon $ui \notin \mathcal{O}(t)$.*

Par exemple, le domaine (i.e l'ensemble des occurrences) du terme $t = f(a, g(a, a), a)$, représenté sur la figure 4.1, est $\mathcal{O}(t) = \{\varepsilon, 1, 2, 3, 21, 22\}$, où ε désigne le mot vide. La représentation par extension de ce terme est donc l'ensemble des couples :

$$\{(\varepsilon, f), (1, a), (2, g), (3, a), (21, a), (22, a)\}$$

Un terme (i.e. un arbre) t est fini si et seulement si $\mathcal{O}(t)$ est fini. Pour un terme t et un entier n, on définit un opérateur de troncature (à la profondeur n) τ_n par :

$$\tau_n(t) = t_{|\tau_n(\mathcal{O}(t))}$$

où $t_{|E}$ est la restriction de t à E et où :

$$\tau_n(\mathcal{O}(t)) = \{u \in \mathcal{O}(t),\ \lg(u) \leqslant n\}$$

Cet opérateur va permettre de munir l'ensemble des arbres (i.e. des termes) d'une fonction de distance. Rappelons qu'étant donné un ensemble E, une fonction $d : E^2 \to \mathbb{R}^+$ définit une distance sur E si elle vérifie les trois propriétés suivantes :

$$\begin{array}{ll} \forall (x, y) \in E^2 & d(x, y) = 0 \Leftrightarrow x = y \\ \forall (x, y) \in E^2 & d(x, y) = d(y, x) \\ \forall (x, y, z) \in E^3 & d(x, z) \leqslant d(x, y) + d(y, z) \end{array}$$

L'ensemble sur lequel est définie une telle fonction est un espace métrique. De plus, si la fonction de distance vérifie la propriété :

$$\forall (x, y, z) \in E^3 \qquad d(x, z) \leqslant \max(d(x, y), d(y, z))$$

on dit que d est ultra-métrique.

Définition 4.3 (Distance sur les termes) *La distance entre deux termes t_1 et t_2 est définie par :*

$$d(t_1, t_2) = \left\{ \begin{array}{cl} 0 & si\ t_1 = t_2 \\ \frac{1}{2^{\inf\{n | \tau_n(t_1) \neq \tau_n(t_2)\}}} & sinon \end{array} \right.$$

Proposition 4.1 ([22]) *L'ensemble des termes muni de la fonction de distance d est un espace (ultra-)métrique.*

PREUVE. Etant donnés deux termes t_1 et t_2, on a clairement $d(t_1,t_2) = 0 \Leftrightarrow t_1 = t_2$ et $d(t_1,t_2) = d(t_2,t_1)$. Soit t_3 un troisième terme tel que :

$$d(t_1,t_2) = \frac{1}{2^n} \qquad d(t_1,t_3) = \frac{1}{2^m} \qquad d(t_3,t_2) = \frac{1}{2^k}$$

Par définition, on a $\tau_{m-1}(t_1) = \tau_{m-1}(t_3)$ et $\tau_{k-1}(t_3) = \tau_{k-1}(t_2)$. En posant $i = \min(m,k)$, il vient $\tau_{i-1}(t_1) = \tau_{i-1}(t_3) = \tau_{i-1}(t_2)$. On a donc $i \leqslant n$ et on peut conclure puisque :

$$d(t_1,t_2) = \frac{1}{2^n} \leqslant \frac{1}{2^{\min(m,k)}} \leqslant \frac{1}{2^n} + \frac{1}{2^k} = d(t_1,t_3) + d(t_3,t_2)$$

En remarquant que :

$$\frac{1}{2^{\min(m,k)}} = \max(\frac{1}{2^m}, \frac{1}{2^k})$$

on peut établir que d définit une distance ultra-métrique puisque d vérifie alors la propriété $d(t_1,t_2) \leqslant \max(d(t_1,t_3), d(t_3,t_2))$. ◀

D'autre part, l'ensemble $T^{\infty}_{\Sigma}[X]$ des termes, finis et infinis, correspond à la complétion métrique de $T_{\Sigma}[X]$ puisque l'on montre que :

Proposition 4.2 ([22]) $(T^{\infty}_{\Sigma}[X], d)$ *est un espace métrique complet.*

Interprétations de Herbrand

La fonction de distance sur les termes est étendue aux atomes. La base de Herbrand $At^C_{\Sigma,\Pi}[\varnothing]$ considérée correspond à la complétion métrique de $At_{\Sigma,\Pi}[\varnothing]$: $At^C_{\Sigma,\Pi}[\varnothing]$ est obtenu en complétant $At_{\Sigma,\Pi}[\varnothing]$ par ajout des limites des suites de Cauchy[1] de $At_{\Sigma,\Pi}[\varnothing]$. Une interprétation de Herbrand est maintenant un sous-ensemble de $At^C_{\Sigma,\Pi}[\varnothing]$. On obtient ainsi un univers du discours, muni d'une structure d'espace métrique complet, comportant des éléments infinis. Le domaine et le co-domaine de l'opérateur T_P associé à un programme P sont étendus à $2^{At^C_{\Sigma,\Pi}[\varnothing]}$. Cet opérateur est $\uparrow$-continu et $\downarrow$-continu et vérifie donc :

$$\mathsf{lfp}(T_P) = T_P^{\uparrow\omega} \qquad \mathsf{gfp}(T_P) = T_P^{\downarrow\omega}$$

1. Une suite $(u_n)_{n\geq 0}$ est une suite de Cauchy si :

$$\forall \varepsilon > 0 \quad \exists N \quad \forall p \quad \forall q \quad (p > N \wedge q > N) \Rightarrow d(u_p, u_q) < \varepsilon$$

Sémantique des dérivations infinies

Etant donnée une dérivation d (potentiellement infinie) :

$$R_0 \overset{C_1,\theta_1}{\to_P} R_1 \overset{C_2,\theta_2}{\to_P} \cdots \overset{C_i,\theta_i}{\to_P} R_i \overset{C_{i+1},\theta_{i+1}}{\to_P} \cdots$$

On note $d_{|i}$ la dérivation finie $R_0 \overset{*}{\to} R_i$. L'ensemble $[\![d_{|i}(R_0)]\!]$ des atomes calculés par une dérivation finie est l'ensemble des instances fermées de $\theta_i \cdots \theta_1 R_0$ dans $At^C_{\Sigma,\Pi}[\varnothing]$. Pour une dérivation infinie d, on définit :

$$[\![d(R_0)]\!] = \bigcap_{i \in \mathbb{N}} [\![d_{|i}(R_0)]\!]$$

Puisque $At^C_{\Sigma,\Pi}[\varnothing]$ est muni d'une structure d'espace métrique complet et puisque $([\![d_{|i}(R_0)]\!])_{i\in\mathbb{N}}$ est une suite décroissante d'ensembles fermés (topologiquement), cet ensemble n'est jamais vide (ce qui ne serait pas le cas si l'on considérait les instances fermées $[d(R_0)]$ dans $At_{\Sigma,\Pi}[\varnothing]$). Les principaux résultats issus de [2] s'énoncent :

Théorème 4.1 ([2])

1. *Pour tout atome A, $[\![A]\!] \cap T_P^{\downarrow\omega} = \bigcup\{[\![d(A)]\!],\ d$ est équitable$\}$.*
2. *Il existe une dérivation finie de succès à partir de $A \in At^C_{\Sigma,\Pi}[\varnothing]$ si et seulement si $A \in T_P^{\uparrow\omega}$.*
3. *Il existe une dérivation finie d'échec à partir de $A \in At^C_{\Sigma,\Pi}[\varnothing]$ si et seulement si $A \notin T_P^{\downarrow\omega}$.*
4. *Il existe une dérivation équitable à partir de $A \in At^C_{\Sigma,\Pi}[\varnothing]$ si et seulement si $A \in T_P^{\downarrow\omega}$.*

La quatrième assertion de ce théorème a l'allure d'un résultat de complétude. Toutefois, il est important de noter que l'atome considéré peut être infini et ne correspond donc pas nécessairement à un atome à partir duquel on construit une dérivation (le langage des requêtes ne contient pas d'éléments infinis).

Dans le chapitre de son livre [67] consacré aux «processus perpétuels», J.W. Lloyd utilise l'approche métrique pour introduire la notion d'atomes calculables à l'infini comme suit.

Définition 4.4 (Atomes calculables à l'infini [67]) *Soit P un programme défini et A un atome de $At^C_{\Sigma,\Pi}[\varnothing] \backslash At_{\Sigma,\Pi}[\varnothing]$ (i.e. un atome infini). A est calculable à l'infini si il existe un atome fini B et une dérivation infinie équitable, dont la séquence d'unificateurs est $\theta_1, \cdots, \theta_i, \cdots$, à partir de B telle que :*

$$\lim_{n\to\infty} d(A, \theta_n \cdots \theta_1 B) = 0$$

On note C_P l'ensemble des atomes calculables à l'infini à partir d'un programme P :

$$C_P = \{A \in At^C_{\Sigma,\Pi}[\varnothing] \backslash At_{\Sigma,\Pi}[\varnothing],\ A \text{ est calculable à l'infini}\}$$

Par exemple, pour le programme défini (4.1), on a :

$$C_P = \bigcup_{k \in \mathbb{N}} \{\mathsf{LN}(k, [k, S(k), S^2(k), \cdots, S^i(k), \cdots])\}$$

En effet, pour tout entier naturel k, on peut construire une dérivation infinie équitable, à partir de l'atome fini $\mathsf{LN}(k, l)$, telle que :

$$\begin{array}{ll} & \lim\limits_{n\to\infty} d(\mathsf{LN}(k, [k, S(k), \cdots, S^i(k), \cdots]), \theta_n \cdots \theta_1 \mathsf{LN}(k, l)) \\ = & \lim\limits_{n\to\infty} d(\mathsf{LN}(k, [k, S(k), \cdots, S^i(k), \cdots]), \mathsf{LN}(k, [k, S(k), \cdots, S^{n-1}(k)|l_n])) \\ = & 0 \end{array}$$

Le lemme qui suit, indépendant de la définition considérée pour la distance, permet d'établir le théorème de validité.

Lemme 4.2 ([67]) *Soit P un programme et $A \in At^C_{\Sigma,\Pi}[\varnothing] \backslash At_{\Sigma,\Pi}[\varnothing]$. A appartient à C_P si et seulement si il existe un atome fini B et une dérivation infinie équitable d à partir de B telle que $[\![d(B)]\!] = \{A\}$.*

Par exemple, si l'on considère la dérivation que l'on peut construire à partir de la requête $\mathsf{LN}(k, l)$ avec le programme défini en (4.1), on a bien :

$$\begin{array}{rcl} [\![d(\mathsf{LN}(k, l))]\!] & = & \bigcap\limits_{n \in \mathbb{N}} [\![\mathsf{LN}(k, [k, S(k), \cdots, S^{n-1}(k)|l_n])]\!] \\ & = & \{\mathsf{LN}(k, [k, S(k), \cdots, S^i(k), \cdots])\} \end{array}$$

Théorème 4.2 (Validité [67]) *Pour tout programme P, $C_P \subseteq \mathsf{gfp}(T_P)$.*

C_P ne contenant que des atomes infinis, on a même $C_P \subseteq \mathsf{gfp}(T_P) \backslash At_{\Sigma,\Pi}[\varnothing]$.

Deux critiques sont traditionnellement adressées à l'approche métrique pour la prise en compte des dérivations infinies. Tout d'abord, le plus grand point fixe de l'opérateur T_P (dont le domaine est étendu aux parties de la base de Herbrand complétée) ne caractérise pas uniquement les atomes effectivement construits lors d'une dérivation infinie à partir du programme P. Considérons par exemple le programme défini (4.2), il est clair que l'ensemble des atomes calculables à l'infini à partir de ce programme est vide. Cependant, le plus grand point fixe de l'opérateur associé à ce programme est :

$$\mathsf{gfp}(T_P) = \{p(t), q(f(t)),\ t \text{ est un terme fermé}\}$$

Le seul terme fermé que l'on peut construire à partir des symboles apparaissant dans P étant f^ω, il vient $\mathsf{gfp}(T_P) = \{p(f^\omega), q(f^\omega)\}$ et donc

$\mathsf{gfp}(T_P) \not\subseteq C_P = \varnothing$. En effet, comme nous l'avons déjà suggéré, la dérivation (4.3) n'effectue aucun calcul, et, en particulier, elle ne constitue pas un procédé effectif de construction de l'atome $p(f^\omega)$ même si :

$$p(f^\omega) \in \bigcup\{[\![d(p(z))]\!],\ d \text{ est équitable}\} = \bigcap_{i\in\mathbb{N}} [\![d_{|i}(p(z))]\!] = [\![p(z)]\!]$$

Le théorème 4.1 permet seulement d'affirmer :

$$\{p(f^\omega)\} = \bigcup\{[\![d(p(f^\omega))]\!],\ d \text{ est équitable}\}$$

Or, le langage des requêtes ne contient pas d'éléments infinis. La deuxième critique adressée à l'approche métrique provient du fait que deux suites distinctes peuvent converger vers la même limite. Considérons par exemple les deux programmes suivants :

$$\begin{array}{lccccccc} P_1 = \{ & p(f(x), f(f(x))) & \leftarrow & p(x, f(x)) & ; & q(x) & \leftarrow & p(x, f(x)) \quad \} \\ P_2 = \{ & p(f(x), f(x)) & \leftarrow & p(x, x) & ; & q(x) & \leftarrow & p(x, f(x)) \quad \} \end{array} \tag{4.5}$$

Même si $\mathsf{gfp}(T_{P_1}) = \mathsf{gfp}(T_{P_2}) = \{p(f^\omega, f^\omega), q(f^\omega)\}$, la requête $q(x)$ donne lieu à une dérivation infinie équitable avec le programme P_1 tandis que le test d'occurrence de l'algorithme d'unification ne permet pas de construire une dérivation à partir de $q(x)$ avec le programme P_2. Cependant, ces deux programmes permettent chacun de construire une dérivation infinie équitable qui «calcule» $p(f^\omega, f^\omega)$. Cette dérivation s'obtient à partir de la requête $p(x, f(x))$ avec le programme P_1 et à partir de la requête $p(x, x)$ avec le programme P_2.

$$p(x, f(x)) \xrightarrow{\theta_1 = \left[\begin{array}{c} x \\ f(x_1) \end{array}\right]}_{P_1} p(x_1, f(x_1)) \rightarrow_{P_1} \cdots \xrightarrow{\theta_i = \left[\begin{array}{c} x_{i-1} \\ f(x_i) \end{array}\right]}_{P_1} p(x_i, f(x_i)) \rightarrow_{P_1} \cdots \tag{4.6}$$

$$p(x, x) \xrightarrow{\theta_1 = \left[\begin{array}{c} x \\ f(x_1) \end{array}\right]}_{P_2} p(x_1, x_1) \rightarrow_{P_2} \cdots \xrightarrow{\theta_i = \left[\begin{array}{c} x_{i-1} \\ f(x_i) \end{array}\right]}_{P_2} p(x_i, x_i) \rightarrow_{P_2} \cdots \tag{4.7}$$

Pour $q(f^\omega)$, la situation est différente : la requête $q(x)$ permet d'obtenir une dérivation infinie équitable avec le programme P_1 qui construit $q(f^\omega)$, mais il n'existe pas d'atome fini à partir duquel on puisse construire $q(f^\omega)$ dans une dérivation infinie avec le programme P_2 : il faut obligatoirement partir de la requête constituée de l'atome infini $q(f^\omega)$ avec P_2 (et donc autoriser la présence de termes infinis dans une requête) si l'on souhaite obtenir une dérivation infinie équitable d telle que $[\![d(q(f^\omega))]\!] = q(f^\omega)$. Les ensembles d'atomes calculables à l'infini C_{P_1} et C_{P_2} diffèrent donc :

$$C_{P_1} = \{p(f^\omega, f^\omega), q(f^\omega)\} \qquad C_{P_2} = \{p(f^\omega, f^\omega)\} \tag{4.8}$$

Nous verrons que le phénomène d'incomplétude de l'approche métrique provient du fait que l'opérateur T_P considéré correspond à l'opérateur associé à l'ensemble de règles $[\![P]\!]$ (l'ensemble des instances fermées des clauses de P dans $At^C_{\Sigma,\Pi}[\varnothing]$).

Enfin, signalons que l'approche métrique n'utilise pas les treillis lors de l'étude de la sémantique des programmes définis. L'utilisation d'espaces métriques complets permet notamment de garantir l'existence d'un unique point fixe pour l'opérateur T_P lorsque celui-ci définit une contraction : étant donné un espace métrique E, une application $f : E \rightarrow E$ est une contraction si il existe un nombre k $(0 \leqslant k < 1)$ tel que :

$$\forall x, y \in E \qquad d(f(x), f(y)) \leqslant k.d(x, y)$$

Puisque, dans un espace métrique complet, une contraction admet un unique point fixe, une approche qui n'utilise pas le théorème de Tarski (qui, par exemple, ne peut pas s'appliquer en présence de négations dans les programmes puisque dans ces cas, T_P n'est pas un opérateur monotone) consiste à définir une distance telle que T_P soit une contraction pour cette distance. Cette idée est suggérée et illustrée par M. Fitting dans [33].

4.1.2 Programmes partiellement complets

Puisque, lorsque l'on considère un univers du discours pouvant contenir des éléments infinis, la SLD-résolution n'est pas complète, J. Hein propose dans [45], d'une part, une caractérisation de la classe des programmes logiques pour lesquels la SLD-résolution est complète (y compris pour les calculs infinis), et, d'autre part, une méthode, permettant de contourner partiellement le problème d'incomplétude, basée sur la complétion d'un programme P en $Comp(P)$. Le cadre dans lequel se place la sémantique par complétion de programmes est similaire à celui de l'approche métrique. La base de Herbrand considérée $At^C_{\Sigma,\Pi}[\varnothing]$ est l'ensemble des atomes finis ou infinis fermés et l'opérateur associé à un programme P est l'opérateur classique T_P dans $2^{At^C_{\Sigma,\Pi}[\varnothing]}$.

Définition 4.5 (Atomes calculables à l'infini [45]) *Soit P un programme et A un atome de $At^C_{\Sigma,\Pi}[\varnothing] \backslash At_{\Sigma,\Pi}[\varnothing]$ (i.e. un atome infini). A est calculable à l'infini ($A \in C_P$) si il existe un atome fini B et une dérivation infinie équitable, dont la séquence d'unificateurs est $\theta_1, \cdots, \theta_i, \cdots$, à partir de B telle que pour tout entier $n \geq 0$, il existe un entier $i > n$ tel que A et $\theta_i \cdots \theta_1 B$ coïncident jusqu'à la profondeur n.*

Cette définition correspond exactement à la définition 4.4 et, d'après le théorème 4.2, on a donc $C_P \subseteq \mathsf{gfp}(T_P) \backslash At_{\Sigma,\Pi}[\varnothing]$.

Complétions parfaites

Dans ce qui suit, on dit qu'un programme Q est une extension de P si $P \subseteq Q$. D'autre part, un programme Q est dit complet relativement à un programme P lorsque :

$$C_Q = \mathsf{gfp}(T_P) \backslash At_{\Sigma,\Pi}[\varnothing]$$

Enfin, un programme P est complet s'il est complet relativement à lui même. C'est, par exemple, le cas du programme défini en (4.9) qui vérifie $C_P = \{p(f^\omega)\}$, mais ce n'est pas le cas du programme défini en (4.2). Bien sûr, la complétion d'un programme P peut correspondre à un programme lui-même non complet, la seule propriété que l'on cherche à obtenir est :

$$C_{Comp(P)} = \mathsf{gfp}(T_P) \backslash At_{\Sigma,\Pi}[\varnothing]$$

Un programme admet potentiellement plusieurs complétions. On dira qu'un programme Q est une complétion parfaite de P si Q est une complétion de P satisfaisant :

1. $\mathcal{M}_P = \mathcal{M}_Q$
2. $\mathsf{gfp}(T_P) = \mathsf{gfp}(T_Q)$
3. P et Q admettent le même ensemble d'échecs finis

Une complétion parfaite de P est donc un programme complet. Considérons, par exemple, le programme :

$$P = \{p(x, f(x)) \leftarrow p(x, x)\}$$

Ce programme n'est pas complet puisque $\mathsf{gfp}(T_P) \backslash At_{\Sigma,\Pi}[\varnothing] = \{p(f^\omega, f^\omega)\}$ alors que C_P est vide. Le programme Q suivant est une complétion pour P :

$$Q = \{p(x, f(x)) \leftarrow p(x, x) \ ; \ p(f(x), f(f(x))) \leftarrow p(x, f(x))\}$$

puisqu'on a alors $C_Q = \mathsf{gfp}(T_P) \backslash At_{\Sigma,\Pi}[\varnothing]$. Ce programme Q est lui-même complet et constitue clairement une complétion parfaite de P puisque les ensembles des succès (resp. des échecs) finis de P et de Q sont vides.

Complétion partielle des clauses unités

Il semble possible de construire la complétion d'un programme de manière incrémentale par approximations à l'aide d'une suite de complétions partielles. Une complétion partielle d'un programme P, appelée dans la suite extension parfaite, est une extension Q de P satisfaisant la propriété $C_Q \subset \mathsf{gfp}(T_P) \backslash At_{\Sigma,\Pi}[\varnothing]$ (tout programme est donc une complétion partielle de lui-même). La complétion envisagée dans [45], appelée complétion des clauses unités, consiste à construire pour tout programme P une extension parfaite Q

de P telle que C_Q contienne tous les atomes fermés de $\mathsf{gfp}(T_P)\backslash At_{\Sigma,\Pi}[\varnothing]$ correspondant aux instances fermées infinies des têtes des clauses unités de P. Si $A \leftarrow$ est une clause unité d'un programme P, on définit l'ensemble de clauses $C(A)$ par :

$$C(A) = \{A \leftarrow\} \cup C_1(A) \cup C_2(A) \cup C_3(A)$$

où les ensembles $C_i(A)$ sont définis par :

- ($C_1(A)$) Pour chaque variable x apparaissant dans $A \leftarrow$ et pour chaque symbole de fonction f apparaissant dans P, $C_1(A)$ contient une clause de la forme :

$$\begin{bmatrix} x \\ f(y_1, \cdots, y_n) \end{bmatrix} A \leftarrow \begin{bmatrix} x \\ y_1 \end{bmatrix} A, \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} A, \cdots, \begin{bmatrix} x_n \\ y_n \end{bmatrix} A$$

  Les ensembles de variables apparaissant dans les atomes du corps de cette clause sont disjoints et toutes les variables apparaissant dans la tête de cette clause apparaissent dans un atome de son corps.
- ($C_2(A)$) Si P contient au moins un symbole de fonction d'arité supérieure ou égale à 2, alors pour chaque variable x apparaissant dans $A \leftarrow$ et pour chaque symbole de constante c apparaissant dans P (s'il n'y a pas de constante dans P, on en introduit une), $C_2(A)$ contient une clause de la forme :

$$\begin{bmatrix} x \\ c \end{bmatrix} A \leftarrow$$

- ($C_3(A)$) Si P contient au moins un symbole de fonction d'arité supérieure ou égale à 2 et si $A \leftarrow$ contient au moins deux symboles de variables distincts, alors pour chaque variable x apparaissant dans $A \leftarrow$ et pour chaque symbole de constante c apparaissant dans P (s'il n'y a pas de constante dans P, on en introduit une), $C_3(A)$ contient une clause de la forme :

$$\begin{bmatrix} x \\ c \end{bmatrix} A \leftarrow \begin{bmatrix} x \\ c \end{bmatrix} A$$

L'ensemble $C(A)$ ainsi construit satisfait le lemme :

Lemme 4.3 ([45]) *Si $A \leftarrow$ est une clause unité d'un programme P, alors $P \cup C(A)$ est une extension parfaite de P.*

Pour tout atome A, $IG(A)$ dénote l'ensemble des atomes fermés appartenant à $\mathsf{gfp}(T_P)\backslash At_{\Sigma,\Pi}[\varnothing]$ et qui sont unifiables avec A. Le résultat final de J. Hein s'énonce alors :

Théorème 4.3 ([45]) *Si $A \leftarrow$ est une clause unité d'un programme P, alors $IG(A) \subset C_{P \cup C(A)}$.*

La preuve donnée pour ce théorème dans [45] est constructive : elle est présentée sous la forme d'un algorithme qui, étant donnés un programme P, une clause unité $A \leftarrow$ de P et un atome $\sigma A \in IG(A)$, construit une dérivation infinie équitable à partir du programme $P \cup C(A)$ qui «calcule à l'infini» σA. Bien sûr, l'auteur prouve que cet algorithme est correct. Ce résultat permet de limiter partiellement l'incomplétude de la SLD-résolution pour les calculs infinis. Considérons, par exemple, le programme défini (4.2) pour lequel $\mathsf{gfp}(T_P) \backslash At_{\Sigma,\Pi}[\varnothing] = \{p(f^\omega), q(f^\omega)\}$ et $C_P = \varnothing$. La seule clause unité de ce programme est $q(f(x)) \leftarrow$ et il est donc possible de construire l'ensemble $C(q(f(x)))$ comme suit :

$$\begin{aligned} C(q(f(x))) = &\ \{q(f(x)) \leftarrow\} \cup C_1(q(f(x))) \cup C_2(q(f(x))) \cup C_3(q(f(x))) \\ = &\ \{q(f(x)) \leftarrow\} \cup \{q(f(f(y))) \leftarrow q(f(y))\} \cup \varnothing \cup \varnothing \end{aligned}$$

On peut alors construire à partir de $q(x)$ une dérivation infinie équitable qui construit $q(f^\omega) \in IG(q(f(x)))$ avec le programme $P \cup C(q(f(x)))$:

$$q(x) \overset{\theta_1 = \left[\begin{array}{c} x \\ f(f(y_1)) \end{array}\right]}{\longrightarrow_P} q(f(y_1)) \overset{\theta_2 = \left[\begin{array}{c} y_1 \\ f(y_2) \end{array}\right]}{\longrightarrow_P} \cdots \overset{\theta_i = \left[\begin{array}{c} y_{i-1} \\ f(y_i) \end{array}\right]}{\longrightarrow_P} q(f(y_i)) \to_P \cdots$$

Toutefois, $p(f^\omega)$ reste «inaccessible» puisqu'il n'existe aucune clause unité dont le symbole de prédicat de tête de clause est p.

Programmes complets

La technique de complétion des clauses unités d'un programme peut servir de point de départ pour la caractérisation de la classe des programmes complets. Les résultats issus de [45] s'énoncent :

Théorème 4.4 ([45]) *Soit P un programme défini.*

1. *Si $A \leftarrow$ est une clause unité de P, alors les programmes $C(A)$ et $C(A) \backslash \{A \leftarrow\}$ sont complets.*
2. *Si P peut se décomposer en $P = K(A_1) \cup \cdots \cup K(A_n)$ (où les A_i correspondent aux têtes des clauses unités de P et où $K(A_i)$ désigne $C(A_i)$ ou $C(A_i) \backslash \{A_i \leftarrow\}$), alors P est complet.*

Ce théorème permet de montrer qu'un programme défini P est complet sans construire C_P ni $\mathsf{gfp}(T_P) \backslash At_{\Sigma,\Pi}[\varnothing]$.

L'approche développée par J. Hein ne résoud pas le problème d'incomplétude mais propose une solution qui permet de le contourner partiellement : la méthode proposée permet seulement de construire certains éléments du plus grand point fixe de T_P, «inaccessibles» avec le programme P, à partir d'une version modifiée de P. Plus précisément, étant donné un programme P, on peut construire un programme Q en ajoutant à P les clauses correspondant à la complétion des clauses unités de P. On peut alors construire à partir de Q les atomes de $\mathsf{gfp}(T_P)\backslash At_{\Sigma,\Pi}[\varnothing]$ qui sont des instances fermées infinies des têtes des clauses unités de P. Hélas, les atomes de $\mathsf{gfp}(T_P)\backslash At_{\Sigma,\Pi}[\varnothing]$ correspondant aux instances fermées infinies des têtes de clauses de P pour lesquels il n'existe pas de clauses unités dans P ne sont pas systématiquement constructibles (i.e. calculables à l'infini) à l'aide d'une dérivation infinie. Enfin, de manière plus générale, rien ne garantit qu'un programme quelconque admette une complétion parfaite.

4.1.3 Complétion par idéaux

En 1988, W.G. Golson [38] utilise les Cpo pour étudier les dérivations infinies en programmation logique. Il définit alors une méthode de complétion par idéaux de la base de Herbrand. Les propriétés mathématiques des Cpo sont détaillées dans [24].

Restriction sur les programmes

Les programmes considérés dans [38] doivent satisfaire la condition suivante : toute variable apparaissant dans le corps d'une clause doit aussi apparaître dans sa tête. Cette restriction est nécessaire pour assurer la $\downarrow$-continuité de l'opérateur T_P défini plus bas. Par exemple, en utilisant cette approche, le programme (4.4) ne pourra pas être considéré.

Complétion de la base de Herbrand

Etant donné un ensemble partiellement ordonné $(E, \preceq)$, un idéal $\mathcal{I}$ de E est un sous-ensemble de E dirigé (i.e. chaque paire d'éléments de $\mathcal{I}$ admet un majorant dans $\mathcal{I}$) et $\preceq\downarrow$-fermé (i.e. si $x_1 \in \mathcal{I}$ et si $x_2 \preceq x_1$, alors $x_2 \in \mathcal{I}$). L'ensemble des idéaux de E, ordonné par l'inclusion ensembliste $\subseteq$, est un Cpo :

- chaque chaîne d'idéaux (et, plus généralement, chaque sous-ensemble dirigé de l'ensemble des idéaux) admet une borne supérieure qui est aussi un idéal et qui représente la limite de la chaîne.
- l'ensemble des idéaux admet un plus petit élément.

La complétion de la base de Herbrand proposée dans [38] repose sur cette technique. Pour tout ensemble fini $\mathcal{A} \subseteq At_{\Sigma,\Pi}[X]$ d'atomes finis et pour toute substitution σ, on note $\sigma\mathcal{A}$ l'ensemble $\{\sigma A, A \in \mathcal{A}\}$. Le préordre $\leqslant$ sur les atomes est étendu aux ensembles d'atomes finis par :

$$\mathcal{A}_1 \leqslant \mathcal{A}_2 \Leftrightarrow \exists \sigma \quad \sigma\mathcal{A}_1 = \mathcal{A}_2$$

Lorsque $\mathcal{A}_1 \leqslant \mathcal{A}_2$, on dit que $\mathcal{A}_1$ approxime $\mathcal{A}_2$. Un lemme de représentation est établi afin de caractériser les idéaux de $At_{\Sigma,\Pi}[X]$, aussi appelés objets, par des ensembles de la forme :

$$\mathcal{A}\Theta = \{\mathcal{A}', \exists\sigma \in \Theta \quad \mathcal{A}' \leqslant \sigma\mathcal{A}\} \qquad (= \{\mathcal{A}', \exists\sigma \in \Theta\ \exists\eta \quad \eta\mathcal{A}' = \sigma\mathcal{A}\})$$

où $\mathcal{A}$ désigne un ensemble d'atomes finis et Θ désigne un ensemble dirigé de substitutions. Un objet $\mathcal{A}\Theta$ est dit infini si le cardinal de l'ensemble des classes d'équivalence (modulo un renommage) de Θ est infini. Cette représentation permet, par exemple, de distinguer les deux séquences distinctes d'approximations construites lors des dérivations obtenues à partir des programmes définis en (4.5). La dérivation (4.6) construit la séquence d'approximations $(p(f^n(x_n), f^{n+1}(x_n)))_{n\geq 0}$ qui peut être représentée par l'objet :

$$\mathcal{A}_1\Theta_1 = \{p(x, f(x))\} \left\{ \left[\begin{array}{c} x \\ f^i(x_i) \end{array} \right] \right\}_{i\geq 0} = \{p(f^n(x_n), f^{n+1}(x_n))\}_{n\geq 0}$$

alors que $(p(f^n(x_n), f^n(x_n)))_{n\geq 0}$ est la séquence d'approximations calculée par la dérivation (4.7) et peut être représentée par l'objet :

$$\mathcal{A}_2\Theta_2 = \{p(x, x)\} \left\{ \left[\begin{array}{c} x \\ f^i(x_i) \end{array} \right] \right\}_{i\geq 0} = \{p(f^n(x_n), f^n(x_n))\}_{n\geq 0}$$

L'ensemble des objets (i.e. des idéaux de $At_{\Sigma,\Pi}[X]$) constitue la base de Herbrand complétée $At^{\mathcal{C}}_{\Sigma,\Pi}[X]$. Une interprétation de Herbrand I est un sous-ensemble $\subseteq\uparrow$-fermé (i.e. si $\alpha_1 \in I$ et si $\alpha_1 \subseteq \alpha_2$, alors $\alpha_2 \in I$) de $At^{\mathcal{C}}_{\Sigma,\Pi}[X]$ et l'ensemble des interprétations de Herbrand, ordonné par l'inclusion ensembliste $\subseteq$, forme un treillis complet. Cette définition est proche de celle donnée pour les $\mathcal{C}$-interprétations, présentées dans le paragraphe 4.2.3. On définit l'ensemble des objets minimaux (i.e. les objets les plus généraux) d'une interprétation I par :

$$\min(I) = \{\alpha \in I, \forall\beta \in I \quad \beta \subseteq \alpha \Rightarrow \alpha = \beta\}$$

Sémantique des dérivations infinies

L'opérateur T_P sur les parties de l'ensemble des interprétations de Herbrand est défini comme suit :

$$T_P(I) = \{ \quad \alpha \text{ (objet)}, \quad \begin{array}{l} \exists\{A_i \leftarrow \mathcal{A}'_i\} \in P \\ \exists\Theta \quad \text{(ensemble dirigé de substitutions)} \\ \alpha = \mathcal{A}\Theta \quad (\mathcal{A} = \cup\{A_i\}) \\ \mathcal{A}'\Theta \in I \quad (\mathcal{A}' = \cup\mathcal{A}_i) \end{array} \quad \}$$

T_P est un opérateur $\downarrow$-continu et vérifie donc $\mathsf{gfp}(T_P) = T_P^{\downarrow\omega}$. Les deux principaux résultats issus de [38] s'énoncent alors :

Théorème 4.5 ([38])

1. $\alpha \in \mathsf{gfp}(T_P)$ *si et seulement si il existe une dérivation équitable à partir* *d'une «requête» $\mathcal{A}$ via les unificateurs (σ_i) telle que $\mathcal{A}\{\cup_i\{\sigma_i\}\} \subseteq \alpha$ où $\mathcal{A}$ contient des variantes distinctes de têtes de clauses de P.*

2. $\alpha \in \min(\mathsf{gfp}(T_P))$ *si et seulement si il existe une dérivation équitable à partir d'une «requête» $\mathcal{A}$ via les unificateurs (σ_i) telle que $\mathcal{A}\{\cup_i\{\sigma_i\}\} = \alpha$ où $\mathcal{A}$ contient des variantes distinctes de têtes de clauses de P.*

Par exemple, pour le programme défini (4.2), on a $\min(\mathsf{gfp}(T_P)) = \{p(z)\}\{[\,]\}$ qui n'est pas un objet infini, ce qui correspond bien au fait que la dérivation (4.3) ne construit aucun objet infini. Par contre, si l'on considère le programme :

$$P = \{p(f(x)) \leftarrow p(x)\} \tag{4.9}$$

alors on a :

$$\min(\mathsf{gfp}(T_P)) = \{p(z)\}\left\{\left[\begin{array}{c} z \\ f^i(x_i) \end{array}\right]\right\}_{i\geq 0}$$

qui est un objet infini pour lequel il existe donc une dérivation équitable qui le construit :

$$p(z) \xrightarrow{\theta_1=\left[\begin{array}{c} z \\ f(x_1) \end{array}\right]}_P p(x_1) \xrightarrow{\theta_2=\left[\begin{array}{c} x_1 \\ f(x_2) \end{array}\right]}_P \cdots \xrightarrow{\theta_i=\left[\begin{array}{c} x_{i-1} \\ f(x_i) \end{array}\right]}_P p(x_i) \xrightarrow{\theta_{i+1}=\left[\begin{array}{c} x_i \\ f(x_{i+1}) \end{array}\right]}_P \cdots \tag{4.10}$$

L'avantage de l'approche développée par W.G. Golson, par rapport à l'approche métrique, provient du fait que l'on peut, par exemple, distinguer les deux programmes P_1 et P_2 définis en (4.5). En effet, tandis que les deux objets :

$$\{p(x, f(x))\}\left\{\left[\begin{array}{c} x \\ f^i(x_i) \end{array}\right]\right\}_{i\geq 0} \qquad \text{et} \qquad \{q(x)\}\left\{\left[\begin{array}{c} x \\ f^i(x_i) \end{array}\right]\right\}_{i\geq 0}$$

sont calculables à partir de P_1, seul l'objet :

$$\{p(f(x), f(x))\}\left\{\left[\begin{array}{c} x \\ f^i(x_i) \end{array}\right]\right\}_{i\geq 0}$$

est calculable à partir de P_2, ce qui rend bien compte de la différence entre les ensembles C_{P_1} et C_{P_2} définis en (4.8).

L'approche développée dans [38] présente deux défauts principaux. Tout d'abord, il existe une restriction syntaxique sur les programmes. D'autre part, toutes les dérivations infinies ne sont pas prises en compte dans cette approche. En effet, seules les dérivations infinies permettant de calculer des objets minimaux (i.e. les plus généraux) sont caractérisées à l'aide des objets minimaux présents dans le plus grand point fixe de l'opérateur T_P. Aussi, à partir du programme :

$$P = \{p(f(x), y) \leftarrow p(x, y)\}$$

on peut construire les deux dérivations infinies équitables suivantes :

$$p(x,y) \overset{\theta_1=\begin{bmatrix} x \\ f(x_1) \end{bmatrix}}{\longrightarrow_P} p(x_1,y) \overset{\theta_2=\begin{bmatrix} x_1 \\ f(x_2) \end{bmatrix}}{\longrightarrow_P} \cdots \overset{\theta_i=\begin{bmatrix} x_{i-1} \\ f(x_i) \end{bmatrix}}{\longrightarrow_P} p(x_i,y) \longrightarrow_P \cdots \tag{4.11}$$

$$p(x,g(y)) \overset{\theta_1=\begin{bmatrix} x \\ f(x_1) \end{bmatrix}}{\longrightarrow_P} p(x_1,g(y)) \overset{\theta_2=\begin{bmatrix} x_1 \\ f(x_2) \end{bmatrix}}{\longrightarrow_P} \cdots \overset{\theta_i=\begin{bmatrix} x_{i-1} \\ f(x_i) \end{bmatrix}}{\longrightarrow_P} p(x_i,g(y)) \to_P \cdots$$

Toutefois, seule la dérivation (4.11) correspond à la construction d'un objet minimal :

$$\{p(x,y)\}\left\{\begin{bmatrix} x \\ f^i(x_i) \end{bmatrix}\right\}_{i\geq 0}$$

4.1.4 Programmation Logique avec Contraintes sur le domaine des arbres infinis

La programmation logique avec contraintes permet d'étendre le domaine du discours classique (i.e. l'univers de Herbrand pour les programmes logiques) en intégrant la possibilité d'utiliser des domaines particuliers. Les calculs sur ces domaines sont exprimés par des contraintes, c'est à dire des relations décidables portant sur un ensemble de variables à valeur dans un certain domaine. Une contrainte peut donc être représentée par une formule logique atomique qui est interprétée dans une structure particulière (et non pas nécessairement dans une interprétation de Herbrand). Les clauses d'un programme définissent alors l'aspect logique du programme tandis que les contraintes constituent l'aspect calculatoire. La règle de résolution est ainsi généralisée en remplaçant l'opération d'unification sur les termes par la résolution de contraintes mettant en jeu un algorithme de résolution propre au domaine considéré. J. Jaffar, P.J. Stuckey et J.L. Lassez proposent dans [47, 50] une approche basée sur les contraintes. Le cadre dans lequel ils se placent pour étudier les dérivations infinies est donc naturellement celui de la programmation logique avec contraintes sur le domaine des termes infinis. Le langage considéré est proche[2] du langage PROLOG II, introduit par A. Colmerauer [17] en 1982, puis étendu en PROLOG III et IV. Dans la plupart des interprètes PROLOG, le test d'occurrence de l'algorithme d'unification n'est pas effectivement réalisé. Par exemple, le but $p(x,x)$ réussit à partir du programme $P = \{p(x, f(x)) \leftarrow\}$ en produisant la réponse $\{x/f(f(f(\cdots)))\}$. Ceci permet, d'une part, de manipuler des structures de données circulaires (i.e. contenant des références croisées) et, d'autre part, d'améliorer les performances de l'algorithme d'unification. Le langage PROLOG II est basé sur un algorithme d'unification correct et complet sur les termes finis et les termes

2. les deux principales différences proviennent de la possibilité d'introduire des inéquations entre termes dans les clauses et de l'algorithme d'unification

infinis rationnels (i.e. représentables par des graphes cycliques finis, c'est à dire des termes admettant un ensemble fini de sous-termes[3]). C'est l'étude du langage PROLOG II qui a conduit, en 1986, J. Jaffar et J.L. Lassez [47] à définir la classe des langages de programmation logique avec contraintes sur un domaine $\mathcal{D}$, notée CLP($\mathcal{D}$).

Programmation logique avec contraintes

Un sytème de contraintes (fini ou infini) est un ensemble constitué d'équations de la forme $t_1 = t_2$ et d'inéquations de la forme $t_1 \neq t_2$, où t_1 et t_2 sont des termes finis. Les programmes considérés sont des ensembles finis de clauses de la forme :

$$p(\vec{x}) \leftarrow E(\vec{x},\vec{y}) \cup \bar{E}(\vec{x},\vec{y}) \mid B_1, \cdots, B_n \tag{4.12}$$

où les B_i sont des atomes et où $E(\vec{x},\vec{y}) \cup \bar{E}(\vec{x},\vec{y})$ désigne un système fini de contraintes constitué, d'une part, d'un ensemble d'équations $E(\vec{x},\vec{y})$ et, d'autre part, d'un ensemble d'inéquations $\bar{E}(\vec{x},\vec{y})$. De plus, cette clause satisfait les trois propriétés suivantes :

- $\vec{x} \cap \vec{y} = \varnothing$
- $\bigcup_{i=1}^{n} var(B_i) \subseteq \vec{y}$
- $\forall i \forall j \quad i \neq j \Rightarrow var(B_i) \cap var(B_j) = \varnothing$

Toute clause de la forme (4.12) peut être transformée en une clause satisfaisant ces propriétés. Les requêtes sont de la forme $\leftarrow S \mid B_1, \cdots, B_n$ où les B_i sont des atomes et où S est un système fini de contraintes.

Sémantique déclarative

La base de Herbrand considérée $At^C_{\Sigma,\Pi}[\varnothing]$ est l'ensemble des atomes fermés éventuellement infinis. Deux sous-ensembles de $At^C_{\Sigma,\Pi}[\varnothing]$ sont définis : $At_{\Sigma,\Pi}[\varnothing]$ désigne les atomes fermés finis de $At^C_{\Sigma,\Pi}[\varnothing]$ et $At^R_{\Sigma,\Pi}[\varnothing]$ désigne les atomes rationnels de $At^C_{\Sigma,\Pi}[\varnothing]$ (i.e. les atomes construits à partir des termes rationnels). Une interprétation de Herbrand est un sous-ensemble de $At^C_{\Sigma,\Pi}[\varnothing]$. Etant donnée une substitution σ, une équation $\sigma(t_1 = t_2)$ (resp. une inéquation $\sigma(t_1 \neq t_2)$) est vraie si $\sigma t_1 = \sigma t_2$ (resp. $\sigma t_1 \neq \sigma t_2$). Un système de contraintes σS est vrai si pour chaque élément $e \in S$, σe est vrai[4]. Une substitution fermée θ est solution fermée d'un système de contraintes S

3. Il est aussi possible de définir les termes rationnels comme les termes qui sont solutions d'équations de la forme $x = t$ ou t est un terme fini qui n'est pas une variable (par exemple, f^ω est un terme rationnel puisqu'il est solution de l'équation $x = f(x)$).

4. σS est vrai n'implique pas forcément que pour toute substitution θ, $\theta\sigma S$ est vrai (par exemple, pour $S = \{x \neq f(a)\}$, $\sigma = \{x/f(y)\}$ et $\theta = \{y/a\}$, σS est vrai tandis que $\theta\sigma S$ n'est pas vrai)

si θS est vrai, et dans ce cas le système S est dit solvable. Une substitution θ est solution de S si pour toute instance fermée σ de θ (i.e. toute substitution fermée $\sigma = \mu\theta$), σS est vrai. Une interprétation de Herbrand I est un modèle d'un programme P si pour chaque clause $p(\vec{x}) \leftarrow S \mid B_1, \cdots, B_n$ de P, et pour chaque solution fermée θ de S, $\theta p(\vec{x}) \in I$ lorsque $\theta B_i \in I$ pour tout $1 \leqslant i \leqslant n$. On note :

$$[\![S \mid A]\!] = \{\theta A \in At^C_{\Sigma,\Pi}[\varnothing],\ \theta \text{ est une solution fermée de } S\}$$

Sémantique opérationnelle

Une dérivation à partir de la requête R_0 avec le programme P est une suite finie ou infinie de buts $R_0, R_1, \cdots, R_i, \cdots$ telle que pour tout $i \geq 0$, si R_i est de la forme $S \mid A_1, \cdots, A_n$ alors S est solvable et R_{i+1} est de la forme $S' \mid \vec{B_1}, \cdots, \vec{B_n}$ où les $\vec{B_i}$ sont les corps des clauses de P de la forme $A'_i \leftarrow S_i \mid \vec{B_i}$ et où le système S' est :

$$S' = S \cup S_1 \cup \cdots \cup S_n \cup \{A_1 = A'_1, \cdots, A_n = A'_n\}$$

où $A_i = A'_i$ désigne, lorsque $A_i = p(\vec{t_1})$ et $A'_i = p(\vec{t_2})$, le système d'équations $\vec{t_1} = \vec{t_2}$. Une dérivation réussit (resp. échoue) lorsqu'elle est finie et que le dernier but s'écrit $S \mid \varnothing$ (resp. $S \mid \vec{A}$ et que l'on ne peut plus produire avec le programme un but à partir de $S \mid \vec{A}$). Les autres dérivations sont infinies. Tandis qu'en programmation logique, une dérivation produit une substitution, en programmation logique avec contraintes, une dérivation produit un système de contraintes : dans le cas d'une dérivation finie, ce système est celui de l'état final de la dérivation, dans le cas d'une dérivation infinie, ce système correspond à l'union des systèmes y apparaissant ($\cup_{i \geq 0} S_i$). Les systèmes de contraintes éventuellement infinis satisfont le lemme suivant, établi par A. Colmerauer :

Lemme 4.4 *Un système de contraintes S, éventuellement infini, contenant un ensemble E d'équations et un ensemble fini $\{\bar{e}_1, \cdots, \bar{e}_n\}$ d'inéquations ($n \geq 1$) est solvable si et seulement si chaque sous-système $E \cup \{\bar{e}_i\}$ est solvable ($1 \leqslant i \leqslant n$).*

On définit les ensembles de succès et d'échecs finis d'un programme P par :

$$\begin{aligned} S_P = \{(S \mid A), &\quad \exists\ S \mid A \xrightarrow{*} S' \mid \varnothing \text{ telle que toute solution de } S' \\ &\quad \text{est solution de } S\} \\ E_P = \{(S \mid A), &\quad \text{toute dérivation à partir de } S \mid A \text{ est une} \\ &\quad \text{dérivation d'échec}\} \end{aligned}$$

L'opérateur T_P associé à un programme P est défini sur $2^{At^C_{\Sigma,\Pi}[\varnothing]}$ par :

$$T_P(I) = \{A, \quad \begin{array}{l} \exists B \leftarrow S \mid B_1, \cdots, B_n \\ \exists \theta \text{ substitution fermée} \\ A = \theta B \\ \theta \text{ est une solution fermée de } S \\ \forall 1 \leqslant i \leqslant n \quad \theta B_i \in I \end{array} \quad \}$$

Cet opérateur est continu et vérifie :

$$T_P^{\uparrow\omega} = \mathsf{lfp}(T_P) \subseteq T_P^{\downarrow\alpha} = \mathsf{gfp}(T_P) \subseteq T_P^{\downarrow\omega} \text{ pour un ordinal } \alpha \geq \omega$$

Considérons à présent le complément Ω_P de $S_P \cup E_P$. Ω_P contient les éléments à partir desquels on ne peut pas construire de dérivation finie de succès et qui sont à l'origine d'au moins une dérivation infinie. $[\![\Omega_P]\!]$ peut être partitionné en deux ensembles. Soit α l'ordinal de fermeture de T_P (i.e. le plus petit ordinal α tel que $\mathsf{gfp}(T_P) = T_P^{\downarrow\alpha}$), on définit $[\![\Omega_P^S]\!]$ comme l'intersection de $[\![\Omega_P]\!]$ avec $T_P^{\downarrow\alpha}$ et $[\![\Omega_P^E]\!]$ comme le complément de $[\![\Omega_P^S]\!]$ dans $[\![\Omega_P]\!]$:

$$\mathsf{gfp}(T_P) = T_P^{\downarrow\alpha} \qquad [\![\Omega_P^S]\!] = [\![\Omega_P]\!] \cap T_P^{\downarrow\alpha} \qquad [\![\Omega_P^E]\!] = [\![\Omega_P]\!] \backslash [\![\Omega_P^S]\!]$$

Ω_P^S (resp. Ω_P^E) représente l'ensemble des succès infinis (resp. des échecs infinis) de P. Le résultat final, issu de [47, 50], s'énonce alors :

Théorème 4.6 ([47, 50]) *Soit P un programme logique avec contraintes.*

1. *Si α est l'ordinal de fermeture de T_P, alors :*
   (a) *Succès finis :* $\mathsf{lfp}(T_P) = T_P^{\uparrow\omega} = [\![S_P]\!]$
   (b) *Succès infinis :* $T_P^{\downarrow\alpha} \backslash T_P^{\uparrow\omega} = \mathsf{gfp}(T_P) \backslash \mathsf{lfp}(T_P) = [\![\Omega_P^S]\!]$
   (c) *Echecs infinis :* $T_P^{\downarrow\omega} \backslash \mathsf{gfp}(T_P) = [\![\Omega_P^E]\!]$
   (d) *Echecs finis :* $At^C_{\Sigma,\Pi}[\varnothing] \backslash T_P^{\downarrow\omega} = [\![E_P]\!]$
   (e) $G \in \Omega_P^E \cup E_P \Leftrightarrow [\![G]\!] \subseteq At^C_{\Sigma,\Pi}[\varnothing] \backslash T_P^{\downarrow\alpha}$
2. (a) *Le système $S_\omega = \cup_{i \geq 0} S_i$ associé à une dérivation infinie issue d'un élément de Ω_P^S est solvable.*
   (b) *Le système $S_\omega = \cup_{i \geq 0} S_i$ associé à une dérivation infinie issue d'un élément de Ω_P^E n'est pas solvable.*

L'approche par contraintes semble plus riche que les approches développées pour la programmation logique «pure» car elle distingue deux types de dérivations infinies en caractérisant, d'une part, les succès infinis et, d'autre part, les échecs infinis. Cependant cette distinction n'est pertinente que lorsque l'on considère des programmes logiques avec des contraintes comportant des inégalités. Ces programmes n'admettent pas de programmes définis (classiques) «équivalents». C'est par exemple le cas du programme :

$$P = \{p(x) \leftarrow \{x = a\} \mid q(y) \ ; \ q(x) \leftarrow \{x = f(y), x \neq f(x)\} \mid q(y)\} \quad (4.13)$$

Pour ces programmes, T_P ne satisfait pas de propriété de fermeture : par exemple, si l'on considère l'espace métrique de la définition 4.3, $At^C_{\Sigma,\Pi}[\varnothing]$ contient toutes les limites des suites de Cauchy de $At_{\Sigma,\Pi}[\varnothing]$ ce qui n'est pas le cas de $T_P(At^C_{\Sigma,\Pi}[\varnothing])$ (où P est le programme (4.13)) qui ne contient pas l'élément limite $q(f^\omega)$. La propriété $\mathsf{gfp}(T_P) = T_P^{\downarrow\omega}$ n'est donc plus vérifiée, même en présence d'éléments infinis dans l'univers du discours. Cette situation est due à la possibilité d'introduire des inégalités dans les clauses du programme. Si, au contraire, on se limite aux programmes logiques avec contraintes que l'on peut obtenir à partir d'un programme défini [5], alors l'ordinal de fermeture de l'opérateur T_P défini sur $At^C_{\Sigma,\Pi}[\varnothing]$ est $\alpha = \omega$ et donc $T_P^{\downarrow\alpha} = T_P^{\downarrow\omega} = \mathsf{gfp}(T_P)$. L'ensemble des échecs infinis de ces programmes est alors vide et on retrouve les résultats obtenus dans l'approche métrique pour les succès infinis. Autrement dit, le système de contraintes S_ω résultant d'une dérivation infinie à partir d'un programme logique avec contraintes issu d'un programme défini est toujours solvable. L'approche par contraintes ne semble donc pas apporter de résultats nouveaux concernant les dérivations infinies que l'on peut obtenir à partir d'un programme défini mais confirme les résultats établis par l'approche métrique.

La programmation concurrente par contraintes [90] est une des extensions proposées pour la programmation logique avec contraintes. Dans ce paradigme de programmation, les calculs sont effectués par des agents qui communiquent en manipulant des contraintes dans une base de contraintes commune et qui se synchronisent en vérifiant la validité des contraintes présentes dans cette base. Les agents correspondent à des atomes (logiques) et les contraintes s'expriment dans des théories décidables quelconques. L'étude des dérivations infinies dans ce modèle de programmation est détaillée dans [25].

4.1.5 Approche par plus petit point fixe

L'utilisation de Cpo permet de définir une sémantique élégante des dérivations infinies par plus petit point fixe. Certaines dérivations infinies en programmation logique peuvent être vues comme une séquence infinie d'approximations de plus en plus précises d'un résultat. Plutôt que d'établir la convergence d'un tel processus d'approximations à l'aide d'une notion de distance, l'approche par Cpo consiste à caractériser une limite par une

5. par exemple le programme (4.9) s'écrit sous la forme d'un programme logique avec contraintes satisfaisant les conditions requises : $P = \{p(x) \leftarrow \{x = f(y)\} \mid p(y)\}$

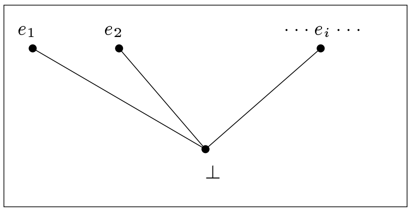


FIG. 4.2 – *Ordre plat*

borne supérieure. Faisant suite à une première tentative de formalisation de la sémantique des dérivations infinies, réalisée en 1985 par M. Falaschi, G. Levi et C. Palamidessi [78], G. Levi et C. Palamidessi [66] proposent, en 1988, une sémantique des dérivations infinies basée sur une version modifiée des programmes dans lesquels des clauses terminales (i.e. dont le corps est vide) contenant un symbole nouveau (correspondant à l'«indéfini») sont ajoutées. Cette méthode définit une sémantique caractérisant les bornes supérieures des ensembles dirigés d'atomes inclus dans le plus petit point fixe d'un opérateur : la limite d'un calcul est la borne supérieure d'une séquence d'approximations.

Complétion de la base de Herbrand

Afin de construire une structure (la base de Herbrand complétée) sur laquelle un opérateur $\uparrow$-continu pourra être défini, l'ensemble des termes est muni d'une structure de CPO. La définition des termes utilisée est celle de la définition 4.2. La structure de CPO de l'ensemble des termes est obtenue en ajoutant à la signature un nouveau symbole de fonction d'arité nulle, noté $\perp$ et correspondant à un objet complètement indéfini. Cette technique est classique et repose sur la proposition suivante.

Proposition 4.3 (Ordre plat (*flat* CPO)) *L'ensemble* $(E_\perp, \preceq)$ *défini par* $E_\perp = E \cup \{\perp\}$ *(où* $\perp \notin E$*) et* $x \preceq y \Leftrightarrow (x = \perp \vee x = y)$ *est un* CPO.

On peut donc munir la nouvelle signature fonctionnelle $\Sigma_\perp$ d'un ordre plat que l'on peut étendre à l'ensemble des termes fermés formés sur $\Sigma_\perp$, noté $T^\infty_{\Sigma_\perp}$, comme suit :

$$t_1 \leqslant t_2 \Leftrightarrow \forall u \in \mathcal{O}(t_1)\,(u \in \mathcal{O}(t_2) \wedge t_1(u) \leqslant t_2(u))$$

Par exemple, on a $f(a, \perp, a) \leqslant f(a, g(a, a), a)$. $(T^\infty_{\Sigma_\perp}, \leqslant)$ forme un CPO (algébrique) et tout sous-ensemble Δ de $T^\infty_{\Sigma_\perp}$ qui admet un majorant admet une borne supérieure. De plus, si Δ est fini et ne contient que des termes finis alors cette borne supérieure est un terme fini. De manière similaire, la relation $\leqslant$ est définie pour les substitutions par :

$$\theta_1 \leqslant \theta_2 \Leftrightarrow \forall x \quad \theta_1(x) \leqslant \theta_2(x)$$

Tout ensemble de substitutions Θ dans $T^{\infty}_{\Sigma_\perp}$ admettant un majorant admet une borne supérieure. De plus, si Θ est fini et ne contient que des substitutions dans les termes finis (i.e. des substitutions θ telles que, pour toute variable x, $\theta(x)$ est fini), cette borne supérieure est une substitution dans les termes finis. $At_{\Sigma_\perp,\Pi}[\varnothing]$ est défini comme l'ensemble des atomes que l'on peut construire à partir des termes finis de $T^{\infty}_{\Sigma_\perp}$, et $At^{C}_{\Sigma_\perp,\Pi}[\varnothing]$ est défini comme l'ensemble des atomes que l'on peut construire à partir de tous les termes de $T^{\infty}_{\Sigma_\perp}$. La relation $\leqslant$ est étendue aux atomes de ces deux ensembles par :

$$p(t_1,\cdots,t_n) \leqslant p(t'_1,\cdots,t'_n) \Leftrightarrow \forall 1 \leqslant i \leqslant n \quad t_i \leqslant t'_i$$

Une interprétation de Herbrand est un sous-ensemble de $At^{C}_{\Sigma_\perp,\Pi}[\varnothing]$. Lorsque les éléments d'une interprétation sont tous dans $At_{\Sigma_\perp,\Pi}[\varnothing]$, on parle d'une interprétation de Herbrand partielle. L'ensemble des interprétations de Herbrand partielles, ordonné par l'inclusion ensembliste $\subseteq$, forme un treillis complet. Toutefois, $At^{C}_{\Sigma_\perp,\Pi}[\varnothing]$ (et $At_{\Sigma_\perp,\Pi}[\varnothing]$) n'admet pas de plus petit élément relativement à $\leqslant$ puisque chaque atome $p(\perp,\cdots,\perp)$ est minimal. On partitionne donc $At^{C}_{\Sigma_\perp,\Pi}[\varnothing]$ en une famille d'ensembles $At^{C}_{\Sigma_\perp,p}[\varnothing]$ pour chaque prédicat p. $(At^{C}_{\Sigma_\perp,p}[\varnothing],\leqslant)_{p\in\Pi}$ forme alors une famille de Cpo. De plus, tout sous-ensemble Δ de $At_{\Sigma_\perp,\Pi}[\varnothing]$ admettant un majorant admet une borne supérieure, et si Δ est fini, on a $\mathsf{lub}(\Delta) \in At_{\Sigma_\perp,\Pi}[\varnothing]$. Chaque élément infini correspond donc à la borne supérieure d'un sous-ensemble dirigé d'éléments finis correspondant à ses approximations finies (c'est la propriété d'algébricité du Cpo).

Sémantique des dérivations infinies

La sémantique proposée dans [66] repose sur la notion de plus petit point fixe d'une variante de l'opérateur T_P. Ce plus petit point fixe est atteint en construisant une suite croissante d'interprétations à partir de $\varnothing$ en ω étapes. Par exemple, étant donné le programme :

$$P = \{\mathsf{LA}([a|l]) \leftarrow \mathsf{LA}(l)\ ;\ \mathsf{LA}([\,]) \leftarrow\}$$

l'opérateur classique T_P, permet de construire la suite d'interprétations :

$$T_P^{\uparrow 0} = \varnothing\ ;\ T_P^{\uparrow 1} = \{\mathsf{LA}([\,])\}\ ;\ T_P^{\uparrow 2} = \{\mathsf{LA}([\,]),\mathsf{LA}([a|[\,]])\}\ ;\cdots$$

Or, cette construction n'est possible que parce qu'il existe une clause unité (aussi appelée clause terminale) pour le prédicat LA. Par contre, à partir du programme :

$$P = \{\mathsf{LA}([a|l]) \leftarrow \mathsf{LA}(l)\} \tag{4.14}$$

il vient $T_P^{\uparrow 0} = T_P^{\uparrow 1} = \cdots = T_P^{\uparrow i} = \cdots = \varnothing$. La solution proposée pour pallier ce problème lorsque le programme ne contient pas au moins une clause unité par prédicat apparaissant dans une tête de clause (et c'est naturellement le cas pour beaucoup de programmes qui permettent de construire des dérivations infinies) consiste à ajouter à P des clauses terminales qui serviront de point de départ pour la construction d'une séquence non vide d'interprétations. Ces clauses sont les clauses unités obtenues en substituant $\perp$ à toutes les variables apparaissant dans les têtes de clauses de P. Par exemple, avec le programme (4.14), l'ensemble des clauses terminales à ajouter au programme est $C_T(P) = \{\mathsf{LA}([a|\perp]) \leftarrow\}$. Cette clause signifie que LA peut «terminer» avec une liste en argument dont le premier élément est a et dont la queue est indéfinie. L'opérateur T_P est défini sur le treillis complet des interprétations de Herbrand partielles par :

$$\begin{array}{ll} T_P(I) = \{A, & \exists B \leftarrow B_1, \cdots B_n \in P \cup C_T(P) \\ & \exists \theta_1, \cdots, \theta_n \text{ dans les termes finis} \\ & \exists \theta = \mathsf{lub}(\theta_1, \cdots, \theta_n) \\ & \theta B = A \\ & \forall 1 \leqslant i \leqslant n \quad \theta_i B_i \in I \\ & \forall x\ (x \in B \wedge x \notin B_i) \Rightarrow \perp \notin \theta(x) \quad \} \end{array}$$

Cet opérateur est monotone et $\uparrow$-continu et admet donc $T_P^{\uparrow\omega}$ pour plus petit point fixe. La sémantique par point fixe d'un programme P est définie par :

$$D_{\Sigma_\perp}(P) = \{A \in At^C_{\Sigma_\perp,\Pi}[\varnothing],\ \exists L \subseteq \mathsf{lfp}(T_P),\ L \text{ est dirigé},\ A = \mathsf{lub}(L)\}$$

Une version restreinte de cet ensemble, nécessaire pour mettre en relation $D_{\Sigma_\perp}(P)$ avec la sémantique opérationnelle, s'obtient en imposant à A de ne pas contenir le symbole $\perp$. L'ensemble ainsi défini est noté $D_\Sigma(P)$. Par exemple, avec le programme (4.14), on a :

$$\mathsf{lfp}(T_P) = \{\mathsf{LA}([a|\perp]), \mathsf{LA}([a,a|\perp]), \cdots\} \quad D_\Sigma(P) = \{\mathsf{LA}([a,a,\cdots])\}$$

L'ensemble $D_\Sigma(P)$ est relié à une notion d'atomes calculables à l'infini à partir de P similaire à celle donnée dans la définition 4.4. Pour cela, l'ensemble des atomes éventuellement infinis $At^C_{\Sigma,\Pi}[X]$ est muni d'une structure de famille de Cpo.

Définition 4.6 (Atomes calculables à l'infini [66]) *Soit P un programme et A un atome de $At^C_{\Sigma,\Pi}[\varnothing] \backslash At_{\Sigma,\Pi}[\varnothing]$ (i.e. un atome infini sans variables ne contenant pas le symbole $\perp$). A est calculable à l'infini ($A \in C_P$) si il existe un atome fini B (pouvant contenir des variables mais ne contenant pas le symbole $\perp$) et une dérivation infinie équitable, dont la séquence d'unificateurs est $\theta_1, \cdots, \theta_i, \cdots$, à partir de B telle que :*

$$A = \mathsf{lub}_{n\in\omega}(\theta_n \cdots \theta_1 B)$$

$\{\theta_n \cdots \theta_1 B\}_{n\in\omega}$ admet bien une borne supérieure puisqu'il s'agit d'une chaîne et que $At^C_{\Sigma,\Pi}[X]$ forme une famille de Cpo (algébriques). Le résultat final de [66] s'énonce alors :

Théorème 4.7 (Validité [66]) $C_P \subseteq D_\Sigma \backslash At_{\Sigma,\Pi}[\varnothing]$.

Pour finir, signalons que l'article de G. Levi et C. Palamidessi [66] établit une comparaison entre $D_\Sigma(P)$ et différents ensembles. Si D_1 (resp. D_2) est le plus petit (resp. le plus grand) point fixe de l'opérateur T_P classique défini sur $At_{\Sigma,\Pi}[\varnothing]$ et si D_3 est le plus grand point fixe de l'opérateur T_P classique sur $At^\infty_{\Sigma,\Pi}[\varnothing]$, alors on a le lemme suivant.

Lemme 4.5 ([66]) *Soit P un programme défini, on a $D_1 \subseteq D_2 \subseteq D_3$ et $D_1 \subseteq D_\Sigma(P)$.*

Comme les autres approches, la sémantique développée dans [66] présente l'inconvénient de ne pas être complète puisque $C_P \neq D_\Sigma \backslash At_{\Sigma,\Pi}[\varnothing]$. G. Levi et C. Palamidessi expliquent cette incomplétude en suggérant que l'ensemble des clauses terminales $C_T(P)$ ajoutées au programme P est trop grand. Considérons, par exemple, le programme :

$$P = \{p(l) \leftarrow q(l,l'), r(l') \ ; \ q([a|l], l') \leftarrow q(l,l') \ ; \ r(l) \leftarrow s\} \tag{4.15}$$

L'ensemble des clauses terminales à ajouter est :

$$C_T(P) = \{p(\bot) \leftarrow \ ; \ q([a|\bot], \bot) \leftarrow \ ; \ r(\bot) \leftarrow\}$$

Il vient alors :

$$\begin{array}{l} \mathsf{lfp}(T_P) = \{q([a|\bot],\bot), q([a,a|\bot],\bot), \cdots, r(\bot), p([a|\bot]), p([a,a|\bot]), \cdots\} \\ p([a,a,\cdots]) \in D_\Sigma(P) \end{array}$$

Cependant, $p([a,a,\cdots]) \notin C_P = \varnothing$ puisque toute dérivation à partir de la requête $p(l)$ n'est infinie que si elle n'est pas équitable (dès que l'atome sélectionné est r la dérivation est une dérivation finie d'échec). Deux solutions potentielles sont suggérées dans [66] : la première consisterait en une restriction de l'ajout de clauses terminales aux seules clauses du programme donnant lieu à au moins un appel récursif (par exemple, l'ensemble $C_T(P)$ du programme (4.15) serait $\{q([a|\bot], \bot) \leftarrow\}$), tandis que la deuxième consisterait en une restriction sur les programmes.

4.2 Induction, co-induction et programmation logique

C'est la «déclarativité» du paradigme de la programmation logique qui est le plus souvent invoquée pour défendre l'idée que ce modèle fournit un cadre avantageux de programmation. En effet, l'écriture d'un programme défini consiste seulement à spécifier un problème à l'aide de formules logiques du premier ordre, les clauses de Horn, ce qui correspond au célèbre slogan «*logic programs as first-order theories*». La sémantique déclarative d'un programme (i.e. l'interprétation d'un programme par un ensemble de formules logiques) coïncide exactement avec sa sémantique opérationnelle, définie en terme d'ensemble des succès fermés finis. Considérons, par exemple, la définition du prédicat even caractérisant les entiers pairs :

$$P = \{\mathsf{even}(0) \leftarrow\ ;\ \mathsf{even}(S(S(x))) \leftarrow \mathsf{even}(x)\} \tag{4.16}$$

La lecture logique de ce programme est obtenue en associant à P la formule logique $\mathrm{F}(P)$ correspondant à la conjonction des clauses de Horn du programme :

$$\mathrm{F}(P) = \mathsf{even}(0) \wedge \forall x(\mathsf{even}(S(S(x))) \vee \neg\mathsf{even}(x))$$

Les atomes $\mathsf{even}(S^{2i}(0))$, pour $i \geq 0$, sont des conséquences sémantiques (logiques) de $\mathrm{F}(P)$, puisque pour tout $i \geq 0$, $\mathrm{F}(P) \models \mathsf{even}(S^{2i}(0))$. even correspond donc bien au prédicat caractérisant les entiers pairs. D'un point de vue sémantique, l'intérêt de la restriction aux clauses de Horn provient de la propriété de fermeture pour l'intersection des modèles, énoncée par le lemme 3.14. La formule logique $\mathrm{F}(P)$ associée à un programme P admet donc un plus modèle de Herbrand. Par exemple, pour le programme (4.16), ce plus petit modèle est :

$$\mathcal{M}_P = \bigcup_{i \geq 0} \{\mathsf{even}(S^{2i}(0))\}$$

D'un point de vue opérationnel, cet ensemble coïncide avec l'ensemble des succès de P puisque l'on peut construire les dérivations :

$$\begin{array}{rrr}
 & & \mathsf{even}(0) \rightarrow \square \\
 & \mathsf{even}(S^2(0)) \rightarrow & \mathsf{even}(0) \rightarrow \square \\
 & \vdots & \\
\mathsf{even}(S^{2i}(0)) \rightarrow \cdots \rightarrow & \mathsf{even}(S^2(0)) \rightarrow & \mathsf{even}(0) \rightarrow \square \\
 & \vdots &
\end{array}$$

Plus généralement, la substitution $\theta = \{x/S^{2i}(0)\}$, pour $i \geq 0$, est une solution pour la requête $\mathsf{even}(x)$ et l'arbre SLD de racine $\mathsf{even}(x)$ contient une branche infinie qui permet l'énumération des atomes $\mathsf{even}(S^{2i}(0))$.

$$\begin{array}{ccccccc}
\mathsf{even}(x) & \overset{\left[\begin{array}{c} x \\ S(S(x_1)) \end{array}\right]}{\rightarrow} & \mathsf{even}(x_1) & \rightarrow \cdots \rightarrow & \mathsf{even}(x_i) & \rightarrow \cdots \\
\downarrow & & \downarrow & & \downarrow & \\
\square & & \square & & \square & \\
x = 0 & & x = S^2(0) & & x = S^{2i}(0) &
\end{array}$$

S'il est séduisant, le paradigme «*logic programs as first-order theories*» n'est pas, d'un point de vue pratique, tout à fait satisfaisant. Par exemple, il ne permet pas de traiter clairement la négation : une requête constituée d'un atome fermé A peut échouer à partir d'un programme P même si $\neg A$ n'est pas une conséquence sémantique (logique) de $\mathrm{F}(P)$. La négation par échec que l'on trouve dans la plupart des interprètes Prolog ne correspond donc pas à la négation logique.

Une alternative à ce paradigme consiste à considérer les clauses d'un programme par des règles : à toute clause définie $A \leftarrow A_1, \cdots, A_n$ peut être associée une fonction c d'arité n que l'on peut interpréter comme une règle de construction de preuves :

$$c : \pi_{A_1} \times \cdots \times \pi_{A_n} \to \pi_A$$

L'application de c à des preuves π_{A_i} des A_i fournit une preuve de A et on note $\pi_A : A$ pour indiquer que π_A est une preuve de A (ce qui peut aussi se lire π_A est de type A). Il est donc possible de voir une clause comme la signature fonctionnelle d'un constructeur de preuves. Cette correspondance entre types et propositions d'une part, et preuves et fonctions d'autre part, est bien connue : il s'agit de l'isomorphisme de Curry-Howard. Si l'on reconsidère le programme (4.16) en utilisant cette approche, les deux clauses définissant le prédicat even peuvent être vues comme un schéma de règles :

$$(\mathsf{even\text{-}0}) : \frac{}{\mathsf{even}(0)} \qquad (\mathsf{even\text{-}S2}) : \frac{\mathsf{even}(x)}{\mathsf{even}(S^2(x))}$$

dont les instances fermées correspondent à l'ensemble infini de règles défini en (1.7). Cette lecture sémantique de la programmation logique permet de considérer un programme, non plus comme une théorie du premier ordre, mais comme la définition d'une «nouvelle logique» (i.e. un système formel). En adoptant ce point de vue, la dénotation d'un programme correspond à l'ensemble des théorèmes qu'il est possible de «dériver» dans cette logique. Cette approche nous conduit donc à identifier un programme à une définition inductive ou co-inductive (selon que l'on s'autorise ou non l'application des règles un nombre infini de fois).

4.2.1 Induction et programmation logique

Définitions inductives, exprimées à partir d'un ensemble de règles, et programmes définis présentent de nombreuses similitudes qui permettent d'envisager le paradigme «*logic programs as inductive definitions*». Un programme défini P peut être vu comme un ensemble de règles $[P]$, éventuellement infini, correspondant aux instances fermées des clauses de P. D'autre part, pour chacune de ces règles, l'ensemble des prémisses est fini et donc $[P]$ est finitaire. La propriété d'intersection des modèles de Herbrand pour un ensemble de clauses de Horn, exprimée par le lemme 3.14, correspond exactement à la propriété de fermeture pour l'intersection d'ensembles $[P]$-clos, exprimée par le lemme 1.1 (rappelons que d'après le lemme 3.16, une interprétation de Herbrand I est un modèle de P si et seulement si $T_P(I) \subseteq I$, c'est à dire si I est T_P-clos). On a donc $\mathcal{M}_P = \mathsf{Ind}([P])$. Cet ensemble est généralement caractérisé en terme de plus petit point fixe de l'opérateur classique T_P associé à P. Cet opérateur n'est rien d'autre que $T_{[P]}$ (défini à partir de $[P]$ en (1.1)), et l'égalité (1.2) permet d'établir l'égalité $\mathsf{Ind}([P]) = \mathsf{Ind}(T_{[P]})$. On peut alors utiliser le théorème de Tarski (théorème 1.1) pour montrer $\mathcal{M}_P = \mathsf{lfp}(T_{[P]})$. Lorsque ce point fixe est caractérisé en terme de puissances ordinales de l'opérateur $T_{[P]}$, c'est le lemme 1.6 qui permet de montrer que $T_{[P]}$ est un opérateur finitaire, ce qui permet, à l'aide du lemme 1.5, de montrer que $T_{[P]}$ est $\uparrow$-continu. Ces deux lemmes permettent alors d'utiliser le théorème 1.2 pour montrer l'égalité $\mathcal{M}_P = \mathsf{Ind}(T_{[P]}) = \mathsf{lfp}(T_{[P]}) = T_{[P]}^{\uparrow\omega}$. Aussi, plutôt qu'un simple outil technique utilisé pour montrer le théorème de complétude de la SLD-résolution, les définitions inductives constituent une alternative «naturelle» au paradigme «*logic programs as first-order theories*» : un programme défini peut être interprété par une définition inductive. En adoptant ce point de vue, on a donc les identifications :

$$\mathsf{even}(S^2(0)) \overset{\mathsf{even\text{-}S2}}{\rightarrow} \mathsf{even}(0) \overset{\mathsf{even\text{-}0}}{\rightarrow} \square \qquad (\mathsf{even\text{-}S2}) : \frac{(\mathsf{even\text{-}0}) : \overline{\mathsf{even}(0)}}{\mathsf{even}(S^2(0))}$$

SLD-dérivation à partir de $[P]$ Preuve dans $[P]$

$\mathsf{even\text{-}S2}(\mathsf{even\text{-}0})$ est donc un terme de preuve de type (i.e. correspondant à une preuve de) $\mathsf{even}(S^2(0))$. D'ailleurs, la facilité que fournit le modèle de programmation logique pour le traitement des langages provient plus du fait que ces langages ont une syntaxe définie inductivement que de l'existence d'une interprétation logique de ces langages. Cette interprétation de la programmation logique, en terme de définitions inductives, est aujourd'hui largement utilisée. Dans un livre consacré à la calculabilité dans le contexte de la programmation logique [32], M. Fitting présente quelques exemples de preuves de correction de programmes en utilisant cette approche. En ce qui concerne la négation, le concept de programme stratifié semble mieux s'interpréter en terme de définitions inductives qu'en terme de théorie du premier ordre. En effet, lorsqu'un programme peut être partitionné en plusieurs définitions inductives de manière à ce que chaque négation fasse uniquement référence à une ensemble déjà considéré (i.e. appartenant à une strate inférieure à celle du prédicat correspondant à la clause dans laquelle la négation apparaît), c'est à dire lorsque le graphe de dépendance du programme ne contient pas de cycles, le programme est dit stratifiable et peut être interprété en terme de définitions inductives itératives. Cette vision de la négation, détaillée dans [92], utilise bien plus les concepts de définitions inductives, de points fixes ou encore de nombres ordinaux que les notions de conséquences sémantiques de la logique du premier ordre. Dans le même esprit, M. Hagiya et T. Sakurai proposent dans [40] un système d'inférence pour fonder la sémantique des programmes définis et étendent le formalisme des clauses de Horn pour permettre l'introduction de formules logiques quelconques dans le corps des clauses. Signalons aussi les travaux de L. Hallnäs et P. Schroeder-Heister qui s'inspirent des techniques de preuve en déduction naturelle pour étendre la puissance d'expression de la programmation logique. La sémantique des programmes qu'ils proposent dans [42, 43] repose, ici encore, sur l'identification des clauses d'un programme avec un ensemble de définitions inductives et permet de modéliser directement les substitutions solutions non nécessairement fermées, plutôt qu'un ensemble d'instances fermées de ces substitutions. Le langage qu'ils définissent, appelé Ghc (pour *Generalized Horn Clauses*), permet la présence d'implications dans le corps des clauses. Aussi, les définitions inductives considérées ne peuvent plus être étudiées dans le cadre présenté dans le chapitre 1 (il ne s'agit plus de définitions inductives monotones) : le paradigme utilisé est celui des définitions inductives partielles, détaillé dans [41]. Enfin, L.C. Paulson et A.W. Smith étudient dans [82] la possibilité de combiner la programmation logique avec la programmation fonctionnelle en adoptant le paradigme «*logic programs as inductive definitions*». L'approche qu'ils proposent, proche de celle développée dans [10], distingue deux types de symboles de fonctions : les constructeurs de termes sont des fonctions qui servent à la définition de l'univers de Herbrand tandis que certaines fonctions sont destinées à être évaluées. Ce point de vue fournit un cadre pour un langage logique avec fonctions dans lequel les clauses peuvent invoquer des fonctions qui permettent d'effectuer des opérations dans l'univers de Herbrand. Le mécanisme d'unification utilisé rend donc possible l'évaluation de certaines fonctions. Ce paradigme est proche de celui de la programmation logique avec égalité consistant à considérer les programmes comme une théorie du premier ordre comprenant l'égalité mais n'est pas sujet à ses restrictions (confluence, terminaison).

4.2.2 Co-induction et programmation logique

Dérivations infinies et dénotation par plus grand point fixe

Les définitions co-inductives permettent de définir des ensembles dont les éléments sont obtenus en appliquant un nombre de fois éventuellement infini les règles de construction. Ces définitions permettent donc de représenter aussi bien les termes infinis d'un univers de Herbrand complété que certaines SLD-dérivations infinies qui peuvent être vues comme une suite infinie d'applications des clauses d'un programme. Il paraît donc possible de considérer un programme défini comme une définition co-inductive afin de prendre en compte explicitement les dérivations infinies qui sont des preuves construites en appliquant un nombre de fois infini les constructeurs de preuves. En suivant cette approche, la dénotation d'un programme P correspond à l'ensemble $\mathsf{CoInd}(\Phi)$, où Φ est un ensemble de règles construites à partir de P. Comme nous l'avons vu, $\mathsf{CoInd}(\Phi)$ peut aussi être caractérisé par le plus grand point fixe d'un opérateur monotone, associé à P, noté T_Φ. Lorsque le domaine (resp. le co-domaine) de cet opérateur est étendu à l'ensemble des parties d'une base de Herbrand complétée et si la technique de complétion utilisée permet d'établir la $\downarrow$-continuité de T_Φ, ce plus grand point fixe est bien sûr $T_\Phi^{\downarrow\omega}$. Aussi, dans les approches développées pour donner un sens aux dérivations infinies en programmation logique qui utilisent cette propriété, c'est l'ensemble $\mathsf{CoInd}(\llbracket P \rrbracket)$, où $\llbracket P \rrbracket$ correspond aux instances fermées des clauses de P dans un domaine complété (i.e. pouvant contenir des atomes infinis), qui est proposé comme sémantique déclarative d'un programme P. Toutefois, dans ce cadre, l'interprétation «*logic programs as co-inductive definitions*» n'est pas complète. En effet, tandis que les atomes de $\mathsf{Ind}([P])$, les clauses de P et les requêtes sont définis sur un même langage (un langage du premier ordre ne contenant que des éléments finis), $\mathsf{CoInd}(\llbracket P \rrbracket)$ est défini sur un langage plus expressif que le langage des clauses de P et des requêtes, puisqu'il autorise la présence de termes infinis. Par exemple, si l'on considère le programme défini en (4.2), on a $p(f^\omega) \in \mathsf{gfp}(T_{\llbracket P \rrbracket})$, même si $p(f^\omega)$ n'est pas calculable à l'infini à partir de P, puisque $p(f^\omega) \leftarrow p(f^\omega) \in \llbracket P \rrbracket$. En autorisant la présence d'éléments infinis dans les requêtes et les programmes, l'approche métrique devient complète puisqu'alors on peut construire la dérivation :

$$p(f^\omega) \to_P p(f^\omega) \to_P \cdots \to_P p(f^\omega) \to_P \cdots$$

Les phénomènes d'incomplétude observés proviennent donc, en partie, du fait que l'opérateur associé à $\llbracket P \rrbracket$ est défini sur un domaine qui contient des éléments infinis sur lesquels on peut prouver directement des propriétés sans les construire (alors que $T_{[P]}^{\uparrow 0} = \varnothing$ et donc tous les éléments de $T_{[P]}^{\uparrow\omega} = \mathsf{Ind}([P])$ sont effectivement construits). Aussi, nous nous intéresserons dans la suite aux dérivations infinies qui ne construisent pas d'objets infinis et nous verrons qu'il existe une sémantique valide et complète pour cette classe de dérivations.

Dérivations infinies et termes de type co-inductif

La définition de $\mathsf{CoInd}(\Phi)$ par l'union de tous les ensembles Φ-denses correspond à une définition déclarative. L'identification d'un programme à une définition co-inductive peut être aussi envisagée en considérant les termes gardés par constructeurs, introduits dans le chapitre 1, que l'on peut définir à partir d'un programme vu comme un ensemble de règles. Par exemple, le programme (4.1) peut être identifié à la définition co-inductive (1.14). Toutefois, la dérivation du tableau 4.1 ne semble pas correspondre au terme de preuve (1.15) : tandis que la dérivation calcule par approximations successives la liste infinie des entiers à partir de k, le terme de preuve applique le schéma d'élimination de l'égalité (1.11) au terme $\mathsf{from}(k)$ qui représente la limite de la suite d'approximations construite par la dérivation. Par contre, si l'on se restreint à considérer des preuves, éventuellement infinies, de propriétés portant sur des objets finis, la correspondance entre termes de type co-inductif et SLD-dérivations infinies semble plus immédiate. Par exemple, le programme défini (4.2) est typique des programmes à l'origine de dérivations qui n'effectuent aucun calcul. C'est le cas de la dérivation (4.3) obtenue à partir de la requête $p(z)$. D'un point de vue «logique», cette dérivation correspond à une preuve par co-induction de $\forall z\, p(z)$, indépendamment du fait que z désigne un élément fini ou infini. En effet, si C désigne le constructeur associé à la clause $p(x) \leftarrow p(x)$, alors pour tout élément z de l'univers du discours $\mathcal{U}$, le terme de preuve défini par :

$$\pi_p := \lambda z : \mathcal{U}.\mathsf{C}(z, \pi_p(z))$$

est bien gardé par constructeurs (i.e. l'appel récursif à π_p est protégé par le constructeur C) et est de type $\Pi_{z:\mathcal{U}} p(z)$. Via l'isomorphisme de Curry-Howard, on peut donc voir ce terme comme une preuve de $\forall z\, p(z)$. L'application du constructeur correspond à la première transition de la dérivation (4.3) tandis que l'appel récursif correspond aux transitions suivantes. C'est bien sûr la présence de prédicats récursifs dans un programme défini qui rend possible la construction de dérivations infinies. Cette récursivité n'est pas toujours directe comme, par exemple, dans le programme :

$$P = \{\underbrace{p(x) \leftarrow q(x)}_{C_p}\ ;\ \underbrace{q(x) \leftarrow p(x)}_{C_q}\}$$

à partir duquel il est possible de construire la dérivation infinie :

$$p(z) \xrightarrow[P]{\theta_1 = \begin{bmatrix} x_1 \\ z \end{bmatrix}} q(z) \xrightarrow[P]{\theta_2 = \begin{bmatrix} x_2 \\ z \end{bmatrix}} \cdots \xrightarrow[P]{\theta_{2i} = \begin{bmatrix} x_{2i} \\ z \end{bmatrix}} p(z) \xrightarrow[P]{\theta_{2i+1} = \begin{bmatrix} x_{2i+1} \\ z \end{bmatrix}} q(z) \rightarrow_P \cdots \tag{4.17}$$

qui correspond, ici encore, à une preuve par co-induction de $\forall z\ p(z)$ représentée par le terme $\pi_p := \lambda x : \mathcal{U}.C_p(x, C_q(x, \pi_p(x)))$. D'autre part, on peut constater que la dérivation (4.17) «contient» aussi une preuve de $\forall z\ q(z)$:

$$q(z) \xrightarrow[P]{\theta_2 = \begin{bmatrix} x_2 \\ z \end{bmatrix}} \cdots \xrightarrow[P]{\theta_{2i} = \begin{bmatrix} x_{2i} \\ z \end{bmatrix}} p(z) \xrightarrow[P]{\theta_{2i+1} = \begin{bmatrix} x_{2i+1} \\ z \end{bmatrix}} q(z) \rightarrow_P \cdots$$

qui contient elle-même une preuve de $\forall z\ p(z)$ ... Les preuves de $\forall z\ p(z)$ et $\forall z\ q(z)$ peuvent donc s'exprimer de manière mutuellement récursive :

$$\pi_p := \lambda x : \mathcal{U}.C_p(x, \pi_q(x)) \qquad \pi_q := \lambda x : \mathcal{U}.C_q(x, \pi_p(x))$$

Toutefois certains prédicats non mutuellement récursifs peuvent donner lieu à de telles preuves. Une dérivation infinie peut donc être vue comme l'application, un nombre de fois infini, des clauses (i.e. constructeurs) et correspond donc à un terme en forme canonique. En identifiant un programme à une définition co-inductive, les clauses définissent des règles d'introduction auxquelle sont associées des schémas d'élimination. Par exemple, les deux clauses du programme :

$$P = \{\underbrace{p(x) \leftarrow p(f(x))}_{C_1}\ ;\ \underbrace{p(x) \leftarrow p(g(x))}_{C_2}\}$$

peuvent être vues comme des règles d'introduction :

$$(C_1) : \frac{\begin{array}{rcl} x & : & \mathcal{U} \\ \pi & : & p(f(x)) \end{array}}{C_1(x, \pi(x)) : p(x)} \qquad (C_2) : \frac{\begin{array}{rcl} x & : & \mathcal{U} \\ \pi & : & p(g(x)) \end{array}}{C_2(x, \pi(x)) : p(x)}$$

et peuvent être associées au schéma d'élimination suivant :

$$\textsf{(Elimination)} : \frac{\begin{array}{rcl} x & : & \mathcal{U} \\ \mathcal{P} & : & p(x) \to s \\ \pi & : & p(x) \\ \pi_1 & : & (x : \mathcal{U})(\pi_f : p(f(x)))\mathcal{P}(C_1(x, \pi_f)) \\ \pi_2 & : & (x : \mathcal{U})(\pi_g : p(g(x)))\mathcal{P}(C_2(x, \pi_g)) \end{array}}{\textsf{Case}\ \pi\ \textsf{of}\ \pi_1\ \pi_2\ \textsf{end} : \mathcal{P}(\pi)}$$

où s est quelconque, muni des deux règles de réduction :

$$\begin{array}{rcl} \textsf{Case}\ C_1(x, \pi')\ \textsf{of}\ \pi_1\ \pi_2\ \textsf{end} & \to & \pi_1(x, \pi') \\ \textsf{Case}\ C_2(x, \pi')\ \textsf{of}\ \pi_1\ \pi_2\ \textsf{end} & \to & \pi_2(x, \pi') \end{array}$$

On voit donc bien qu'une dérivation à partir de la requête $p(x)$ commence nécessairement en utilisant l'une des deux clauses C_1 ou C_2. Si l'on se restreint à la clause C_1, on obtient la dérivation suivante :

$$p(z) \to \left[\begin{array}{c} z \\ f(z) \end{array}\right] p(z) \to \left[\begin{array}{c} z \\ f(z) \end{array}\right] \left[\begin{array}{c} z \\ f(z) \end{array}\right] p(z) \to \cdots$$

qui peut être associée au terme de preuve :

$$\pi := \lambda z : \mathcal{U}.C_1 \left(z, \pi \left(\left[\begin{array}{c} z \\ f(z) \end{array}\right] p(z)\right)\right)$$

L'application du constructeur C_1 correspond bien à la première transition tandis que l'appel récursif correspond aux suivantes, c'est à dire à la dérivation à partir de $p(f(z))$. Un terme de preuve peut donc être vu comme la définition récursive de la suite des clauses appliquées dans la dérivation et *vice versa*. Néanmoins, signalons qu'un terme représentant une dérivation infinie n'admet pas systématiquement de représentation finie (i.e. ce terme n'est pas forcément définissable à l'aide d'une définition récursive gardée par constructeurs). En effet, à chaque étape n de toute dérivation commençant par $p(z)$, il est possible d'appliquer C_1 et C_2 et si la fonction :

$$\mathcal{F}_C : \mathbb{N} \to \{C_1, C_2\}$$

qui indique la clause utilisée à chaque étape n, n'est pas «calculable», il n'est pas possible de donner une représentation finie au terme correspondant à cette preuve. En effet, un tel terme s'écrit :

$$\pi{:=}\lambda x : \mathcal{U}.\pi_d(0,x) : \Pi_{x:\mathcal{U}}p(x)$$

avec :

$$\pi_d{:=}\lambda n : \mathbb{N}.\lambda x : \mathcal{U}.(\mathcal{F}_C(n))(x, \pi_d(S(n), (\mathcal{F}_F(n))(x)))$$

où $\mathcal{F}_F$ est la fonction définie par :

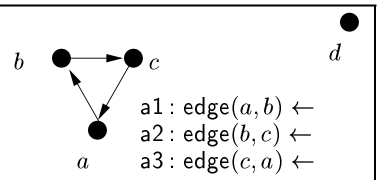
$$\mathcal{F}_F : \mathbb{N} \to \{f, g\}{:=}\lambda n : \mathbb{N}. \begin{pmatrix} \textsf{si } \mathcal{F}_C(n) = C_1 \\ \textsf{alors } f \\ \textsf{sinon } g \end{pmatrix}$$

et n'est «calculable» que si $\mathcal{F}_C$ l'est. Réciproquement, à partir d'une preuve π de $\forall x p(x)$, il est possible de définir la fonction $\mathcal{F}_C$ par $\mathcal{F}_P(\pi)$ où :

$$\mathcal{F}_P : \forall x p(x) \to \mathbb{N} \to \{C_1, C_2\}$$

$$\begin{array}{lll}
\mathcal{F}_P(\pi, n){:=} & \textsf{Case } \pi \textsf{ of} & \\
 & [x_1 : \mathcal{U}][\pi_1 : p(f(x_1))] & \textsf{Case } n \textsf{ of} \\
 & & C_1 \\
 & & [k_1 : \mathbb{N}]\mathcal{F}_P(\pi_1, k_1) \\
 & & \textsf{end} \\
 & [x_2 : \mathcal{U}][\pi_2 : p(g(x_2))] & \textsf{Case } n \textsf{ of} \\
 & & C_2 \\
 & & [k_2 : \mathbb{N}]\mathcal{F}_P(\pi_2, k_2) \\
 & & \textsf{end} \\
 & \textsf{end.} &
\end{array}$$

Tous les exemples présentés dans ce paragraphe montrent bien qu'il existe une correspondance directe entre une preuve infinie (i.e. un terme de type co-inductif) portant sur un objet fini et une dérivation infinie qui ne construit pas de termes infinis, alors que si l'objet considéré est infini, le terme de preuve décrit l'application, un nombre de fois infini, des constructeurs (i.e. des clauses) sur ce terme infini ce qui ne correspond pas à la dérivation infinie, si elle existe, qui représente l'application de clauses sur des termes finis.



a1 : $\mathsf{edge}(a,b) \leftarrow$
a2 : $\mathsf{edge}(b,c) \leftarrow$
a3 : $\mathsf{edge}(c,a) \leftarrow$

c1 : $\mathsf{path}(x,x) \leftarrow$
c2 : $\mathsf{path}(x,z) \leftarrow \mathsf{edge}(x,y), \mathsf{path}(y,z)$

FIG. 4.3 – *Chemin entre deux sommets d'un graphe orienté*

Un exemple

L'identification d'un programme défini à un ensemble de définitions co-inductives modifie la sémantique déclarative d'un programme à laquelle le paradigme «*logic programs as first-order theories*» nous a habitué. Considérons, par exemple, le programme de la figure 4.3 permettant de caractériser les chemins d'un graphe orienté. L'existence d'un cycle dans ce graphe rend possible la construction d'une dérivation infinie à partir d'une requête $\mathsf{path}(s_1, s_2)$ où s_1 est un sommet apparaissant sur un cycle et s_2 est un sommet quelconque. Par exemple, on a :

$$\begin{array}{ll} \mathsf{path}(a,d) & \rightarrow_P \mathsf{edge}(a,b), \mathsf{path}(b,d) \rightarrow_P \mathsf{path}(b,d) \\ & \rightarrow_P \mathsf{edge}(b,c), \mathsf{path}(c,d) \rightarrow_P \mathsf{path}(c,d) \\ & \rightarrow_P \mathsf{edge}(c,a), \mathsf{path}(a,d) \rightarrow_P \mathsf{path}(a,d) \rightarrow_P \cdots \end{array} \tag{4.18}$$

et plus généralement :

$$\begin{array}{ll} \mathsf{path}(a,x) & \rightarrow_P \mathsf{edge}(a,b), \mathsf{path}(b,x) \rightarrow_P \mathsf{path}(b,x) \\ & \rightarrow_P \mathsf{edge}(b,c), \mathsf{path}(c,x) \rightarrow_P \mathsf{path}(c,x) \\ & \rightarrow_P \mathsf{edge}(c,a), \mathsf{path}(a,x) \rightarrow_P \mathsf{path}(a,x) \rightarrow_P \cdots \end{array} \tag{4.19}$$

Il est possible de construire les termes de preuve π_{ad} et π_{ax} gardés par constructeurs correspondant aux dérivations (4.18) et (4.19) :

$$\begin{array}{lll} \pi_{ad} := & & \mathsf{c2}(a, b, d, \mathsf{a1}, \mathsf{c2}(b, c, d, \mathsf{a2}, \mathsf{c2}(c, a, d, \mathsf{a3}, \pi_{ad}))) \\ \pi_{ax} := & \lambda x : \mathcal{U}. & \mathsf{c2}(a, b, x, \mathsf{a1}, \mathsf{c2}(b, c, x, \mathsf{a2}, \mathsf{c2}(c, a, x, \mathsf{a3}, \pi_{ax}(x)))) \end{array}$$

Ces deux termes ont pour type $\mathsf{path}(a,d)$ et $\Pi_{x:\mathcal{U}}\mathsf{path}(a,x)$ et correspondent donc à une preuve de $\mathsf{path}(a,d)$ et $\forall x\ \mathsf{path}(a,x)$. En modifiant la définition du prédicat path en :

$$\mathsf{path}(x,x) \leftarrow\ ;\ \mathsf{path}(x,y) \leftarrow \mathsf{edge}(x,y)\ ;\ \mathsf{path}(x,z) \leftarrow \mathsf{path}(x,y), \mathsf{path}(y,z)$$

il devient même possible de prouver qu'il existe un chemin entre deux sommets quelconques d'un graphe orienté quelconque, puisque la dernière clause permet de construire des dérivations infinies sans faire intervenir le prédicat edge. Un phénomène «rassurant» peut cependant être observé si l'on rajoute un argument au prédicat path correspondant à la longueur du chemin qui sépare les deux sommets considérés :

$$\mathsf{path}(x, x, 0) \leftarrow \quad ; \quad \mathsf{path}(x, z, S(n)) \leftarrow \mathsf{edge}(x, y), \mathsf{path}(y, z, n)$$

Dans ce cas, la longueur du chemin séparant le sommet a du sommet d est S^ω, où S^ω est défini récursivement par le terme (protégé par constructeur) $S^\omega := S(S^\omega)$ et les sommets reliés par un chemin de longueur finie sont exactement ceux qui sont caractérisés par $\mathsf{Ind}([\mathsf{path}])$. Ici, la longueur du chemin considéré correspond à la longueur de la preuve de l'existence d'un chemin. Avec cette version du prédicat path, toute dérivation infinie construit un terme infini : elle effectue un calcul.

4.2.3 $\mathcal{C}$-sémantique

Les exemples présentés dans le paragraphe précédent illustrent bien le fait que s'il existe une correspondance «presque» directe entre les dérivations infinies qui «prouvent sur» un objet fini et les preuves par co-induction sur des objets finis, cette correspondance est moins immédiate dès que la preuve porte sur un objet infini. Typiquement, avec le programme défini en (4.9), la dérivation (4.10), qui semble établir $p(f^\omega)$ à partir de ce programme, ne correspond pas directement au terme de preuve de type $p(f^\omega)$: prouver par co-induction $p(f^\omega)$ revient à appliquer une infinité de fois le constructeur de preuve associé à la clause $p(f(x)) \leftarrow p(x)$ en utilisant à chaque «appel récursif» la propriété $f(f^\omega) = f^\omega$. D'autre part, certaines dérivations infinies qui ne construisent pas d'objets infinis correspondent à des preuves portant sur un objet dans lequel apparaît des variables. Aussi, la $\mathcal{C}$-sémantique fournit un cadre adéquat pour étudier cette classe de dérivations.

La $\mathcal{C}$-sémantique a été développée et étudiée par M. Falaschi, G. Levi, M. Martelli et C. Palamidessi dans [30, 31]. Il s'agit d'un sémantique des programmes définis basée sur une notion d'interprétation contenant des atomes non nécessairement fermés (contrairement aux interprétations de Herbrand). Cette approche, proposée pour la première fois par K.L. Clark dans [15], a été utilisée par W.G. Golson dans [38] pour étudier certaines dérivations infinies. Initialement, cette sémantique a été introduite pour permettre de prendre en compte l'aspect constructif des dérivations (i.e. la construction de substitutions) qui n'est pas considéré dans la théorie classique établissant la correspondance entre sémantique déclarative et sémantique opérationnelle uniquement pour les atomes fermés.

Sémantique déclarative

Le préordre $\leqslant$ sur les termes (défini par $t_1 \leqslant t_2 \Leftrightarrow \exists\theta\ \theta t_1 = t_2$) induit une relation d'équivalence, notée $\approx$, sur les termes (définie par $t_1 \approx t_2 \Leftrightarrow t_1 \leqslant t_2 \wedge t_2 \leqslant t_1$). Cette relation d'équivalence correspond à l'équivalence au renommage près des variables. L'univers de Herbrand considéré est l'ensemble quotient $T_\Sigma[X]_{/\approx}$. Afin d'alléger les notations, on confondra un terme avec sa classe d'équivalence. Cet ensemble peut être muni d'une relation d'ordre définie à partir de $\leqslant$, que l'on peut étendre à $(T_\Sigma[X]^n)_{/\approx}$[6]. La base de Herbrand considérée est $At_{\Sigma,\Pi}[X]_{/\approx}$ (i.e. l'ensemble des atomes $p(\vec{t})$ où p est un symbole de Π d'arité n et $\vec{t} \in (T_\Sigma[X]^n)_{/\approx}$). Cet ensemble est muni de la relation d'ordre $\leqslant$ définie par :

$$p(\vec{t_1}) \leqslant p(\vec{t_2}) \Leftrightarrow \vec{t_1} \leqslant \vec{t_2}$$

Considérer les sous-ensembles de $At_{\Sigma,\Pi}[X]_{/\approx}$ comme des interprétations n'est envisageable que si lorsqu'un atome appartient à une interprétation, ses instances (non nécessairement fermées) appartiennent aussi à cette interprétation. Aussi, une interprétation est définie comme un sous-ensemble $\uparrow$-fermé de $At_{\Sigma,\Pi}[X]_{/\approx}$ (i.e. $(A \in I \wedge A \leqslant B) \Rightarrow B \in I$) et on parle alors de $\mathcal{C}$-interprétations [30, 31]. Dans ce qui suit, on adopte les notations suivantes issues de [30] (E est un ensemble d'atomes) :

$$\begin{array}{lcl} [E] & = & \{\theta A \in At_{\Sigma,\Pi}[\varnothing],\ A \in E\} \\ \lceil E \rceil & = & \{A \in At_{\Sigma,\Pi}[X],\ \exists A' \in E\ A' \leqslant A\} \\ \lfloor E \rfloor & = & \{A \in At_{\Sigma,\Pi}[\varnothing],\ A \in E\} \end{array}$$

On a donc $[E] = \lfloor \lceil E \rceil \rfloor$. D'autre part, étant donné un ensemble quelconque d'atomes I, $[I]$ est une interprétation de Herbrand et $\lceil I \rceil$ est une $\mathcal{C}$-interprétation. Autrement dit I est une interprétation de Herbrand (resp. une $\mathcal{C}$-interprétation) si et seulement si $I = [I]$ (resp. $I = \lceil I \rceil$). La notion de «$\mathcal{C}$-vérité» relative à une $\mathcal{C}$-interprétation est définie comme suit :

Définition 4.7 ($\mathcal{C}$-vérité [31]) *Soit I une $\mathcal{C}$-interprétation.*

- *Un atome A est $\mathcal{C}$-vrai dans I si et seulement si $A \in I$ (i.e. la classe d'équivalence de $A \in I$).*
- *Une clause $A \leftarrow B_1, \cdots, B_q$ est $\mathcal{C}$-vraie dans I si et seulement si pour toute substitution θ, si les atomes θB_i $(1 \leqslant i \leqslant q)$ sont $\mathcal{C}$-vrais dans I, alors θA est $\mathcal{C}$-vrai dans I (en particulier une clause unité $A \leftarrow$ est $\mathcal{C}$-vraie dans I si et seulement si pour toute substitution θ, θA est $\mathcal{C}$-vrai dans I).*

6. Toutefois, il est important de noter que $\vec{t_1} \approx \vec{t_3}$ et $\vec{t_2} \approx \vec{t_4}$ n'implique pas forcément $(\vec{t_1}, \vec{t_2}) \approx (\vec{t_3}, \vec{t_4})$. Par exemple, $f(x) \approx f(y)$ et $g(x) \approx g(z)$ mais $(f(x), g(x))$ n'est pas équivalent à $(f(y), g(z))$.

Définition 4.8 *Une interprétation I est un $\mathcal{C}$-modèle d'un programme défini P si toutes les clauses de P sont $\mathcal{C}$-vraies dans I.*

Remarque. Il est possible d'éviter de contraindre les sous-ensembles de $At_{\Sigma,\Pi}[X]_{/\approx}$ à être $\uparrow$-fermés en considérant une notion différente pour la «vérité» : cette approche, présentée dans [13, 29, 30, 31], correspond à la $\mathcal{S}$-sémantique et sert à distinguer des programmes ayant les mêmes $\mathcal{C}$-modèles mais un comportement différent. Par exemple [31], les deux programmes :

$$P_1 = \{p(x) \leftarrow\ ;\ p(a) \leftarrow\} \qquad P_2 = \{p(x) \leftarrow\}$$

admettent les mêmes $\mathcal{C}$-modèles alors qu'à partir de la requête $p(x)$, le programme P_1 peut produire la réponse $\{x/a\}$ tandis qu'avec le programme P_2, seule la substitution identité peut être obtenue.

Il est possible de relier la notion de $\mathcal{C}$-modèle à la notion classique (modèle de Herbrand) :

Lemme 4.6 ([30]) *Soit I une $\mathcal{C}$-interprétation et P un programme défini. Si I est un $\mathcal{C}$-modèle de P, alors $[I]$ est un modèle de Herbrand de P.*

L'ensemble des $\mathcal{C}$-interprétations, muni de l'inclusion ensembliste, forme un treillis complet : tout ensemble E de $\mathcal{C}$-interprétations admet une borne inférieure et une borne supérieure :

$$\mathsf{glb}(E) = \bigcap_{I \in E} I \quad \text{et} \quad \mathsf{lub}(E) = \bigcup_{I \in E} I$$

Lemme 4.7 ([31]) *Soit P un programme défini.*

1. *L'intersection de $\mathcal{C}$-modèles de P est un $\mathcal{C}$-modèle de P.*
2. *Tout programme défini P admet un plus petit $\mathcal{C}$-modèle noté $\mathcal{M}_P^{\mathcal{C}}$.*
3. $\mathcal{M}_P = [\mathcal{M}_P^{\mathcal{C}}]$

Sémantique par point fixe

La sémantique par point fixe est obtenue de manière classique en associant à tout programme P un opérateur satisfaisant de «bonnes» propriétés.

Définition 4.9 ([31]) *Etant donné un programme P, $\lceil P \rceil$ est défini par :*

$$\lceil P \rceil = \{\theta C,\ C \in P,\ \theta : X \to T_\Sigma[X]\}$$

et peut être associé à l'opérateur :

$$\begin{array}{lll} T_{\lceil P \rceil}(I) = \{A \in At_{\Sigma,\Pi}[X], & \exists A \leftarrow A_1, \cdots, A_n \in \lceil P \rceil & \\ & A_i \in I \quad (1 \leqslant i \leqslant n) & \} \end{array}$$

Lemme 4.8 ([31]) *$T_{\lceil P \rceil}$ est un opérateur monotone et $\uparrow$-continu.*

Lemme 4.9 ([31]) $[T_{\lceil P \rceil}(I)] = T_{\lceil P \rceil}([I])$.

Lemme 4.10 ([31]) *Une $\mathcal{C}$-interprétation I est un $\mathcal{C}$-modèle d'un programme défini P si et seulement si $T_{\lceil P \rceil}(I) \subseteq I$.*

Théorème 4.8 ([31]) $\mathcal{M}_P^{\mathcal{C}} = \mathsf{lfp}(T_{\lceil P \rceil}) = \mathsf{Ind}(T_{\lceil P \rceil}) = T_{\lceil P \rceil}^{\uparrow\omega}$

Sémantique opérationnelle

L'équivalent de l'ensemble des succès (fermés) de la théorie «classique» est défini comme suit :

Définition 4.10 $S_P^{\mathcal{C}} = \{A \in At_{\Sigma,\Pi}[X],\ A \stackrel{*,\theta}{\to}_P \Box \ \text{ et } \theta A = A\}$

et les théorèmes de validité et de complétude s'énoncent comme suit.

Théorème 4.9 ([30]) $S_P^{\mathcal{C}} = \mathcal{M}_P^{\mathcal{C}} = T_{\lceil P \rceil}^{\uparrow\omega}$

Théorème 4.10 ($\mathcal{C}$-validité [31]) *Si il existe une réfutation :*

$$R = A_1, \cdots, A_q \stackrel{*,\theta}{\to}_P \Box$$

alors il existe un ensemble d'atomes $\{A'_1, \cdots, A'_q\} \subseteq \mathcal{M}_P^{\mathcal{C}}$ et une substitution θ' tels que $\theta' = \mathsf{mgu}((A_1, \cdots, A_q), (A'_1, \cdots, A'_q))$ et $\theta'_{\restriction var(R)} = \theta_{\restriction var(R)}$.

Théorème 4.11 ($\mathcal{C}$-complétude [31]) *Soit R la requête $A_1, \cdots, A_q$. Si il existe un ensemble d'atomes $\{A'_1, \cdots, A'_q\} \subseteq \mathcal{M}_P^{\mathcal{C}}$ et une substitution θ' tels que $\theta' = \mathsf{mgu}((A_1, \cdots, A_q), (A'_1, \cdots, A'_q))$, alors il existe une réfutation*

$$R = A_1, \cdots, A_q \stackrel{*,\theta}{\to}_P \Box$$

telle que $\theta'_{\restriction var(R)} \geq \theta_{\restriction var(R)}$.

4.2.4 Arbres de preuve et SLD-preuves

Un autre manière de caractériser les éléments d'un ensemble défini co-inductivement utilise la notion classique d'arbres de preuve. Dans ce paragraphe, après avoir introduit cette notion, nous montrons comment à partir d'un arbre de preuve pour un ensemble de règles Φ, il est possible de construire une SLD-dérivation avec le programme Φ.

Arbres de preuve

Définition 4.11 (Arbres de preuve) *Soit Φ un ensemble de règles sur un univers $\mathcal{B}$.*

- *Un arbre de preuve de $x \in \mathcal{B}$ pour Φ est un arbre fini ou infini de racine x tel que pour tout noeud z de l'arbre admettant $z_1, \cdots, z_q$ pour fils, $z \leftarrow z_1, \cdots, z_q \in \Phi$ (en particulier, si z est un feuille, alors $z \leftarrow \in \Phi$).*
- *Un arbre de preuve partiel de $x \in \mathcal{B}$ pour Φ est un arbre de preuve de x dont les feuilles ne correspondent pas nécessairement à des règles de la forme $z \leftarrow \in \Phi$.*

La notion d'arbres de preuve est directement liée à la notion d'ensemble co-inductif par le lemme suivant.

Lemme 4.11 *Soit Φ un ensemble de règles sur un univers $\mathcal{B}$ et $x \in \mathcal{B}$. $x \in \mathsf{CoInd}(\Phi)$ si et seulement si x est racine d'un arbre de preuve pour Φ.*

Preuve. ($\Rightarrow$). Soit $x \in \mathsf{CoInd}(\Phi)$, d'après le théorème 1.3, on a $x \in \mathsf{gfp}(T_\Phi)$ et il vient $x \in T_\Phi(\mathsf{gfp}(T_\Phi))$. Il existe donc une règle $x \leftarrow x_1, \cdots, x_q \in \Phi$ telle que $\{x_1, \cdots, x_q\} \subseteq \mathsf{gfp}(T_\Phi)$. On peut donc construire un arbre de racine x ayant $\{x_1, \cdots, x_q\}$ pour fils et en réitérant ce processus sur l'ensemble des fils de x qui sont dans $\mathsf{CoInd}(\Phi)$, on obtient un arbre de preuve de x pour Φ. ($\Leftarrow$). Soit x la racine d'un arbre de preuve pour Φ et Z l'ensemble des noeuds de l'arbre. On a $Z \subseteq T_\Phi(Z)$. En effet, si $z \in Z$, alors il existe $z \leftarrow z_1, \cdots, z_q \in \Phi$ où $\{z_1, \cdots, z_q\}$ est l'ensemble des fils de z. On a donc $\{z_1, \cdots, z_q\} \subseteq Z$ et donc $z \in T_\Phi(Z)$. Puisque $Z \subseteq T_\Phi(Z)$, Z est Φ-dense et, par définition, il vient $x \in \mathsf{CoInd}(\Phi)$. ◀

Signalons que lorsque l'arbre de preuve de x pour Φ est fini, on peut montrer que $x \in \mathsf{Ind}(\Phi)$. Aussi, le théorème de complétude de la SLD-résolution pourrait être établi en utilisant un résultat similaire à celui exprimé par le théorème 4.12 prouvé à la fin de ce paragraphe. La preuve de ce théorème utilise les trois lemmes ci-dessous.

Renommage et unification

Lemme 4.12 *Soit A_1 et A_2 deux atomes ne partageant pas de variables. Si pour une substitution θ, satisfaisant $dom(\theta) \subseteq var(A_2)$, $A_1 = \theta A_2$, alors θ est un unificateur principal de A_1 et A_2.*

Preuve. Tout d'abord puisque $var(A_1) \cap var(A_2) = \varnothing$ et puisque $dom(\theta) \subseteq var(A_2)$, on a $\theta A_1 = A_1 = \theta A_2$. θ est donc bien un unificateur de A_1 et A_2. D'autre part, si $v \in range(\theta)$, il existe une variable $y \in dom(\theta) \subseteq var(A_2)$ telle que $v \in \theta y$. On a donc $v \in \theta A_2 = A_1$ et donc $v \notin dom(\theta)$ puisque $\theta A_1 = A_1$ ce qui permet de conclure à l'idempotence de θ. De plus, il est clair que θ est bien supportée et couverte par A_1 et A_2. Montrons à présent que θ est un unificateur le plus général. Soit μ un unificateur de A_1 et A_2. Puisque $\theta A_1 = A_1$, on a $\mu\theta A_1 = \mu A_2$ et il vient $\mu\theta A_2 = \mu A_2$. Montrons alors que pour toute variable v, on a $\mu\theta v = \mu v$. Si $v \notin dom(\theta)$ alors $\mu\theta v = \mu v$ est immédiat, sinon, puisque $dom(\theta) \subseteq var(A_2)$ on peut conclure puisque $\mu\theta A_2 = \mu A_2$. On a donc bien $\theta \leqslant \mu$. ◀

Lemme 4.13 *Soit $Z \subset X$ et C une clause $A \leftarrow B_1, \cdots, B_q$. Il existe une clause C' et une substitution de renommage idempotente et injective sur son domaine ρ vérifiant :*

$$\begin{array}{ll} var(C') \cap (var(C) \cup Z) = \varnothing & dom(\rho) = var(C') \\ \rho C' = C & range(\rho) = var(C) \end{array}$$

PREUVE. Si $var(C) = \{x_1, \cdots, x_n\}$ et si $\{y_1, \cdots, y_n\}$ est un ensemble de variables distinctes tel que $\{y_1, \cdots, y_n\} \cap (var(C) \cup Z) = \varnothing$, alors C' et ρ peuvent s'écrire :

$$C' = \left[\begin{array}{ccc} x_1 & \cdots & x_n \\ y_1 & \cdots & y_n \end{array} \right] C \qquad \rho = \left[\begin{array}{ccc} y_1 & \cdots & y_n \\ x_1 & \cdots & x_n \end{array} \right]$$

et vérifient bien les propriétés énoncées. ◀

Lemme 4.14 *Si Z est un ensemble de variables, C_T une clause et r_0 une substitution de renommage telle que $range(r_0) \cap var(C_T) = \varnothing$, alors il existe une transition :*

$$r_0 C_T^+ \stackrel{C,\theta}{\rightarrow} \theta C^-$$

où C est une clause telle que $var(C) \cap (var(r_0 C_T) \cup Z) = \varnothing$ et θ est une substitution de renommage idempotente injective sur son domaine telle que $dom(\theta) = var(C^+)$. D'autre part, il existe une substitution de renommage r vérifiant $range(r) = var(C^-) \backslash var(C^+)$ telle que $r r_0 C_T^- = \theta C^-$.

PREUVE. D'après le lemme 4.13, il existe une clause C et une substitution de renommage idempotente et injective sur son domaine ρ vérifiant :

$$\begin{array}{ll} var(C) \cap (var(r_0 C_T) \cup Z) = \varnothing & dom(\rho) = var(C) \\ \rho C = r_0 C_T & range(\rho) = var(r_0 C_T) \end{array}$$

D'après le lemme 4.12, la restriction θ de ρ aux variables de C^+ est un unificateur principal de $r_0 C_T^+$ et C^+ ce qui permet de construire la transition :

$$r_0 C_T^+ \stackrel{C,\theta}{\rightarrow} \theta C^-$$

Si la substitution ρ s'écrit :

$$\rho = \left[\begin{array}{ccccccccc} x_1^+ & \ldots & x_{n_1}^+ & x_1^\pm & \ldots & x_{n_2}^\pm & x_1^- & \ldots & x_{n_3}^- \\ y_1^+ & \ldots & y_{n_1}^+ & y_1^\pm & \ldots & y_{n_2}^\pm & y_1^- & \ldots & y_{n_3}^- \end{array} \right]$$

avec :

$$\begin{array}{l}\{x_1^+,...,x_{n_1}^+\} = var(C^+)\backslash var(C^-)\\ \{x_1^\pm,...,x_{n_2}^\pm\} = var(C^+)\cap var(C^-)\\ \{x_1^-,...,x_{n_3}^-\} = var(C^-)\backslash var(C^+)\end{array}$$

on peut définir la substitution de renommage $r_1 = rr_0$ par :

$$r_1 = \begin{bmatrix} y_1^- & ... & y_{n_3}^- \\ x_1^- & ... & x_{n_3}^- \end{bmatrix} r_0$$

En effet, on a :

$$\begin{array}{rcll} r_1 C_T^- &=& \begin{bmatrix} y_1^- & ... & y_{n_3}^- \\ x_1^- & ... & x_{n_3}^- \end{bmatrix} r_0 C_T^- & \\ &=& \begin{bmatrix} y_1^- & ... & y_{n_3}^- \\ x_1^- & ... & x_{n_3}^- \end{bmatrix} \rho C^- & \\ &=& \begin{bmatrix} y_1^- & ... & y_{n_3}^- \\ x_1^- & ... & x_{n_3}^- \end{bmatrix} \begin{bmatrix} x_1^\pm & ... & x_{n_2}^\pm & x_1^- & ... & x_{n_3}^- \\ y_1^\pm & ... & y_{n_2}^\pm & y_1^- & ... & y_{n_3}^- \end{bmatrix} C^- & \\ &=& \begin{bmatrix} x_1^\pm & ... & x_{n_2}^\pm \\ y_1^\pm & ... & y_{n_2}^\pm \end{bmatrix} C^- & = \theta C^- \end{array}$$

puisque ρ est injective sur son domaine. ◀

SLD-Preuves

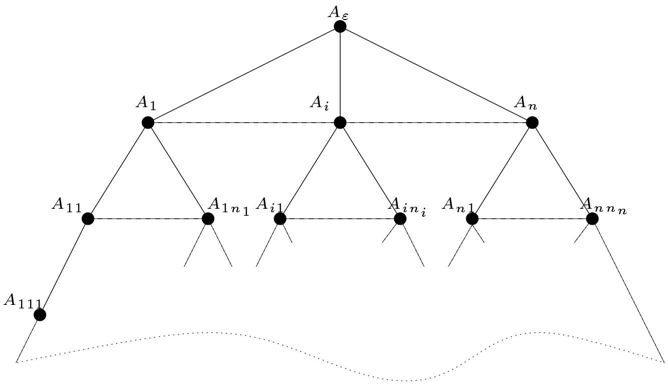
Nous pouvons à présent relier la notion d'arbres de preuve avec celle de SLD-preuves définie comme suit.

Définition 4.12 (SLD-preuves) *Une SLD-preuve est soit une réfutation soit une dérivation infinie équitable.*

Théorème 4.12 *Soit P un programme et A un atome. Si $A \in \mathsf{CoInd}(P)$, alors il existe une SLD-preuve à partir de A avec le programme P telle qu'à chaque étape i, l'unificateur θ_i mis en jeu est un renommage, injectif sur son domaine, de toutes les variables apparaissant dans la tête de la clause utilisée.*

Preuve. Si $A \in \mathsf{CoInd}(P)$, alors d'après le lemme 4.11, A est racine d'un arbre de preuve T pour P. Les noeuds de cet arbre peuvent être indicés par des éléments de $\mathbb{N}^\star$: l'indice d'un noeud correspond aux étiquettes rencontrées sur le chemin qui mène de la racine à ce noeud ; les arcs sont étiquetés de gauche à droite par des entiers successifs à partir de 1 (ε désigne le mot vide). Le parcours en largeur de T fournit la liste de noeuds $\mathcal{L}$:

$$A_\varepsilon, A_1, ..., A_n, A_{11}, ..., A_{1n_1}, ..., A_{i1}, ..., A_{in_i}, ..., A_{n1}, ..., A_{nn_n}, A_{111}, ...$$



Par définition, pour chaque noeud $A_{\vec{\imath}}$ il existe une clause $C_{T,\vec{\imath}} \in P$ qui s'écrit :

$$A_{\vec{\imath}} \leftarrow A_{\vec{\imath}1}, \cdots, A_{\vec{\imath}n_{\vec{\imath}}}$$

On note $\prec_\ell$ l'ordre lexicographique sur $\mathbb{N}^\star$ et $|\vec{\imath}|$ la longueur de $\vec{\imath} \in \mathbb{N}^\star$. L'ensemble des indices des noeuds de l'arbre peut aussi être ordonné par la relation $\prec$ définie par :

$$\vec{\imath} \prec \vec{\jmath} \Leftrightarrow A_{\vec{\imath}} \text{ se trouve avant } A_{\vec{\jmath}} \text{ dans } \mathcal{L}$$

De manière équivalente, $\prec$ peut être défini par :

$$\vec{\imath} \prec \vec{\jmath} \Leftrightarrow ((|\vec{\imath}| < |\vec{\jmath}|) \vee (|\vec{\imath}| = |\vec{\jmath}| \wedge \vec{\imath} \prec_\ell \vec{\jmath}))$$

Etant donné un indice $\vec{\imath} \in \mathbb{N}^\star$ correspondant à un noeud de l'arbre, on note $\overleftarrow{\imath}$ l'indice du noeud qui précède $A_{\vec{\imath}}$ dans $\mathcal{L}$. Plus formellement :

$$\overleftarrow{\imath} = \max{}^{\prec}\{\vec{\jmath},\ \vec{\jmath} \prec \vec{\imath}\}$$

Enfin, on note $A_{\triangleright k}$ (resp. $A_{k\triangleleft}$) le noeud le plus à droite (resp. le plus à gauche) à la profondeur k :

$$\forall k \in \mathbb{N} \quad \begin{array}{l} \triangleright k = \max{}^{\prec_\ell}\{\vec{\imath},\ |\vec{\imath}| = k\} \\ k\triangleleft = \min{}^{\prec_\ell}\{\vec{\imath},\ |\vec{\imath}| = k\} \end{array}$$

Soit Z_T l'ensemble, éventuellement infini, des variables apparaissant T :

$$Z_T = \bigcup var(C_{T,\vec{\imath}})$$

D'après le lemme 4.14, étant donnés une clause $C_{T,\vec{\imath}} \in P$, une substitution de renommage $r_0^{\vec{\imath}}$, telle que $range(r_0^{\vec{\imath}}) \cap var(C_{T,\vec{\imath}}) = \varnothing$, et un ensemble de variables $Z_{\vec{\imath}}$, il existe une substitution $\theta_{\vec{\imath}}$, une clause $C_{\vec{\imath}}$ et une substitution de renommage $r_1^{\vec{\imath}} = r^{\vec{\imath}} r_0^{\vec{\imath}}$ vérifiant :

$$\begin{array}{cc} var(C_{\vec{\imath}}) \cap (var(r_0^{\vec{\imath}} C_{T,\vec{\imath}}) \cup Z_{\vec{\imath}}) = \varnothing & r_1^{\vec{\imath}} C^-_{T,\vec{\imath}} = \theta_{\vec{\imath}} C^-_{\vec{\imath}} \\ dom(\theta_{\vec{\imath}}) = var(C^+_{\vec{\imath}}) & range(r^{\vec{\imath}}) = var(C^-_{\vec{\imath}}) \backslash var(C^+_{\vec{\imath}}) \end{array}$$

où $\theta_{\vec{\imath}}$ est une substitution de renommage idempotente et injective sur son domaine correspondant à un unificateur principal de $C_{\vec{\imath}}^{+}$ et $r_0^{\vec{\imath}} C_{T,\vec{\imath}}^{+}$. Dans ce qui suit, on note $\mathcal{T}(C_{T,\vec{\imath}}, r_0^{\vec{\imath}}, Z_{\vec{\imath}})$ la transition :

$$r_0^{\vec{\imath}} C_{T,\vec{\imath}}^{+} \overset{C_{\vec{\imath}},\theta_{\vec{\imath}},Z_{\vec{\imath}}}{\rightarrow_P} \theta_{\vec{\imath}} C_{\vec{\imath}}^{-}$$

A partir de $\mathcal{L}$, il est possible de définir la liste de transitions :

$$\mathcal{L}_D = t_\varepsilon, t_1, \cdots, t_n, t_{11}, \cdots, t_{1n_1}, \cdots, t_{i1}, \cdots, t_{in_i}, \cdots, t_{n1}, \cdots, t_{nn_n}, t_{111}, \cdots$$

comme suit :

$$\left\{ \begin{array}{ll} t_\varepsilon = \mathcal{T}(C_{T,\varepsilon}, s_{id}, Z_T) & \\ t_{\vec{\imath}k} = \mathcal{T}\left(C_{T,\vec{\imath}k}, r_0^{\vec{\imath}k}, Z_{\vec{\imath}k}\right) & \begin{array}{l} 1 \leqslant k \leqslant n_{\vec{\imath}} \\ r_0^{\vec{\imath}k} = r_1^{\vec{\imath}} \\ Z_{\vec{\imath}k} = Z_T \cup \bigcup\limits_{\vec{\jmath} \prec \vec{\imath}k} var(C_{\vec{\jmath}}) \end{array} \end{array} \right.$$

Pour vérifier que la définition de la liste des transitions est correcte, il faut montrer que :

$$\forall \vec{\imath} \quad range(r_0^{\vec{\imath}}) \cap var(C_{T,\vec{\imath}}) = \varnothing$$

Plus généralement, nous allons montrer que :

$$\forall \vec{\imath} \quad range(r_0^{\vec{\imath}}) \cap Z_T = \varnothing$$

On procède par induction sur $\vec{\imath}$. Supposons la propriété vérifiée pour tout $\vec{\jmath} \prec \vec{\imath}$. Si $|\vec{\imath}| = 0$, alors la propriété est vérifiée puisque $range(s_{id}) \cap var(C_{T,\varepsilon}) = \varnothing$. Sinon, $\vec{\imath}$ s'écrit $\vec{\jmath}k$ et, par définition, on a :

$$r_0^{\vec{\jmath}k} = r_1^{\vec{\jmath}} = r^{\vec{\jmath}} r_0^{\vec{\jmath}}$$

où $r^{\vec{\jmath}}$ est une substitution vérifiant $range(r^{\vec{\jmath}}) \cap Z_T = \varnothing$ puisque :

$$range(r^{\vec{\jmath}}) \subseteq var(C_{\vec{\jmath}}^{-}) \backslash var(C_{\vec{\jmath}}^{+}) \qquad Z_T \subseteq Z_{\vec{\jmath}} \qquad var(C_{\vec{\jmath}}) \cap Z_{\vec{\jmath}} = \varnothing$$

L'hypothèse d'induction permet alors de conclure puisque :

$$range(r_0^{\vec{\jmath}k}) = range(r_1^{\vec{\jmath}}) \subseteq (range(r_0^{\vec{\jmath}}) \cup range(r^{\vec{\jmath}})) \quad var(C_{T,\vec{\jmath}k}) \subseteq Z_T$$

On peut à présent construire la dérivation équitable à partir des transitions ainsi définies :

$$A = A_\varepsilon \overset{C_\varepsilon,\theta_\varepsilon,Z_T}{\rightarrow_P} R \overset{C_1,\theta_1,Z_1}{\rightarrow_P} R_1 \rightarrow \ldots \rightarrow R_{n-1} \overset{C_n,\theta_n,Z_n}{\rightarrow_P} R_n \overset{C_{11},\theta_{11},Z_{11}}{\rightarrow_P} R_{11} \rightarrow \ldots$$

Montrons que cette dérivation est bien définie et vérifie bien les propriétés énoncées.

(1). $\forall \vec{\imath} \quad var(C_{\vec{\imath}}) \cap \left(var(A) \cup \bigcup_{\vec{\jmath} \prec \vec{\imath}} var(C_{\vec{\jmath}}) \right) = \varnothing$
Immédiat par définition de $Z_{\vec{\imath}}$.

(2). La transition $A \overset{C_\varepsilon, \theta_\varepsilon, Z_T}{\rightarrow}_P R$ est bien définie.
Immédiat puisqu'il s'agit de la transition t_ε.

(3). $\forall \vec{\imath} \quad range(r_1^{\vec{\imath}}) \subseteq \bigcup_{\vec{\jmath} \preceq \vec{\imath}} \left(var(C_{\vec{\jmath}}^-) \backslash var(C_{\vec{\jmath}}^+) \right)$
Induction sur $\vec{\imath}$.
• Si $|\vec{\imath}| = 0$, alors on a bien :

$$range(r_1^\varepsilon) = range(r_0^\varepsilon r^\varepsilon) = range(r^\varepsilon) = var(C_\varepsilon^-) \backslash var(C_\varepsilon^+)$$

• Si $\vec{\imath} = \vec{\jmath}k$, alors l'hypothèse d'induction permet de conclure puisque :

$$r_1^{\vec{\jmath}k} = r^{\vec{\jmath}k} r_0^{\vec{\jmath}k} = r^{\vec{\jmath}k} r_1^{\vec{\jmath}} \qquad range(r^{\vec{\jmath}k}) = var(C_{\vec{\jmath}k}^-) \backslash var(C_{\vec{\jmath}k}^+)$$

(4). $\forall p \in \mathbb{N} \left\{ \begin{array}{ll} \forall \vec{\imath} \; ((p+1)\triangleleft \preceq \vec{\imath} \preceq \triangleright(p+1)) & r_0^{\vec{\imath}} A_{\vec{\imath}} \in R_{\triangleright p} \\ \forall \vec{\imath} \; ((p+1)\triangleleft \prec \vec{\imath} \preceq \triangleright(p+1)) & r_0^{\vec{\imath}} A_{\vec{\imath}} \in R_{\overleftarrow{\imath}} \end{array} \right.$
(en particulier $r_0^{(p+1)\triangleleft} A_{(p+1)\triangleleft} \in R_{\triangleright p}$)
Induction sur p.
• Si $p = 0$, alors pour tout k tel que $1 \leqslant k \leqslant n$, on a :

$$r_0^k A_k = r_1^\varepsilon A_k \in \theta_\varepsilon C_\varepsilon^- = R_\varepsilon = R_{\triangleright 0}$$

Montrons que pour tout k $(1 < k \leqslant n)$, on a $r_0^k A_k = r_1^\varepsilon A_k \in R_{k-1}$. Puisque $R_\varepsilon = \theta_\varepsilon C_\varepsilon^- = r_1^\varepsilon C_{T,\varepsilon}^-$, on sait que $r_1^\varepsilon A_k \in R_\varepsilon$. D'autre part, pour tout m $(1 \leqslant m \leqslant k-1)$, on a :

$$dom(\theta_m) = var(C_m^+) \quad \text{et} \quad var(C_m) \cap \left(Z_T \cup \bigcup_{j<m} C_j \right) = \varnothing$$

Enfin, puisque $r_1^\varepsilon = r^\varepsilon r_0^\varepsilon = r^\varepsilon$ avec $range(r^\varepsilon) = var(C_\varepsilon^-) \backslash var(C_\varepsilon^+)$, on a :

$$var(r_1^\varepsilon A_k) \subseteq (Z_T \cup var(C_\varepsilon))$$

Par conséquent, $var(r_1^\varepsilon A_k) \cap var(C_m) = \varnothing$ et il vient :

$$\theta_m r_1^\varepsilon A_k = r_1^\varepsilon A_k$$

On peut alors conclure, puisque à chaque étape m, c'est l'atome $r_1^\varepsilon A_m$ qui est sélectionné et l'unificateur utilisé θ_m n'affecte pas $r_1^\varepsilon A_k$ qui est donc bien

présent dans la requête R_m.
• Si $p > 0$, alors, par hypothèse d'induction, on a :

$$\begin{cases} \forall \vec{\imath}\ (p\triangleleft \preceq \vec{\imath} \preceq \triangleright p) & r_0^{\vec{\imath}} A_{\vec{\imath}} \in R_{\triangleright(p-1)} \\ \forall \vec{\imath}\ (p\triangleleft \prec \vec{\imath} \preceq \triangleright p) & r_0^{\vec{\imath}} A_{\vec{\imath}} \in R_{\overleftarrow{\imath}} \end{cases}$$

et il suffit alors de montrer que :

$$\forall \vec{\jmath}\ (p\triangleleft \prec \vec{\jmath} \preceq \triangleright p) \quad dom(\theta_{\vec{\jmath}}) \cap var\left(\bigcup_{\vec{u} \prec \vec{\jmath}} \bigcup_{k=1}^{n_{\vec{u}}} r_0^{\vec{u}k} A_{\vec{u}k}\right) = \varnothing$$

En effet, si $\vec{\jmath}$ est tel que $p\triangleleft \prec \vec{\jmath} \preceq \triangleright p$, on sait que :

$$dom(\theta_{\vec{\jmath}}) = var(C_{\vec{\jmath}}^{+}) \quad \text{et} \quad var(C_{\vec{\jmath}}) \cap \left(Z_T \cup \bigcup_{\vec{u} \prec \vec{\jmath}} C_{\vec{u}}\right) = \varnothing$$

D'autre part, on a :

$$var\left(\bigcup_{\vec{u} \prec \vec{\jmath}} \bigcup_{k=1}^{n_{\vec{u}}} r_0^{\vec{u}k} A_{\vec{u}k}\right) = var\left(\bigcup_{\vec{u} \prec \vec{\jmath}} \bigcup_{k=1}^{n_{\vec{u}}} r_1^{\vec{u}} A_{\vec{u}k}\right) = var\left(\bigcup_{\vec{u} \prec \vec{\jmath}} r_1^{\vec{u}} C_{T,\vec{u}}^{-}\right)$$

et puisque :

$$var\left(\bigcup_{\vec{u} \prec \vec{\jmath}} r_1^{\vec{u}} C_{T,\vec{u}}^{-}\right) \subseteq \bigcup_{\vec{u} \prec \vec{\jmath}} \Big(range(r_1^{\vec{u}}) \cup var(C_{T,\vec{u}}^{-})\Big)$$

(1) et (3) permettent alors de conclure.

(5). La dérivation est construite à partir de la liste des noeuds de l'arbre obtenue lors d'un parcours en largeur de cet arbre, et correspond donc à une SLD-preuve, puisque soit l'arbre de preuve est fini et dans ce cas la dérivation correspond à une réfutation, soit l'arbre de preuve est un arbre fini en largeur (puisque le corps de chaque clause contient un nombre fini d'atomes) et infini en profondeur, et le parcours en largeur garantit que tous les atomes sont sélectionnés à une étape de la dérivation. ◀

4.3 SLD-preuves

Nous avons vu au paragraphe précédent que la présence d'éléments infinis dans l'univers du discours ne permettait pas de définir une sémantique des dérivations infinies basée sur la notion de plus grand point fixe. Aussi, les deux paragraphes qui suivent définissent une sémantique valide et complète pour deux classes de dérivations qui ne construisent pas de termes infinis. Ces dérivations peuvent être utiles pour décrire le comportement de certains «programmes non-déterministes», aussi appelés «*flowgraph programs*» dans [7]. En effet, K.R. Apt et M.H. Van Emden établissent une correspondance entre les exécutions possibles d'un tel programme, décrite par un graphe G, et les dérivations, éventuellement infinies, engendrées par un programme défini $P(G)$ obtenu à partir de G. La définition de $P(G)$ ne faisant intervenir aucun symbole de fonction d'arité strictement positive, il est clair que les dérivations engendrées par $P(G)$ ne construisent pas de termes infinis. Dans le domaine des bases de données déductives, les programmes DATALOG utilisés constituent une classe de programmes définis exprimés sur une signature fonctionnelle ne contenant que des constantes. Ici encore, les dérivations engendrées par de tels programmes ne construisent pas de termes infinis.

4.3.1 SLD-preuves directes

On s'intéresse dans ce paragraphe aux dérivations qui «prouvent sans calculer». Ces dérivations peuvent être étudiées en considérant un cas particulier de la $\mathcal{C}$-sémantique.

Définition 4.13 (SLD-preuve directe) *Une SLD-preuve directe est une SLD-preuve de la forme :*

$$R_0 \overset{C_1,\theta_1}{\to_P} R_1 \to_P \cdots \to_P R_{i-1} \overset{C_i,\theta_i}{\to_P} R_i \to_P \cdots$$

telle que pour tout $i \geq 1$, $dom(\theta_i) \subseteq var(C_i^+)$. En particulier, une SLD-réfutation directe est une SLD-preuve directe finie se terminant par la requête vide.

Sémantique déclarative

Il est possible d'étendre la notion de $\mathcal{C}$-vérité comme suit :

Définition 4.14 ($\mathcal{C}^+$-vérité) *Soit I une $\mathcal{C}$-interprétation.*

- *Un atome A est $\mathcal{C}^+$-vrai dans I si et seulement si $A \in I$ (i.e. la classe d'équivalence de $A \in I$).*
- *Une clause $A \leftarrow B_1, \cdots, B_q$ est $\mathcal{C}^+$-vraie dans I si et seulement si pour toute substitution θ vérifiant $dom(\theta) \subseteq var(A)$, si les atomes θB_i $(1 \leqslant i \leqslant q)$ sont $\mathcal{C}^+$-vrais dans I, alors θA est $\mathcal{C}^+$-vrai dans I.*

On vérifie facilement que cette nouvelle notion de vérité étend la $\mathcal{C}$-vérité puisque :

Lemme 4.15 *Soit I une $\mathcal{C}$-interprétation.*

1. *Un atome est $\mathcal{C}^+$-vrai dans I si et seulement si il est $\mathcal{C}$-vrai dans I.*

2. *Si une clause est $\mathcal{C}$-vraie dans I alors elle est $\mathcal{C}^+$-vraie dans I.*

La notion de $\mathcal{C}^+$-modèle est définie de manière classique :

Définition 4.15 *Une interprétation I est un $\mathcal{C}^+$-modèle d'un programme défini P si toutes les clauses de P sont $\mathcal{C}^+$-vraies dans I.*

Lemme 4.16 *Soit I une $\mathcal{C}$-interprétation et P un programme défini. Si I est un $\mathcal{C}$-modèle de P, alors I est un $\mathcal{C}^+$-modèle de P.*

PREUVE. Immédiat d'après le lemme 4.15. ◀

Par contre, tous les $\mathcal{C}^+$-modèles d'un programme P ne sont pas forcément des $\mathcal{C}$-modèles de P. Considérons par exemple le programme :

$$P = \{p(x) \leftarrow p(y) \ ; \ p(f(w)) \leftarrow\} \tag{4.20}$$

La $\mathcal{C}$-interprétation définie par $I = \lceil p(f(z)) \rceil$ est clairement un $\mathcal{C}^+$-modèle de P, puisque pour toute substitution θ telle que $dom(\theta) \subseteq var(p(x))$, $\theta p(y) = p(y) \notin I$. Par contre, I n'est pas un $\mathcal{C}$-modèle de P, puisque si l'on considère la substitution $\theta = \{y/f(y)\}$, on a bien $\theta p(y) = p(f(y)) \in I$ mais $\theta p(x) = p(x) \notin I$. De même, la correspondance entre un $\mathcal{C}$-modèle et un modèle de Herbrand, établie par le lemme 4.6, n'est plus vérifiée pour les $\mathcal{C}^+$-modèles. En effet, si l'on reconsidère le programme défini en (4.20), $I = \lceil p(f(z)) \rceil$ est bien un $\mathcal{C}^+$-modèle de P mais $[[\lceil p(f(z)) \rceil]]$ n'est pas un modèle de Herbrand de P puisque pour la substitution $\theta = \{x/k \ ; \ y/f(k)\}$, on a $\theta p(y) = p(f(k)) \in [[\lceil p(f(z)) \rceil]]$ mais $\theta p(x) = p(k) \notin [[\lceil p(f(z)) \rceil]]$.

Lemme 4.17 *Soit P un programme défini.*

1. *L'intersection de $\mathcal{C}^+$-modèles de P est un $\mathcal{C}^+$-modèle de P.*
2. *Tout programme défini P admet un plus petit $\mathcal{C}^+$-modèle noté $\mathcal{M}_P^{\mathcal{C}^+}$.*

PREUVE. (1). Soit $\{I_1, \cdots, I_i, \cdots\}$ un ensemble de $\mathcal{C}^+$-modèles de P. Soit $A \leftarrow B_1, \cdots, B_q$ une clause de P et θ une substitution telle que $dom(\theta) \subseteq var(A)$. Supposons que :

$$\{\theta B_1, \cdots, \theta B_q\} \subseteq \bigcap_{i \geq 1} I_i$$

alors on a $\forall i \geq 1 \quad \{\theta B_1, \cdots, \theta B_q\} \subseteq I_i$ et puisque les I_i sont des $\mathcal{C}^+$-modèles de P et que $dom(\theta) \subseteq var(A)$, on peut conclure puisqu'il vient :

$$\theta A \in \bigcap_{i \geq 1} I_i$$

(2). $At_{\Sigma,\Pi}[X]_{/\approx}$ est clairement un $\mathcal{C}^+$-modèle de P. L'ensemble des $\mathcal{C}^+$-modèles de P est donc non vide et l'intersection de tous les $\mathcal{C}^+$-modèles de P correspond au plus petit $\mathcal{C}^+$-modèle de P. ◀

Sémantique par point fixe

Définition 4.16 *Etant donné un programme P, $\lceil P \rceil^+$ est défini par :*

$$\lceil P \rceil^+ = \{\theta C,\ C \in P \quad et \quad dom(\theta) \subseteq var(C^+)\}$$

et peut être associé à l'opérateur :

$$T_{\lceil P \rceil^+}(I) = \{A \in At_{\Sigma,\Pi}[X],\ \begin{array}{l} \exists A \leftarrow A_1, \cdots, A_n \in \lceil P \rceil^+ \\ A_i \in I \quad (1 \leqslant i \leqslant n) \end{array} \}$$

Cet opérateur vérifie les propriétés classiques.

Lemme 4.18 *$T_{\lceil P \rceil^+}$ est un opérateur monotone et $\uparrow$-continu.*

Preuve. Puisque $T_{\lceil P \rceil^+}$ est l'opérateur associé à l'ensemble de règles $\lceil P \rceil^+$, le lemme 1.2 permet d'établir la monotonie de $T_{\lceil P \rceil^+}$. D'autre part, puisque $\lceil P \rceil^+$ est finitaire, d'après le lemme 1.6, l'opérateur $T_{\lceil P \rceil^+}$ est finitaire et le lemme 1.5 permet alors d'établir la $\uparrow$-continuité de $T_{\lceil P \rceil^+}$. ◀

Lemme 4.19 *Si I est un ensemble $\uparrow$-fermé d'atomes, alors $T_{\lceil P \rceil^+}(I)$ est aussi un ensemble $\uparrow$-fermé.*

Preuve. Soit un atome $A \in T_{\lceil P \rceil^+}(I)$. Par définition, il existe une clause $A' \leftarrow B_1, \cdots, B_q$ de P telle que pour une substitution θ vérifiant $dom(\theta) \subseteq var(A')$ on ait $\theta A' = A$ et $\{\theta B_1, \cdots, \theta B_q\} \subseteq I$. Soit A_0 un atome tel que $A \leqslant A_0$. Il existe une substitution μ vérifiant $dom(\mu) \subseteq var(A)$ telle que $\mu A = A_0$. En considérant la restriction de $\mu\theta$ aux variables apparaissant dans A', il vient $\mu\theta A' = A_0$ et on peut alors conclure puisque $\{\mu\theta B_1, \cdots, \mu\theta B_q\} \subseteq I$ car I est $\uparrow$-fermé. ◀

Lemme 4.20 $\mathsf{Ind}(T_{\lceil P \rceil^+}) = \mathsf{lfp}(T_{\lceil P \rceil^+}) = T_{\lceil P \rceil^+}^{\uparrow\omega}$

Preuve. D'après le lemme 4.18, $T_{\lceil P \rceil^+}$ est un opérateur monotone et $\uparrow$-continu et le théorème 1.2 permet de conclure. ◀

Lemme 4.21 *Une $\mathcal{C}$-interprétation I est un $\mathcal{C}^+$-modèle d'un programme défini P si et seulement si $T_{\lceil P \rceil^+}(I) \subseteq I$.*

Preuve. ($\Rightarrow$). Soit I un $\mathcal{C}^+$-modèle de P et $A \in T_{\lceil P \rceil^+}(I)$. Par définition, il existe une clause $A \leftarrow B_1, \cdots, B_q \in \lceil P \rceil^+$ telle que $\{B_1, \cdots, B_q\} \subseteq I$. Puisque I est un $\mathcal{C}^+$-modèle de P, on a alors $A \in I$ ce qui permet de conclure. ($\Leftarrow$). Soit I une $\mathcal{C}$-interprétation telle que $T_{\lceil P \rceil^+}(I) \subseteq I$. Si $A \leftarrow B_1, \cdots, B_q$ est une clause de P et θ est une substitution telle que $dom(\theta) \subseteq var(A)$, alors si $\{\theta B_1, \cdots, \theta B_q\} \subseteq I$, on a $\theta A \in T_{\lceil P \rceil^+}(I)$ et puisque $T_{\lceil P \rceil^+}(I) \subseteq I$ il vient $\theta A \in I$. I est donc bien un $\mathcal{C}^+$-modèle de P. ◀

Théorème 4.13 $\mathcal{M}_P^{\mathcal{C}^+} = \mathsf{lfp}(T_{\lceil P \rceil^+}) = \mathsf{Ind}(T_{\lceil P \rceil^+}) = T_{\lceil P \rceil^+}^{\uparrow\omega}$

PREUVE. Par définition, $\mathcal{M}_P^{\mathcal{C}^+}$ est l'intersection de tous les $\mathcal{C}^+$-modèles de P, qui d'après le lemme 4.21 sont des $\mathcal{C}$-interprétations I vérifiant $T_{\lceil P \rceil^+}(I) \subseteq I$. $\mathcal{M}_P^{\mathcal{C}^+}$ correspond donc à l'intersection des parties $T_{\lceil P \rceil^+}$-closes, c'est à dire à $\mathsf{Ind}(T_{\lceil P \rceil^+})$, et le lemme 4.20 permet de conclure. ◀

Lemme 4.22 $\mathcal{M}_P^{\mathcal{C}^+} \subseteq \mathcal{M}_P^{\mathcal{C}}$

PREUVE. Puisque $\lceil P \rceil^+ \subseteq \lceil P \rceil$, il vient :

$$\begin{array}{ll}
 & \forall I \quad T_{\lceil P \rceil^+}(I) \subseteq T_{\lceil P \rceil}(I) \\
\Rightarrow & \forall n \quad T_{\lceil P \rceil^+}^{\uparrow n} \subseteq T_{\lceil P \rceil}^{\uparrow n} \\
\Rightarrow & T_{\lceil P \rceil^+}^{\uparrow \omega} \subseteq T_{\lceil P \rceil}^{\uparrow \omega} \\
\Rightarrow & \mathcal{M}_P^{\mathcal{C}^+} \subseteq \mathcal{M}_P^{\mathcal{C}} \qquad\qquad \text{(théorème 4.13)}
\end{array}$$

◀

Par contre, la réciproque de ce lemme n'est pas vérifiée : $\mathcal{M}_P^{\mathcal{C}}$ n'est pas un sous-ensemble de $\mathcal{M}_P^{\mathcal{C}^+}$. Par exemple, avec le programme défini en (4.20), on a $\mathcal{M}_P^{\mathcal{C}^+} = \lceil p(f(x)) \rceil$ et $\mathcal{M}_P^{\mathcal{C}} = \lceil p(x) \rceil$.

Sémantique opérationnelle

Dérivations finies La $\mathcal{C}^+$-sémantique peut être vue comme un cas particulier de la $\mathcal{C}$-sémantique. On a donc les résultats classiques de validité et de complétude pour les SLD-réfutations directes.

Théorème 4.14 ($\mathcal{C}^+$-validité) *Si il existe une SLD-réfutation directe à partir d'une requête $A_1, \cdots, A_q$, alors $\{A_1, \cdots, A_q\} \subseteq \mathcal{M}_P^{\mathcal{C}^+}$.*

PREUVE. Induction (droite) sur la dérivation :

$$A_1, \cdots, A_q \overset{C_1,\theta_1}{\rightarrow}_P R_1 \rightarrow_P \cdots \rightarrow_P R_{i-1} \overset{C_i,\theta_i}{\rightarrow}_P R_i \rightarrow_P \cdots \rightarrow_P \Box$$

• Si la réfutation est un transition, alors $q = 1$ et C_1 est une clause unité $A \leftarrow$. Par hypothèse on a $A_1 = \theta_1 A$ et puisque $dom(\theta_1) \subseteq var(C_1^+)$ on peut conclure car $\mathcal{M}_P^{\mathcal{C}^+}$ est un $\mathcal{C}^+$-modèle de P (et donc de C_1).
• Si la réfutation s'écrit :

$$A_1, \cdots, A_q \overset{C_1,\theta_1}{\rightarrow}_P R_1 \overset{*,\sigma}{\rightarrow}_P \Box$$

et si k $(1 \leqslant k \leqslant q)$ est la position de l'atome sélectionné lors de la première transition, alors R_1 peut s'écrire $\theta_1(A_1, \cdots, A_{k-1}, C_1^-, A_{k+1}, \cdots, A_q)$. R_1 peut aussi s'écrire $A_1, \cdots, A_{k-1}, \theta_1 C_1^-, A_{k+1}, \cdots, A_q$, puisque $dom(\theta_1) \subseteq var(C_1^+)$. Par hypothèse d'induction $R_1 \subseteq \mathcal{M}_P^{\mathcal{C}^+}$ et il suffit alors de montrer que $A_k \in \mathcal{M}_P^{\mathcal{C}^+}$. C'est bien le cas puisque $\mathcal{M}_P^{\mathcal{C}^+}$ est un $\mathcal{C}^+$-modèle de

P (et donc de C_1) et l'hypothèse d'induction permet de conclure puisque $\theta_1 C_1^+ = A_k$. ◀

Théorème 4.15 ($\mathcal{C}^+$-complétude) *Si* $\{A_1, \cdots, A_q\} \subseteq \mathcal{M}_P^{\mathcal{C}^+}$*, alors il existe une SLD-réfutation directe à partir de la requête* $A_1, \cdots, A_q$*.*

Preuve. On montre tout d'abord le théorème pour $q = 1$. D'après le lemme 4.13, $A_1 \in T_{\lceil P \rceil^+}^{\uparrow\omega}$ et il existe un entier k tel que $A_1 \in T_{\lceil P \rceil^+}^{\uparrow k}$. Montrons, par induction sur k, que pour tout k, si $A \in T_{\lceil P \rceil^+}^{\uparrow k}$, alors il existe une SLD-réfutation directe à partir de A.

• Si $k = 0$, alors on peut conclure puisque $A \in T_{\lceil P \rceil^+}^{\uparrow 0} = \varnothing$ induit une contradiction.

• Si $k = m + 1$, alors $A \in T_{\lceil P \rceil^+}^{\uparrow k} = T_{\lceil P \rceil^+}(T_{\lceil P \rceil^+}^{\uparrow m})$ et il existe une clause C' s'écrivant $A' \leftarrow B'_1, \cdots, B'_r$ et une substitution θ, dont le domaine est inclus dans $var(A')$, vérifiant $\theta A' = A$ et $\{\theta B'_1, \cdots, \theta B'_r\} \subseteq T_{\lceil P \rceil^+}^{\uparrow m}$. De plus, d'après le lemme 4.13, il est possible de supposer que les variables de C' sont disjointes des variables de A. D'après le lemme 4.12, θ est donc un unificateur principal de A et A' et on peut construire la transition :

$$A \overset{C',\theta}{\to}_P \theta B'_1, \cdots, \theta B'_r$$

D'autre part, puisque $\{\theta B'_1, \cdots, \theta B'_r\} \subseteq T_{\lceil P \rceil^+}^{\uparrow m}$, par hypothèse d'induction, il existe r SLD-réfutations directes :

$$\theta B'_1 \overset{*}{\to}_P \Box \quad , \cdots , \quad \theta B'_r \overset{*}{\to}_P \Box$$

En «renommant» ces réfutations, on peut obtenir r SLD-réfutations directes :

$$\begin{array}{ll} d'_1 & \theta B'_1 \overset{*}{\to}_P \Box \\ \vdots & \\ d'_r & \theta B'_r \overset{*}{\to}_P \Box \end{array}$$

telles que :

$$\forall i \quad (1 \leqslant i \leqslant r) \quad \vartheta(d'_i) \cap \left(var(A) \cup var(C') \cup \bigcup_{1 \leqslant j < i} \vartheta(d'_j) \right) = \varnothing$$

Toutes les conditions sont à présent réunies pour pouvoir construire la SLD-réfutation directe :

$$A \overset{C',\theta}{\to}_P \theta B'_1, \cdots, \theta B'_r \overset{*}{\to}_P \Box$$

En reprenant le raisonnement mené pour combiner plusieurs SLD-réfutations directes, on montre le théorème pour q quelconque. ◀

Les atomes appartenant à $\mathcal{M}_P^{\mathcal{C}} \backslash \mathcal{M}_P^{\mathcal{C}^+}$ sont donc les atomes n'admettant pas de SLD-réfutation directe. Par exemple, pour le programme défini en (4.20), $p(z)$ appartient à $\mathcal{M}_P^{\mathcal{C}}$ mais n'appartient pas à $\mathcal{M}_P^{\mathcal{C}^+}$ ce qui correspond au fait que même s'il existe une réfutation :

$$p(z) \overset{\left[\begin{array}{c} x_1 \\ z \end{array}\right]}{\to_P} p(y_1) \overset{\left[\begin{array}{c} y_1 \\ f(w_1) \end{array}\right]}{\to_P} \square$$

il n'existe pas de réfutation directe à partir de $p(z)$. Toutefois, si toutes les clauses de P sont telles que chaque variable apparaissant dans le corps d'une clause apparaît aussi dans sa tête, alors on a $\mathcal{M}_P^{\mathcal{C}} = \mathcal{M}_P^{\mathcal{C}^+}$, puisque dans ce cas $\lceil P \rceil^+ = \lceil P \rceil$. De tels programmes vérifient de plus l'égalité $\mathsf{gfp}(T_{\lceil P \rceil^+}) = T_{\lceil P \rceil^+}^{\downarrow\omega}$.

Dérivations quelconques Nous montrons à présent qu'il existe une sémantique valide et complète pour les SLD-preuves directes infinies. La validité s'obtient directement : il est aisé d'exhiber un ensemble $T_{\lceil P \rceil^+}$-dense à partir d'une dérivation infinie.

Théorème 4.16 ($\mathcal{C}^+$-validité) *Soit P un programme et A_0 un atome. Si il existe une SLD-preuve directe :*

$$A_0 \overset{C_1,\theta_1}{\to}_P R_1 \to_P \cdots \to_P R_{i-1} \overset{C_i,\theta_i}{\to}_P R_i \to_P \cdots$$

alors $A_0 \in \mathsf{gfp}(T_{\lceil P \rceil^+})$.

Preuve. D'après la définition 1.2, il suffit de montrer qu'il existe un ensemble $T_{\lceil P \rceil^+}$-dense contenant A_0. Montrons que l'ensemble :

$$\bigcup_{i \geq 1} \theta_i C_i^+$$

satisfait ces deux propriétés.

1. Par définition, $A_0 = \theta_1 A_0 = \theta_1 C_1^+ \subseteq \cup_{i\geq 1} \theta_i C_i^+$.
2. Montrons que $\cup_{i\geq 1}\theta_i C_i^+ \subseteq T_{\lceil P \rceil^+}\left(\cup_{i\geq 1}\theta_i C_i^+\right)$. Si $A \in \cup_{i\geq 1}\theta_i C_i^+$, alors il existe un entier $k \geq 1$ tel que $A = \theta_k C_k^+$ et, puisque $dom(\theta_k) \subseteq var(C_k^+)$ et donc $\theta_k C_k \in \lceil P \rceil^+$, il suffit de montrer que tous les atomes de $\theta_k C_k^-$ apparaissent dans $\cup_{i\geq 1}\theta_i C_i^+$. Si $A_k \in \theta_k C_k^-$, alors $A_k \in R_k$ et puisque la dérivation est équitable, il existe une étape de résolution où le résidu de A_k est sélectionné :

$$\cdots \overset{C_k,\theta_k}{\to}_P R_k \to \cdots \to R_m \overset{C_{m+1},\theta_{m+1}}{\to}_P R_{m+1} \to \cdots$$

Pour $m \geq k$, on a donc $\theta_{m+1} \cdots \theta_{k+1} A_k = \theta_{m+1} C_{m+1}^+$. Puisque chaque clause C_i ne partage pas de variables avec les clauses C_j $(j < i)$ et puisque, par hypothèse, à chaque étape de résolution j, l'unificateur θ_j utilisé vérifie $dom(\theta_j) \subseteq var(C_j^+)$, il vient $\theta_{m+1} \cdots \theta_{k+1} A_k = A_k = \theta_{m+1} C_{m+1}^+ \subseteq \cup_{i \geq 1} \theta_i C_i^+$ ce qui permet de conclure.

◀

Lorsque la dérivation infinie n'est pas équitable, il est possible de caractériser un ensemble d'hypothèses suffisantes pour que l'atome «partiellement prouvé» par la dérivation appartienne à $\mathsf{CoInd}(\lceil P \rceil^+)$. Pour ce faire, nous définissons la requête résiduelle d'une dérivation :

$$R_0 \overset{C_1,\theta_1}{\rightarrow}_P R_1 \rightarrow_P \cdots \rightarrow_P R_{i-1} \overset{C_i,\theta_i}{\rightarrow}_P R_i \rightarrow_P \cdots$$

par :

$$R_\infty = \bigcup_{p \geq 0} \bigcap_{p \leqslant n} R_n$$

Signalons que même si la dérivation infinie est équitable, R_∞ n'est pas nécessairement l'ensemble vide (par exemple, pour la dérivation équitable (4.3), on a $R_\infty = \{p(z)\}$). Le théorème 4.16 ne doit donc pas être vu comme un corollaire du théorème suivant mais comme une autre façon d'interpréter une dérivation infinie qui ne construit pas de termes infinis : ces dérivations sont vues ici comme des preuves partielles, et expriment donc une implication. Ce théorème peut être comparé au lemme 3.13, établissant l'implication $P \models R \Rightarrow P \models \theta R_0$ pour une dérivation finie $R_0 \overset{\theta,*}{\rightarrow}_P R$.

Théorème 4.17 *Soit P un programme défini et A_0 un atome. Si il existe une dérivation infinie «directe» :*

$$R_0 = A_0 \overset{C_1,\theta_1}{\rightarrow}_P R_1 \rightarrow_P \cdots \rightarrow_P R_{i-1} \overset{C_i,\theta_i}{\rightarrow}_P R_i \rightarrow_P \cdots$$

alors $R_\infty \subseteq \mathsf{gfp}(T_{\lceil P \rceil^+}) \Rightarrow A_0 \in \mathsf{gfp}(T_{\lceil P \rceil^+})$.

Preuve. Supposons $R_\infty \subseteq \mathsf{gfp}(T_{\lceil P \rceil^+})$ et montrons qu'il existe un arbre de preuve de A_0 pour $\lceil P \rceil^+$. Pour cela, on définit la suite $T_1, \cdots, T_i, \cdots$ d'arbres de preuve partiels où chaque T_i est un arbre de preuve partiel de A_0 pour $\lceil P \rceil^+$ tel que tous les atomes de R_i sont des feuilles de T_i.

- L'arbre T_1 est construit à partir de la première transition : sa racine est $A_0 = \theta_1 A_0$ qui admet tous les atomes de $\theta_1 C_1^-$ pour fils ; tous les fils de A_0 sont des feuilles. Il s'agit bien d'un arbre de preuve pour $\lceil P \rceil^+$ puisque $\theta_1 C_1 \in \lceil P \rceil^+$ et $A_0 = \theta_1 C_1^+$. D'autre part, les atomes de $R_1 = \theta_1 C_1^-$ sont bien des feuilles de T_1.
- Si l'on dispose d'un arbre de preuve partiel T_{n-1} de A_0 (correspondant aux $n-1$ premières transtions) pour $\lceil P \rceil^+$ tel que les atomes de R_{n-1} sont des feuilles de T_{n-1}, alors si A est l'atome sélectionné dans R_{n-1}

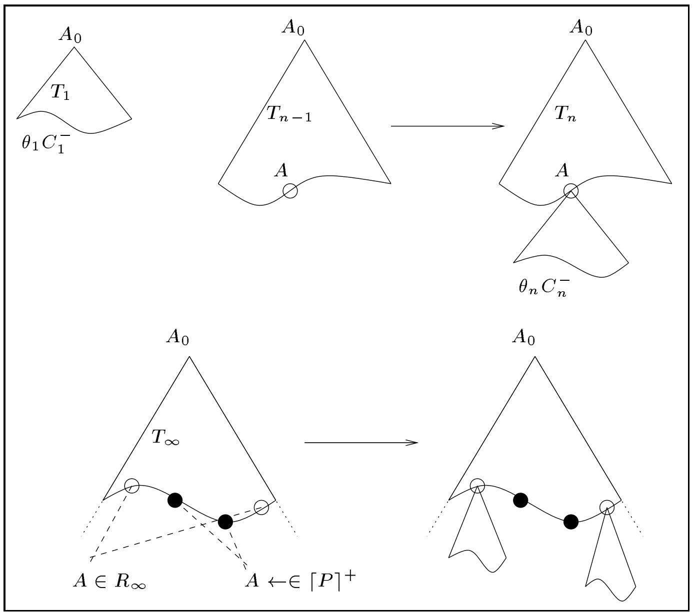


FIG. 4.4 – *Preuve du théorème 4.17*

à la position k, c'est que A est une feuille de T_{n-1} et puisque $A = \theta_n A$, il suffit d'insérer tous les atomes de $\theta_n C_n^-$ comme fils de A (les noeuds insérés sont des feuilles) pour obtenir T_n qui vérifie bien les propriétés énoncées puisque $R_n = \theta_n R_{n-1}[k \leftarrow C_n^-] = R_{n-1}[k \leftarrow \theta_n C_n^-]$.

En itérant le processus de construction des T_i on obtient un arbre de preuve partiel T_∞ de A_0 pour $\lceil P \rceil^+$ dont les feuilles correspondent soit à une instance de clause unité de P utilisée dans la dérivation, soit à un atome de R_∞ puisque les atomes de R_∞ ne sont jamais sélectionnés et ne sont pas affectés par les unificateurs utilisés. Par hypothèse, chaque atome de R_∞ est racine d'un arbre de preuve pour $\lceil P \rceil^+$ et on obtient alors un arbre de preuve de A_0 en substituant aux feuilles de T_∞ correspondant aux atomes de R_∞ les arbres de preuve associés. Les constructions effectuées dans cette preuve sont illustrées sur la figure 4.4. ◀

Le théorème de complétude s'obtient à l'aide des deux lemmes qui suivent.

Lemme 4.23 *Soit P un programme défini et A_0 un atome. Etant donné un ensemble, éventuellement infini, $\{C_1, \cdots, C_i, \cdots\} \subseteq \lceil P \rceil^+$ satisfaisant :*

$$\forall i > 0 \quad var(C_i) \cap \left(var(A_0) \cup \bigcup_{1 \leqslant j < i} var(C_j) \right) = \varnothing$$

il existe un ensemble $\{C_{P,1}, \cdots, C_{P,i}, \cdots\}$ de variantes de clauses de P satis-

faisant :

$$\forall i > 0 \quad var(C_{P,i}) \cap \left(var(A_0) \cup \bigcup_{1 \leqslant j < i} var(C_{P,j}) \right) = \varnothing$$

et tel que chaque clause $C_{P,i}$ vérifie $C_i = \mu_i C_{P,i}$ où μ_i est une substitution idempotente vérifiant $dom(\mu_i) = var(C_{P,i}^{+})$.

PREUVE. Pour tout i, puisque $C_i \in \lceil P \rceil^{+}$, pour une substitution σ_i et pour une clause $C_i^P \in P$, on a $C_i = \sigma_i C_i^P$ avec $dom(\sigma_i) \subseteq var(C_i^{P\,+})$:

$$\sigma_i = \begin{bmatrix} x_1 & \cdots & x_k \\ t_1 & \cdots & t_k \end{bmatrix}$$

Soit $\{y_1, \cdots, y_q\} = var(C_i^{P\,+}) \backslash dom(\sigma_i)$. Afin de définir $C_{P,i} = r_i C_i^P$, qui est bien une variante d'une clause de P, on introduit une substitution de renommage r_i idempotente et injective sur son domaine :

$$r_i = \begin{bmatrix} x_1 & \cdots & x_k & y_1 & \cdots & y_q \\ w_1 & \cdots & w_k & z_1 & \cdots & z_q \end{bmatrix}$$

telle que :

$$range(r_i) \cap \left(var(A_0) \cup \bigcup_{1 \leqslant j < i} var(C_{P,j}) \cup \bigcup_{j \geq 1} var(C_j) \cup var(C_i^P) \right) = \varnothing$$

Montrons que :

$$\forall i > 0 \quad var(C_{P,i}) \cap \left(var(A_0) \cup \bigcup_{1 \leqslant j < i} var(C_{P,j}) \right) = \varnothing \tag{4.21}$$

Tout d'abord, on montre que :

$$\forall i > 0 \quad var(C_{P,i}) \subseteq (range(r_i) \cup var(C_i))$$

En effet, soit $v \in var(C_{P,i})$. Puisque $C_{P,i} = r_i C_i^P$, deux cas se présentent :

1. Soit $v \in range(r_i)$ et on peut conclure.
2. Soit $v \in var(C_i^P)$ et dans ce cas :
   (a) Soit $v \in var(C_i^{P\,+})$ et puisque $dom(r_i) = var(C_i^{P\,+})$, on aboutit à une contradiction car r_i est idempotente.
   (b) Soit $v \in var(C_i^{P\,-}) \backslash var(C_i^{P\,+})$ et alors puisque $C_i = \sigma_i C_i^P$ et $dom(\sigma_i) = var(C_i^{P\,+})$, il vient $v \in C_i$ ce qui permet de conclure.

D'après la définition de r_i, les variables apparaissant dans $C_{P,i}$ issues de $range(r_i)$ satisfont bien la propriété (4.21) et puisque $var(C_i) \cap var(A_0) = \varnothing$, il reste à montrer que :

$$var(C_i) \cap \bigcup_{1 \leqslant j < i} var(C_{P,j}) = \varnothing$$

ce qui est vérifié si :

$$var(C_i) \cap \bigcup_{1 \leqslant j < i} (range(r_j) \cup var(C_j)) = \varnothing$$

Or cette propriété est bien satisfaite puisque :

$$\forall j > 0 \quad range(r_j) \cap \bigcup_{k \geq 1} var(C_k) = \varnothing \quad \text{et} \quad var(C_i) \cap \bigcup_{1 \leqslant j < i} var(C_j) = \varnothing$$

Enfin, il reste à montrer qu'il existe une substitution idempotente μ_i telle que $dom(\mu_i) = var(C^+_{P,i})$ et vérifiant $C_i = \mu_i C_{P,i}$. Cette substitution peut être définie par :

$$\mu_i = \begin{bmatrix} w_1 & \cdots & w_k & z_1 & \cdots & z_q \\ t_1 & \cdots & t_k & y_1 & \cdots & y_q \end{bmatrix}$$

En effet, montrons que $C_i = \sigma_i C^P_i = \mu_i r_i C^P_i = \mu_i C_{P,i}$. Soit $v \in var(C^P_i)$. Deux cas se présentent :

1. Soit $v \in var(C_i^{P\,+})$ et dans ce cas :
   (a) Soit $v = y_j$ $(1 \leqslant j \leqslant q)$ et on peut conclure puisque $\mu_i r_i y_j = \mu_i z_j = y_j = \sigma_i y_j$.
   (b) Soit $v = x_j$ $(1 \leqslant j \leqslant k)$ ce qui permet aussi de conclure puisque $\mu_i r_i x_j = \mu_i w_j = t_j = \sigma_i x_j$.
2. Soit $v \in var(C_i^{P\,-}) \backslash var(C_i^{P\,+})$ et il vient $v \notin dom(\sigma_i)$ et $v \notin dom(r_i)$ et on peut conclure puisque, par définition de r_i, on a $v \notin dom(\mu_i) = range(r_i)$.

Montrons que μ_i est idempotente, c'est à dire que $dom(\mu_i) \cap range(\mu_i) = \varnothing$. Tout d'abord, puisque r_i est idempotente, et puisque $dom(\mu_i) = range(r_i)$, on a bien $dom(\mu_i) \cap \{y_1, \cdots, y_q\} = \varnothing$. D'autre part, puisque $C_i = \sigma_i C^P_i$, on sait que :

$$\bigcup_{1 \leqslant j \leqslant k} var(t_j) \subseteq var(C_i)$$

et on peut alors conclure puisque $range(r_i) \cap var(C_i) = \varnothing$. ◀

Lemme 4.24 *Si il existe une transition* $R_0 \overset{\mu C, \theta}{\to} R_1$ *et si :*

$$dom(\theta) = var(\mu C^+) \qquad var(C) \cap var(R_0) = \varnothing \qquad dom(\mu) = var(C^+)$$

alors il existe une transition $R_0 \overset{C, \sigma}{\to} R_1$ *telle que* $dom(\sigma) = var(C^+)$.

PREUVE. Soit A l'atome sélectionné dans R_0 à la position k. Puisque, par hypothèse, $dom(\theta) = var(\mu C^+)$, on a $A = \theta A = \theta\mu C^+$. D'autre part, puisque $var(C) \cap var(R_0) = \varnothing$, d'après le lemme 4.12, la restriction σ de $\theta\mu$ aux variables de C^+ est un unificateur principal de A et C^+ et on peut alors construire la transition :

$$R_0 \stackrel{C,\sigma}{\rightarrow} R'_1$$

Montrons que $R'_1 = R_1$. Puisque :

$$\begin{array}{rrll} & R_1 = & \theta R_0[k \leftarrow \mu C^-] & = R_0[k \leftarrow \theta\mu C^-] \\ \text{et} & R'_1 = & \sigma R_0[k \leftarrow C^-] & = R_0[k \leftarrow \sigma C^-] \end{array}$$

il suffit de montrer que $\theta\mu C^- = \sigma C^-$. Soit $v \in var(C^-)$. Deux cas se présentent :

1. Si $v \in var(C^+)$, alors puisque σ est la restriction de $\theta\mu$ aux variables de C^+, on a bien $\theta\mu v = \sigma v$.
2. Si $v \in var(C^-)\backslash var(C^+)$, alors, par hypothèse, $v \notin dom(\sigma)$ et $v \notin dom(\mu)$. On a donc $v \in var(\mu C^-)\backslash var(\mu C^+)$ et donc $v \notin dom(\theta)$ ce qui permet de conclure puisqu'il vient alors $\theta\mu v = \sigma v = v$.

◀

Nous sommes à présent en mesure de prouver le théorème de complétude suivant.

Théorème 4.18 ($\mathcal{C}^+$-complétude) *Soit P un programme défini et A un atome. Si $A \in \mathsf{gfp}(T_{\lceil P\rceil^+})$, alors il existe une SLD-preuve directe à partir de A avec le programme P.*

PREUVE. Si $A \in \mathsf{gfp}(T_{\lceil P\rceil^+})$, alors, d'après le théorème 1.3, $A \in \mathsf{CoInd}(\lceil P\rceil^+)$ et d'après le lemme 4.12, il existe une SLD-preuve (directe) à partir de A avec le programme $\lceil P\rceil^+$ telle qu'à chaque étape i, si θ_i est l'unificateur principal mis en jeu et C_i est la clause utilisée, θ_i est un renommage injectif sur son domaine tel que $dom(\theta_i) = var(C_i^+)$:

$$A \stackrel{C_1,\theta_1}{\rightarrow}_{\lceil P\rceil^+} R_1 \rightarrow_{\lceil P\rceil^+} \cdots \rightarrow_{\lceil P\rceil^+} R_{i-1} \stackrel{C_i,\theta_i}{\rightarrow}_{\lceil P\rceil^+} R_i \rightarrow_{\lceil P\rceil^+} \cdots$$

Les conditions de renommage des clauses de cette dérivation permettent d'appliquer le lemme 4.23 pour établir l'existence d'un ensemble de variantes de clauses de P, $\{C_{P,1}, \cdots, C_{P,i}, \cdots\}$, satisfaisant :

$$\forall i > 0 \quad var(C_{P,i}) \cap \left(var(A_0) \cup \bigcup_{1 \leqslant j < i} var(C_{P,j}) \right) = \varnothing$$

et tel que chaque clause $C_{P,i}$ vérifie $C_i = \mu_i C_{P,i}$ où μ_i est une substitution idempotente vérifiant $dom(\mu_i) = var(C_{P,i}^+)$. Le lemme 4.24 permet alors d'établir l'existence d'une SLD-preuve directe à partir de A avec le programme P telle qu'à chaque étape i, la clause utilisée est $C_{P,i}$ et telle que l'unificateur principal mis en jeu σ_i vérifie $dom(\sigma_i) = var(C^+_{P,i})$. ◀

4.3.2 SLD-preuves infinies sur un domaine fini

Les restrictions imposées pour définir la classe de dérivations infinies étudiée dans le paragraphe précédent sont trop contraignantes. Elles ne permettent pas de prendre en compte des dérivations qui «calculent» un terme fini et «prouvent» une propriété sur ce terme. Nous nous intéressons donc dans ce paragraphe aux dérivations infinies qui ne «calculent» pas de termes infinis.

Définition 4.17 *Une SLD-preuve sur un domaine fini est soit une réfutation, soit une dérivation infinie équitable :*

$$R_0 \overset{C_1,\theta_1}{\to_P} R_1 \to_P \cdots \to_P R_{i-1} \overset{C_i,\theta_i}{\to_P} R_i \to_P \cdots$$

telle que :

$$\forall k \geq 0 \quad \exists p > k \quad \forall q \geq p \quad \theta_q \cdots \theta_p \cdots \theta_{k+1} R_k \approx \theta_p \cdots \theta_{k+1} R_k$$

En particulier, toute SLD-réfutation directe est une SLD-preuve sur un univers fini.

Il ne suffit donc pas que l'atome «calculé à l'infini» par la dérivation soit fini : aucun terme infini ne doit être construit par la dérivation. Par exemple, avec le programme :

$$P = \{q(x) \leftarrow p(x) \ ; \ p(f(x)) \leftarrow p(x)\}$$

on peut obtenir la dérivation équitable suivante :

$$q(z) \overset{\begin{bmatrix} x \\ z \end{bmatrix}}{\to_P} p(z) \overset{\begin{bmatrix} z \\ f(x_1) \end{bmatrix}}{\to_P} p(x_1) \to_P \cdots \overset{\begin{bmatrix} x_{i-1} \\ f(x_i) \end{bmatrix}}{\to_P} p(x_i) \to_P \cdots$$

Même si aucun des unificateurs utilisés n'affecte la requête initiale $q(z)$ (et donc l'atome «calculé» par la dérivation est fini), cette dérivation ne constitue pas une SLD-preuve sur un domaine fini puisqu'elle «calcule à l'infini» l'atome $p(f^\omega)$. Nous utiliserons par la suite une définition équivalente pour les dérivations sur un domaine fini. Cette définition est mise en relation avec la définition 4.17 par le lemme suivant.

Lemme 4.25 *La dérivation infinie équitable :*

$$R_0 \overset{C_1,\theta_1}{\to_P} R_1 \to_P \cdots \to_P R_{i-1} \overset{C_i,\theta_i}{\to_P} R_i \to_P \cdots$$

est une SLD-preuve infinie sur un domaine fini si et seulement si

$$\forall i \geq 0 \quad \exists R \quad \forall n \geq i+1 \quad \theta_n \theta_{n-1} \cdots \theta_{i+1} R_i \leqslant R$$

où R est une requête (i.e. R ne contient que des atomes finis).

PREUVE. ($\Rightarrow$). Par définition :

$$\forall k \geq 0 \; \exists p > k \; \forall q \geq p \quad \theta_q \cdots \theta_p \cdots \theta_{k+1} R_k \approx \theta_p \cdots \theta_{k+1} R_k$$

et il vient $\forall k \geq 0 \quad \forall q \geq k+1 \quad \theta_q \cdots \theta_{k+1} R_k \leqslant \theta_p \cdots \theta_{k+1} R_k$.
($\Leftarrow$). Si $k \geq 0$ et si il existe une requête R de longueur ℓ telle que :

$$\forall n \geq k+1 \quad \theta_n \theta_{n-1} \cdots \theta_{k+1} R_k \leqslant R$$

alors, pour chaque atome $A_i \in R_k$ $(1 \leqslant i \leqslant \ell)$, il existe un atome $B_i \in R$ tel que $\forall n \geq k+1$, $\theta_n \theta_{n-1} \cdots \theta_{k+1} A_i \leqslant B_i$. Il est clair que :

$$A_1 \leqslant A_2 \Rightarrow |\mathcal{O}(A_1)| \leqslant |\mathcal{O}(A_2)|$$

où $|E|$ dénote le cardinal de l'ensemble E, et $\mathcal{O}(A)$ désigne l'ensemble des occurrences de l'arbre A introduit dans la définition 4.2. Aussi, la suite $(|\mathcal{O}(\theta_n \cdots \theta_{k+1} A_i)|)_{n \geq k+1}$ est croissante et majorée par $|\mathcal{O}(B_i)|$. Cette suite est donc convergente et il existe alors $p'_i \geq k+1$ tel que :

$$\forall q'_i \geq p'_i \quad |\mathcal{O}(\theta_{q'_i} \cdots \theta_{k+1} A_i)| = |\mathcal{O}(\theta_{p'_i} \cdots \theta_{k+1} A_i)|$$

A présent, en remarquant que :

$$(A_1 \leqslant A_2 \wedge |\mathcal{O}(A_1)| = |\mathcal{O}(A_2)|) \Rightarrow |var(A_1)| \geq |var(A_2)|$$

on sait que la suite $(|var(\theta_n \cdots \theta_{k+1} A_i)|)_{n \geq p'_i}$ est décroissante et minorée par 0. Cette suite est donc convergente et il existe alors $p_i \geq p'_i$ tel que :

$$\forall q_i \geq p_i \quad |var(\theta_{q_i} \cdots \theta_{k+1} A_i)| = |var(\theta_{p_i} \cdots \theta_{k+1} A_i)|$$

Enfin, puisque chaque unficateur θ_j $(j \geq k+1)$ est, par définition, idempotent, il vient :

$$\forall q_i \geq p_i \quad \theta_{q_i} \cdots \theta_{k+1} A_i \approx \theta_{p_i} \cdots \theta_{k+1} A_i$$

En effet, si θ est une substitution idempotente telle que $\theta A_1 = A_2$ et si $|var(A_1)| = |var(A_2)|$ et $|\mathcal{O}(A_1)| = |\mathcal{O}(A_2)|$, alors $A_1 \approx A_2$. On peut alors conclure puisque $p = \max_{1 \leqslant i \leqslant \ell}(p_i)$ est tel que :

$$\forall q \geq p \quad \theta_q \cdots \theta_p \cdots \theta_{k+1} R_k \approx \theta_p \cdots \theta_{k+1} R_k$$

◀

Le théorème de validité est obtenu en associant à toute SLD-preuve sur un domaine fini un arbre de preuve.

Lemme 4.26 *Si il existe une SLD-preuve sur un domaine fini :*

$$A_0 \overset{C_1,\theta_1}{\rightarrow_P} R_1 \rightarrow_P \cdots \rightarrow_P R_{i-1} \overset{C_i,\theta_i}{\rightarrow_P} R_i \rightarrow_P \cdots$$

alors il existe $k \geq 0$ tel que $\theta_k \cdots \theta_1 A_0 \in \mathsf{gfp}(T_{\lceil P \rceil})$.

Preuve. D'après le théorème 1.3 et le lemme 4.11, il suffit de montrer qu'il existe un entier k tel que $\theta_k \cdots \theta_1 A_0$ soit racine d'un arbre de preuve pour $\lceil P \rceil$. Pour cela, on définit la suite $T_1, \cdots, T_i, \cdots$ d'arbres de preuve partiels où chaque T_i est un arbre de preuve partiel de $\theta_i \cdots \theta_1 A_0$ pour $\lceil P \rceil$ tel que tous les atomes de R_i sont des feuilles de T_i (voir figure 4.5).

- L'arbre T_1 est construit à partir de la première transition : sa racine est $\theta_1 A_0$ qui admet tous les atomes de $\theta_1 C_1^-$ pour fils ; tous les fils de $\theta_1 A_0$ sont des feuilles. Il s'agit bien d'un arbre de preuve pour $\lceil P \rceil$ puisque $\theta_1 C_1 \in \lceil P \rceil$ et $\theta_1 A_0 = \theta_1 C_1^+$. D'autre part, les atomes de $R_1 = \theta_1 C_1^-$ sont bien des feuilles de T_1.
- Si l'on dispose d'un arbre de preuve partiel T_{n-1} de $\theta_{n-1} \cdots \theta_1 A_0$ (correspondant aux $n-1$ premières transitions) pour $\lceil P \rceil$ tel que les atomes de R_{n-1} sont des feuilles de T_{n-1}, alors en appliquant la substitution θ_n à tous les noeuds de T_{n-1}, on obtient un arbre de preuve partiel de $\theta_n \cdots \theta_1 A_0$ pour $\lceil P \rceil$ tel que les atomes de $\theta_n R_{n-1}$ sont des feuilles. Si A est l'atome sélectionné dans R_{n-1}, alors A est une feuille de T_{n-1} et $\theta_n A$ est une feuille dans le nouvel arbre. Il suffit alors d'insérer tous les atomes de $\theta_n C_n^-$ comme fils de $\theta_n A$ (les noeuds insérés sont des feuilles) pour obtenir T_n qui vérifie bien les propriétés énoncées puisque $R_n = \theta_n R_{n-1}[k \leftarrow C_n^-]$.

En itérant le processus de construction des T_i, on obtient un arbre de preuve de $\theta_k \cdots \theta_1 A_0$ pour $\lceil P \rceil$. En effet, puisque la dérivation en hypothèse est une SLD-preuve sur un domaine fini, il existe un entier $k \geq 0$ tel que pour tout $q \geq k$, $\theta_q \cdots \theta_k \cdots \theta_1 A_0 \approx \theta_k \cdots \theta_1 A_0$ et de plus chaque noeud de l'arbre correspond à un atome fini. D'autre part, chaque feuille correspond à une clause unité de $\lceil P \rceil$ puisque la dérivation en hypothèse du lemme est équitable. ◀

Ce lemme se généralise facilement et on obtient le théorème de validité sous la forme suivante.

Théorème 4.19 (Validité) *Si il existe une SLD-preuve sur un domaine fini :*

$$A_1, \cdots, A_n \overset{C_1,\theta_1}{\rightarrow_P} R_1 \rightarrow_P \cdots \rightarrow_P R_{i-1} \overset{C_i,\theta_i}{\rightarrow_P} R_i \rightarrow_P \cdots$$

alors il existe $k \geq 0$ tel que pour tout i ($1 \leqslant i \leqslant n$) $\theta_k \cdots \theta_1 A_i \in \mathsf{gfp}(T_{\lceil P \rceil})$.

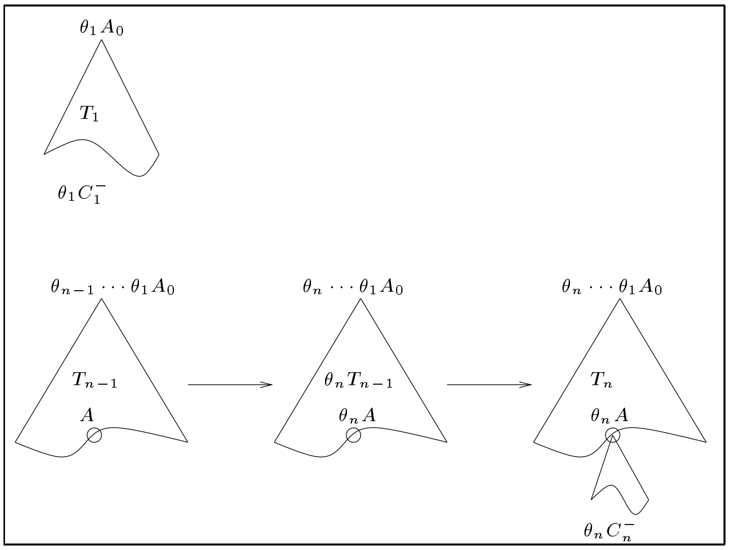


FIG. 4.5 – *Preuve du lemme 4.26*

PREUVE. La démonstration est similaire à celle du lemme 4.26 : la seule différence provient du fait qu'au lieu de construire une suite d'arbres partiels, on construit une suite de n-uplets d'arbres partiels pour $\lceil P \rceil$:

$$((T_1^1, \cdots, T_1^n), \cdots, (T_i^1, \cdots, T_i^n), \cdots)$$

tel que $\theta_i \cdots \theta_1 A_j$ est racine de l'arbre T_i^j $(1 \leqslant j \leqslant n)$ et tel que chaque atome présent dans la requête R_i est une feuille d'un des arbres T_i^j. ◀

Le théorème de validité de la $\mathcal{C}^+$-sémantique peut être vu comme un corollaire de ce théorème. Le théorème de complétude va s'obtenir à partir des trois lemmes qui suivent.

Lemme 4.27 *Soit P un programme défini et A_0 un atome. Etant donné un ensemble, éventuellement infini, $\{C_1, \cdots, C_i, \cdots\} \subseteq \lceil P \rceil$ satisfaisant :*

$$\forall i > 0 \quad var(C_i) \cap \left(var(A_0) \cup \bigcup_{1 \leqslant j < i} var(C_j) \right) = \varnothing$$

il existe un ensemble $\{C_{P,1}, \cdots, C_{P,i}, \cdots\}$ de variantes de clauses de P satisfaisant :

$$\forall i > 0 \quad var(C_{P,i}) \cap \left(var(A_0) \cup \bigcup_{1 \leqslant j < i} var(C_{P,j}) \cup \bigcup_{k \geq 0} var(C_k) \right) = \varnothing$$

et tel que chaque clause $C_{P,i}$ vérifie $C_i = \mu_i C_{P,i}$ où μ_i est une substitution idempotente vérifiant $dom(\mu_i) = var(C_{P,i})$.

Preuve. Pour tout i, puisque $C_i \in \lceil P \rceil$, pour une substitution σ_i et pour une clause $C_i^P \in P$, on a $C_i = \sigma_i C_i^P$ avec $dom(\sigma_i) \subseteq var(C_i^P)$:

$$\sigma_i = \begin{bmatrix} x_1 & \cdots & x_k \\ t_1 & \cdots & t_k \end{bmatrix}$$

Soit $\{y_1, \cdots, y_q\} = var(C_i^P) \backslash dom(\sigma_i)$. Afin de définir $C_{P,i} = r_i C_i^P$, qui est bien une variante d'une clause de P, on introduit une substitution de renommage r_i idempotente et injective sur son domaine :

$$r_i = \begin{bmatrix} x_1 & \cdots & x_k & y_1 & \cdots & y_q \\ w_1 & \cdots & w_k & z_1 & \cdots & z_q \end{bmatrix}$$

telle que :

$$range(r_i) \cap \left(var(A_0) \cup \bigcup_{1 \leqslant j < i} var(C_{P,j}) \cup \bigcup_{j \geq 1} var(C_j) \right) = \varnothing$$

Il reste à montrer qu'il existe une substitution idempotente μ_i telle que $dom(\mu_1) = var(C_{P,i})$ et $C_i = \mu_i C_{P,i}$. Cette substitution peut être définie par :

$$\mu_i = \begin{bmatrix} w_1 & \cdots & w_k & z_1 & \cdots & z_q \\ t_1 & \cdots & t_k & y_1 & \cdots & y_q \end{bmatrix}$$

En effet, montrons que $C_i = \sigma_i C_i^P = \mu_i r_i C_i^P = \mu_i C_{P,i}$. Si $v \in var(C_i^P)$, alors deux cas se présentent. Soit $v \notin dom(\sigma_i)$ et dans ce cas $v = y_j$ $(1 \leqslant j \leqslant q)$ ce qui permet de conclure puisque $\sigma_i y_j = y_j = \mu_i z_j = \mu_i r_i y_j$. Soit $v \in dom(\sigma_i)$ et dans ce cas $v = x_j$ $(1 \leqslant j \leqslant k)$ ce qui permet de conclure puisque $\sigma_i x_j = t_j = \mu_i w_j = \mu_i r_i x_j$. Montrons que μ_i est idempotente, c'est à dire que $dom(\mu_i) \cap range(\mu_i) = \varnothing$. Tout d'abord, puisque r_i est idempotente, et puisque $dom(\mu_i) = range(r_i)$, on a bien $dom(\mu_i) \cap \{y_1, \cdots, y_q\} = \varnothing$. D'autre part, puisque $C_i = \sigma_i C_i^P$, on sait que :

$$\bigcup_{1 \leqslant j \leqslant k} var(t_j) \subseteq var(C_i)$$

et on peut alors conclure puisque $range(r_i) \cap var(C_i) = \varnothing$. ◀

Lemme 4.28 *Soit A, A_0 et B trois atomes et θ une substitution telle que $\theta A = \theta B$ et $\theta A_0 = A_0$. Il existe un unificateur principal ρ de A et B tel que $\rho A_0 = A_0$.*

Preuve. Puisque $\theta A = \theta B$, il existe un unificateur principal σ de A et B tel que $\sigma \leqslant \theta$. Pour une substitution η, on a donc $\eta\sigma = \theta$. De plus, par hypothèse, $\theta A_0 = A_0$ et il vient $\eta\sigma A_0 = A_0$. Aussi, la restriction de σ aux variables de A_0 peut être vu comme une substitution de renommage :

$$\sigma = \begin{bmatrix} v_1 & \cdots & v_k & x_1 & \cdots & x_n \\ t_1 & \cdots & t_k & y_1 & \cdots & y_n \end{bmatrix}$$

où :

$$\{v_1, \cdots, v_k\} = dom(\sigma) \backslash var(A_0) \quad \text{et} \quad \{x_1, \cdots, x_k\} = dom(\sigma) \cap var(A_0)$$

On définit alors la substitution de renommage suivante, qui est idempotente et injective sur son domaine et qui correspond à une restriction de la substitution η :

$$r = \begin{bmatrix} y_1 & \cdots & y_n \\ x_1 & \cdots & x_n \end{bmatrix}$$

Montrons alors que $\rho = r\sigma$. Tout d'abord, il est clair que $\rho A_0 = A_0$. Il suffit donc de montrer que ρ est un unificateur principal de A et B. $\rho A = \rho B$ est immédiat (car $\sigma A = \sigma B$). Montrons que ρ est une substitution idempotente. Soit $v \in dom(\rho)$, deux cas se présentent :

1. Si $v \in dom(\sigma)$, alors $v = v_j$ $(1 \leqslant j \leqslant k)$ puisque $\rho x_i = x_i$ pour tout $i \in \{1, \cdots, n\}$. Aussi, $v \notin \{x_1, \cdots, x_n\}$ et $v \notin \cup_{1 \leqslant i \leqslant k} var(rt_i)$ puisque $\cup_{1 \leqslant i \leqslant k} var(rt_i) \subseteq \cup_{1 \leqslant i \leqslant k} var(t_i) \cup range(r)$ et σ est idempotente. On a donc $v \notin range(\rho)$.
2. Si $v \in dom(r)$, alors $v = y_j$ $(1 \leqslant j \leqslant n)$ et puisque $v \notin \{x_1, \cdots, x_n\}$ et $v \notin \cup_{1 \leqslant i \leqslant k} var(rt_i)$, on a $v \notin range(\rho)$.

Montrons à présent que ρ est un unificateur minimal. Soit μ une substitution telle que $\mu A = \mu B$. Puisque σ est un unificateur principal de A et$B$, on a $\sigma \leqslant \mu$ et il existe une substitution ν telle que $\nu\sigma = \mu$. Il vient alors $\rho \leqslant \mu$ puisque l'on peut montrer que $\nu r^{-1}\rho = \nu r^{-1} r\sigma = \nu\sigma = \mu$ où r^{-1} est la substitution «inverse» de r (i.e. $rr^{-1} = r^{-1}r = s_{id}$). Soit w une variable, deux cas se présentent :

1. Si $w \in dom(\sigma)$, alors :

   (a) si $w = v_j$ $(1 \leqslant j \leqslant k)$, alors il vient $\nu r^{-1} r\sigma v_j = \nu r^{-1} rt_j$ et on peut conclure puisque $r^{-1}rt_j = t_j$ car pour chaque variable $y \in var(t_j)$:

      i. soit $y \in dom(r)$ et $r^{-1}ry = y$ est immédiat
      ii. soit $y \notin dom(r)$ et on a $r^{-1}y = y$ puisque σ est idempotente et $var(t_j) \subseteq range(\sigma)$ et $dom(r^{-1}) \subseteq dom(\sigma)$

   (b) si $w = x_j$ $(1 \leqslant j \leqslant n)$, alors on peut aussi conclure puisque $\nu r^{-1} r\sigma x_j = \nu r^{-1} ry_j = \nu r^{-1} x_j = \nu y_j = \nu\sigma x_j$.

2. Si $w \notin dom(\sigma)$, alors montrons que $\nu r^{-1}rw = \nu w$. En effet, soit $w \in dom(r)$ et $r^{-1}rw = w$ ce qui permet de conclure, soit $w \notin dom(r)$ et il vient $r^{-1}w = w$ puisque $dom(r^{-1}) \subseteq dom(\sigma)$ et $w \notin dom(\sigma)$.

◀

Puisque d'après le theorème 4.12 il existe une SLD-preuve (directe) avec le programme $\lceil P \rceil$ à partir de chaque atome appartenant à $\mathsf{CoInd}(T_{\lceil P \rceil})$, le lemme qui suit permet de montrer comment on peut transformer une SLD-preuve (directe) avec le programme $\lceil P \rceil$ en une SLD-preuve sur un domaine fini avec le programme P. Ce lemme jouera pour le théorème de complétude que nous montrerons par la suite le même rôle que le lemme de généralisation (lemme 3.7) dans la preuve du théorème de complétude de la SLD-résolution.

Lemme 4.29 (Program lifting lemma) *Si il existe une SLD-preuve :*

$$A_0 \overset{C_1,\theta_1}{\to}_{\lceil P \rceil} R_1 \to_{\lceil P \rceil} \cdots \to_{\lceil P \rceil} R_{i-1} \overset{C_i,\theta_i}{\to}_{\lceil P \rceil} R_i \to_{\lceil P \rceil} \cdots$$

où pour tout $i \geq 1$, θ_i est une substitution de renommage idempotente et injective sur son domaine telle que $dom(\theta_i) = var(C_i^+)$, alors il existe une SLD-preuve sur un domaine fini :

$$A_0 \overset{C_{P,1},\sigma_1}{\to}_P R'_1 \to_P \cdots \to_P R'_{i-1} \overset{C_{P,i},\sigma_i}{\to}_P R'_i \to_P \cdots$$

telle que pour tout $i \geq 1$, $\sigma_i A_0 = A_0$, $C_i = \mu_i C_{P,i}$ et $R_i = \rho_i R'_i$ où ρ_i est la restriction de $\theta_i\mu_i\theta_{i-1}\mu_{i-1}\cdots\theta_1\mu_1$ aux variables de R'_i.

Preuve. Les conditions de renommage des clauses de la dérivation en hypothèse du lemme permettent d'appliquer le lemme 4.27 pour établir l'existence d'un ensemble de variantes $\{C_{P,1}, \cdots, C_{P,i}, \cdots\}$ de clauses de P satisfaisant :

$$\forall i > 0 \quad var(C_{P,i}) \cap \left(var(A_0) \cup \bigcup_{1 \leqslant j < i} var(C_{P,j}) \cup \bigcup_{j \geq 1} var(C_j) \right) = \varnothing$$

et tel que chaque clause $C_{P,i}$ vérifie $C_i = \mu_i C_{P,i}$ où μ_i est une substitution idempotente satisfaisant $dom(\mu_i) = var(C_{P,i})$.

• *Pour la première transition.* Par définition $\theta_1 A_0 = \theta_1 C_1^+ = \theta_1\mu_1 C_{P,1}^+$. De plus, puisque $dom(\mu_1) = var(C_{P,1})$ et $var(C_{P,1}) \cap var(A_0) = \varnothing$, on a $\mu_1 A_0 = A_0$ et il vient $\theta_1\mu_1 A_0 = \theta_1\mu_1 C_{P,1}^+$. De manière similaire, puisque $dom(\theta_1) = var(C_1^+)$ et $var(C_1) \cap var(A_0) = \varnothing$, on a $\theta_1 A_0 = A_0$. Aussi, on a $\theta_1\mu_1 A_0 = A_0 = \theta_1\mu_1 C_{P,1}^+$ et d'après le lemme 4.12, la restriction σ_1 de $\theta_1\mu_1$ aux variables de $C_{P,1}^+$ est un unificateur principal de A_0 et $C_{P,1}^+$ et on obtient la transition :

$$A_0 \overset{C_{P,1},\sigma_1}{\to}_P R'_1$$

Puisque $\theta_1\mu_1 A_0 = A_0$, on a bien $\sigma_1 A_0 = A_0$. Il reste à montrer que la restriction ρ_1 de $\theta_1\mu_1$ aux variables de R'_1 vérifie $\rho_1 R'_1 = R_1$. Montrons donc que $\rho_1 R'_1 = \rho_1\sigma_1 C^-_{P,1} = \theta_1\mu_1 C^-_{P,1} = \theta_1 C^-_1 = R_1$. Si $v \in var(C^-_{P,1})$, alors deux cas sont possibles. Si $v \in var(C^+_{P,1})$, alors $\sigma_1 v = \theta_1\mu_1 v$ et on peut conclure puisque $\theta_1\mu_1\theta_1\mu_1 v = \theta_1\mu_1 v$. Sinon, $v \notin var(C^+_{P,1})$, et on a $\rho_1\sigma_1 v = \rho_1 v = \theta_1\mu_1 v$ ce qui permet de conclure.

• *Pour la i-ème transition.* Montrons comment à partir de la transition :

$$R_{i-1} \overset{C_i,\theta_i}{\rightarrow}_{\lceil P \rceil} R_i$$

il est possible de construire une transition à partir de la requête R'_i vérifiant $\rho_{i-1} R'_{i-1} = R_{i-1}$ où ρ_{i-1} est la restriction de $\theta_{i-1}\mu_{i-1}\cdots\theta_1\mu_1$ aux variables de R'_{i-1} :

$$R'_{i-1} \overset{C_{P,i},\sigma_i}{\rightarrow}_{P} R'_i$$

telle que $\sigma_i A_0 = A_0$ et telle que la restriction ρ_i de $\theta_i\mu_i\theta_{i-1}\mu_{i-1}\cdots\theta_1\mu_1$ aux variables de R'_i vérifie $\rho_i R'_i = R_i$. Si A est l'atome sélectionné dans R_{i-1} à la position k, alors il existe un atome A' dans R'_{i-1} à la position k tel que $A = \rho_{i-1} A'$ et il vient $\theta_i\rho_{i-1}A' = \theta_i\mu_i C^+_{P,i}$. De plus, puisque :

$$\begin{array}{rl} & dom(\rho_{i-1}) \subseteq var(R'_{i-1}) \subseteq \left(\bigcup_{1 \leqslant j < i} var(C_{P,j}) \cup var(A_0) \right) \\ \text{et} & var(C_{P,i}) \cap \left(\bigcup_{1 \leqslant j < i} var(C_{P,j}) \cup var(A_0) \right) = \varnothing \end{array}$$

on a $\rho_{i-1} C_{P,i} = C_{P,i}$ et donc $\theta_i\rho_{i-1}A' = \theta_i\mu_i\rho_{i-1}C^+_{P,i}$. Puisque $dom(\mu_i) = var(C_{P,i})$ on a $\mu_i A = A$. Aussi, $\theta_i\mu_i\rho_{i-1}A' = \theta_i\mu_i\rho_{i-1}C^+_{P,i}$, et puisque $\theta_i\mu_i\cdots\theta_1\mu_1 A_0 = A_0$, d'après le lemme 4.28, il existe un unificateur principal σ_i de A' et $C^+_{P,i}$ tel que $\sigma_i A_0 = A_0$. Pour une substitution η_i, on a donc $\eta_i\sigma_i = \theta_i\mu_i\rho_{i-1}$. Il est alors possible de construire la transition :

$$R'_{i-1} \overset{C_{P,i},\sigma_i}{\rightarrow}_{P} R'_i$$

Pour établir le lien existant entre R_i et R'_i, on utilise l'égalité $\theta_i\mu_i\rho_{i-1}\sigma_i = \theta_i\mu_i\rho_{i-1}$ qui provient du fait que σ_i est idempotente et que par conséquent $\theta_i\mu_i\rho_{i-1}\sigma_i = \eta_i\sigma_i\sigma_i = \eta_i\sigma_i = \theta_i\mu_i\rho_{i-1}$. Nous pouvons alors prouver que $\theta_i\mu_i\rho_{i-1}R'_i = R_i$:

$$\begin{array}{rcll} \theta_i\mu_i\rho_{i-1}R'_i & = & \theta_i\mu_i\rho_{i-1}\sigma_i R'_{i-1}[k \leftarrow C^-_{P,i}] & \\ & = & \theta_i\mu_i\rho_{i-1}R'_{i-1}[k \leftarrow C^-_{P,i}] & (\theta_i\mu_i\rho_{i-1}\sigma_i = \theta_i\mu_i\rho_{i-1}) \\ & = & \theta_i\rho_{i-1}R'_{i-1}[k \leftarrow \mu_i C^-_{P,i}] & (\mu_i R_{i-1} = R_{i-1}) \\ & = & \theta_i(\rho_{i-1}R'_{i-1})[k \leftarrow \mu_i C^-_{P,i}] & (\rho_{i-1}C_i = C_i)) \\ & = & \theta_i R_{i-1}[k \leftarrow C_i] & \\ & = & R_i & \end{array}$$

La restriction ρ_i de $\theta_i\mu_i\rho_{i-1}$ aux variables de R'_i satisfait donc bien $\rho_i R'_i = R_i$. Pour finir, il faut montrer que la dérivation ainsi construite est bien une dérivation sur un domaine fini. Pour cela, on montre, par induction, que :

$$\forall n \geq i+1 \quad \rho_n\sigma_n\sigma_{n-1}\cdots\sigma_{i+1}R'_i = R_i$$

Pour $n = i+1$, on a bien :

$$\rho_{i+1}\sigma_{i+1}R'_i = \theta_{i+1}\mu_{i+1}\rho_i\sigma_{i+1}R'_i = \theta_{i+1}\mu_{i+1}\rho_i R'_i = \theta_{i+1}\mu_{i+1}R_i = R_i$$

Pour $n > i+1$, par hypothèse d'induction, on a $\rho_{n-1}\sigma_{n-1}\cdots\sigma_{i+1}R'_i = R_i$. Aussi, pour prouver $\rho_n\sigma_n\sigma_{n-1}\cdots\sigma_{i+1}R'_i = R_i$, il suffit de montrer que pour toute variable v apparaissant dans $\sigma_{n-1}\cdots\sigma_{i+1}R'_i$, on a $\rho_n\sigma_n v = \rho_{n-1}v$. Puisque $\rho_n\sigma_n = \theta_n\mu_n\rho_{n-1}\sigma_n = \theta_n\mu_n\rho_{n-1}$, il suffit de prouver que $\theta_n\mu_n\rho_{n-1}v = \rho_{n-1}v$. De plus, si $v \in \sigma_{n-1}\cdots\sigma_{i+1}R'_i$, alors on a :

$$v \in \left(var(R'_i) \cup \bigcup_{i+1\leqslant j\leqslant n-1} var(C_{P,j})\right) \subseteq \left(var(A_0) \cup \bigcup_{1\leqslant j\leqslant n-1} var(C_{P,j})\right)$$

Aussi, si $v \in dom(\rho_{n-1})$, alors $var(\rho_{n-1}v) \subseteq R_{n-1}$ et $\theta_n\mu_n\rho_{n-1}v = \rho_{n-1}v$ puisque $\theta_n\mu_n R_{n-1} = R_{n-1}$, sinon on peut aussi conclure puisque $\theta_n\mu_n v = v$. Par conséquent, puisque R_i ne contient que des atomes finis et puisque pour tout $n \geq i+1$, $\sigma_n\sigma_{n-1}\cdots\sigma_{i+1}R'_i \leqslant R_i$, d'après le lemme 4.25, la dérivation obtenue constitue une SLD-preuve sur un domaine fini. ◀

Le théorème de complétude s'obtient directement à partir de ce lemme et du théorème 4.12.

Théorème 4.20 (Complétude) *Soit P un programme et A un atome. Si $A \in \mathsf{gfp}(T_{\lceil P\rceil})$, alors il existe une SLD-preuve sur un domaine fini à partir de A avec le programme P :*

$$A \overset{C_{P,1},\sigma_1}{\to_P} R'_1 \to_P \cdots \to_P R'_{i-1} \overset{C_{P,i},\sigma_i}{\to_P} R'_i \to_P \cdots$$

telle que pour tout $i \geq 1$, $\sigma_i A = A$.

Preuve. Conséquence immédiate du théorème 1.3, du lemme 4.11, du théorème 4.12 et du lemme 4.29. ◀

Conclusion

Avant d'établir des propriétés formelles sur les programmes, il semble «naturel» de spécifier formellement leur sémantique opérationnelle. La SLD-résolution et les preuves des résultats classiques de cette relation, propriétés essentielles pour son utilisation (validité et complétude), ont donc été formalisées dans ce travail.

Dans un premier temps, nous avons cherché à formaliser l'unification des termes du premier ordre, mécanisme de base de la programmation logique. Pour ce faire, puisque qu'un développement similaire existait, nous avons présenté une méthode pour transposer des propriétés formelles. La démarche utilisée pour obtenir la preuve présentée diffère beaucoup de celle employée par J. Rouyer (proche de l'algorithme d'unification). Nous avons construit l'ensemble $T_\Sigma[X]$ des termes du premier ordre, et nous cherchions à résoudre le problème de l'unification sur cet ensemble. Ce problème étant résolu sur l'ensemble $Q_\Sigma[X]$ des quasi-termes, nous avons construit la bijection qui existait entre un sous-ensemble de $Q_\Sigma[X]$ et $T_\Sigma[X]$ et nous avons montré que les propriétés cherchées étaient préservées par cette bijection. Nous sommes donc arrivé au résultat attendu, sans nous soucier vraiment de l'unification, mais en établissant un lien étroit entre ces deux ensembles, évitant ainsi une preuve faisant appel à des théories plus complexes (substitutions, unifications, ... [28, 64, 77]). La principale difficulté de cette transposition fut de mettre en relation un ensemble (défini inductivement), les termes, avec un sous-ensemble caractérisé par un prédicat (sur un ensemble inductif) : les quasi-termes «compatibles». Nous avons vu deux méthodes pour construire ce lien (i.e. pour définir un procédé effectif de transformation d'un quasi-terme «compatible» en un terme). La première repose sur une technique courante en programmation : dans un cas absurde, c'est à dire qui ne se produira jamais, on peut «faire n'importe quoi» sans compromettre la correction de la construction. Cette approche utilise un axiome («*ex falso quodlibet sequitur*») et procède par induction sur la structure d'un quasi-terme. La seconde solution, et c'est celle que nous avons développée, a consisté à construire inductivement des ensembles de preuves, pour pouvoir raisonner par induction ou construire des fonctions récursives sur la structure de ces preuves. Nous avons ainsi pu effectuer de véritables calculs à partir du contenu algorithmique (explicite) de ces preuves, en procédant par induction, non plus sur la structure d'un quasi-terme, mais sur la structure de la preuve de sa «compatibilité», exploitant ainsi l'isomorphisme de Curry-Howard («*programmer = prouver*»). Cette «technique de preuve» illustre bien l'intérêt que pourrait présenter une bibliothèque de preuves, constituée de modules suffisament généraux, regroupant l'essentiel des notions de base utilisées en informatique, pour pouvoir être utilisés par diverses applications. En effet, si la présentation de l'approche développée dans le chapitre 2 ne semble ni plus simple, ni plus courte qu'une approche plus classique, le codage de la preuve du théorème d'unification sur les termes a nécessité seulement 75 lemmes en transposant le résultat obtenu par J. Rouyer dans [89] sur les quasi-termes qui compte environ 140 lemmes.

Dans des présentations plus informelles de la programmation logique, les problèmes liés aux renommages des variables sont souvent passés sous silence. Comme nous l'avons suggéré dans l'introduction, les preuves obtenues après formalisation diffèrent donc des preuves initiales que l'on peut trouver dans la littérature. Bien sûr, la principale différence provient du processus de renommage des clauses lors d'une (SLD-)dérivation : celui-ci est complètement explicité, ce qui introduit des difficultés dans les preuves. Un exemple typique est le théorème de complétude qui nécessite deux lemmes «techniques» (lemmes 3.18 et 3.19) sur les variables utilisées et dont les preuves ne sont pas si immédiates. Ces deux lemmes montrent bien à quel point certains détails peuvent se révéler importants. Par exemple, l'opération de «combinaison» de dérivations requiert plusieurs renommages, ces derniers devant être menés avec précaution, afin de garantir les hypothèses de séparation des variables dans la dérivation finale. Les deux principaux résultats dont les preuves sont «sensibles» à ces problèmes sont le lemme de généralisation et/ou le théorème de complétude. On trouvera dans [60, 93] une discussion intéressante sur ces «points de détail» mais peu d'articles sont consacrés aux propriétés des substitutions, objets de base de la programmation logique dont la manipulation s'est avérée délicate. Tout comme la β-réduction pour les λ-termes, la SLD-résolution définit une relation entre clauses. Certaines propriétés de ces deux relations peuvent être comparées. Alors que la β-réduction est confluente, la SLD-résolution satisfait seulement une propriété plus faible (propriété de commutation). Les propriétés de ce modèle sont souvent énoncées «à un renommage près», ce qui rappelle l'α-conversion. De même, l'explicitation des substitutions de renommage des variables lors d'une (SLD-)dérivation rappelle le $\lambda\sigma$-calcul (il s'agit d'un système de réécriture du premier ordre dans lequel on peut «encoder» le λ-calcul de manière à gérer pas à pas les substitutions, sans avoir à effectuer d'α-conversion ; ce λ-calcul dans lequel les substitutions sont manipulées de manière explicite fournit ainsi un lien entre le λ-calcul classique et ses implémentations concrètes) [1]. Les auteurs de l'article [1] n'hésitent pas à qualifier les substitutions d'«*éminence grise*» du λ-calcul ; cette remarque semble tout aussi pertinente concernant la SLD-résolution.

Signalons ici que la formalisation de théorèmes de complétude a été envisagée dans d'autres contextes. Dans [83], H. Persson présente une formalisation, à l'aide de l'assistant à la preuve ALF, d'une preuve constructive de la complétude de la logique intuitionniste du premier ordre. Dans [44], J. Harrison présente une discussion sur une formalisation dans HOL des résultats de base de la théorie des modèles (théorèmes de compacité et de Lowenheim-Skolem). Dans [62], J.L. Krivine décrit une formalisation dans une logique du second ordre d'une preuve intuitionniste de la complétude de la logique du premier ordre classique.

Dans le dernier chapitre de cette thèse, la sémantique des dérivations infinies a été étudiée. En programmation logique, il est courant de comprendre les dérivations comme des «preuves qui calculent» et c'est bien sûr l'aspect calculatoire de telles preuves qui permet l'utilisation de programmes définis dans la résolution de problèmes. C'est aussi cet aspect calculatoire qui est considéré dans la plupart des articles consacrés aux dérivations infinies : une dérivation infinie est alors vue comme processus de calcul à l'infini d'un objet infini. Toutefois, les dérivations sont avant tout des preuves. En interprétant le mécanisme d'exécution d'un programme logique par un processus de preuve, il a été possible de définir une sémantique valide et complète pour les dérivations infinies qui ne construisent pas de termes infinis. Cette approche est basée sur l'identification des SLD-preuves avec les preuves par co-induction et est justifiée parce que la complétude de la sémantique proposée ne pouvait être obtenue qu'en considérant un univers du discours ne comportant pas d'éléments infinis. En effet, en présence d'éléments infinis, seul un sous-ensemble du plus grand point fixe de l'opérateur associé à un programme pourrait caractériser les objets infinis calculables à l'infini à partir de ce programme. D'ailleurs, l'approche développée par W.G. Golson, présentée dans le paragraphe 4.1.3 (complétion par idéaux), permet de caractériser les objets «calculés» par une certaine classe de dérivations en termes d'objets minimaux présents dans le plus grand point fixe de l'opérateur T_P : tous les éléments de cet ensemble ne sont pas considérés. Cependant, de manière intuitive, c'est l'approche développée par G. Levi et C. Palamidessi, présentée dans le paragraphe 4.1.5 (fermeture topologique du plus petit point fixe d'un opérateur associé à une version modifiée du programme), qui semble le mieux capturer la notion d'atomes infinis calculables à l'infini : les dérivations infinies qui construisent de tels objets procèdent par approximations successives et la notion de limite d'une suite d'approximations finies semble plus adéquate pour définir la dénotation d'un programme. C'est du moins ce que suggère la comparaison, effectuée dans le paragraphe 4.2, des termes de preuve par co-induction portant sur des objets infinis avec les dérivations infinies qui construisent effectivement ces objets. Quoi qu'il en soit, toutes ces approches restent incomplètes et le problème de la définition d'une sémantique pour les dérivations infinies ne connait pas encore de solution générale. Les approches existantes sont exclusivement dédiées à la caractérisation d'objets infinis. L'approche que nous avons développée dans le dernier chapitre correspond à une démarche «opposée» : l'univers du discours considéré ne contient pas d'éléments infinis et seules les dérivations ne construisant aucun terme infini sont considérées (de telles dérivations correspondent à des termes de preuve par co-induction). Il s'agit là d'une restriction sur les dérivations qui permet de considérer le plus grand point fixe de l'opérateur T_P tout entier. En envisageant uniquement les dérivations qui ne calculent pas à l'infini, il a alors été possible de définir une sémantique valide et complète pour les dérivations qui «prouvent à l'infini».

Pour finir, signalons que, même si les résultats présentés dans le chapitre 4 n'ont pas fait l'objet d'une formalisation dans un méta-système logique, les problèmes «techniques» liés au renommage des clauses, évoqués dans le chapitre 3, n'ont pas été ignorés pour autant. Par exemple, la construction d'une dérivation à partir d'un arbre de preuve (théorème 4.12) nécessite la définition d'une «procédure récursive de renommage». Ici encore, le renommage constitue une étape essentielle lorsque l'on souhaite relier un objet déclaratif (l'arbre de preuve) à un objet «calculatoire» (la dérivation). De plus, ce sont uniquement les propriétés satisfaites par ce renommage qui permettent d'établir rigoureusement des résultats de ce chapitre.

Annexe A

Formalisation des théorèmes de point fixe

Nous présentons dans cette annexe une formalisation des résultats classiques sur les points fixes d'applications monotones et/ou continues, présentés dans le chapitre 1. Certains de ces résultats ont été utilisés lors de la formalisation de la preuve du théorème de complétude de la SLD-résolution.

Définitions

Soit A un ensemble quelconque (`A:Set`) et 2^A le type des prédicats sur A, caractérisant les parties de A.

```
Definition PA := A -> Prop.
```

Nous considérons une définition extensionnelle de l'égalité sur 2^A :

```
Definition EQP : PA -> PA -> Prop :=
[p1:PA][p2:PA]((a:A)((p1 a)->(p2 a)) /\ ((p2 a) -> (p1 a))).
```

munie de l'axiome d'extensionnalité :

```
Axiom Ext_EQ:(p1,p2:PA)(EQP p1 p2)->p1==p2.
```

Le type des relations sur 2^A est introduit par :

```
Definition RA := PA -> PA -> Prop.
```

Par exemple, la relation d'inclusion $\subseteq$ est une relation sur 2^A, définissant un ordre partiel (relation reflexive, antisymétrique et transitive), définie par :

```
Definition INCLUDE:RA:=[p1:PA][p2:PA]((a:A)(p1 a)->(p2 a)).
```

Soit 2^{2^A} le type des prédicats sur 2^A, caractérisant les parties de 2^A.

```
Definition PPA := PA -> Prop.
```

$\bigcup_{p(p_0)} p_0$	`Inductive PU [p:PPA;a:A]:Prop:=` `pu:(p0:PA)((p p0)/\(p0 a))->(PU p a).`
$\bigcap_{p(p_0)} p_0$	`Definition PI:PPA->PA:=` `[p:PPA]([a:A]((p0:PA)(p p0)->(p0 a))).`
$\mathsf{lub}(p)$	`Lemma p_sup : (p:PPA)(sup p INCLUDE (PU p)).`
$\mathsf{glb}(b)$	`Lemma p_inf : (p:PPA)(inf p INCLUDE (PI p)).`

TAB. A.1 – *Bornes supérieures et inférieures*

Etant donnée une partie p de 2^A (i.e. un élément de type 2^{2^A}), on définit, relativement à un ordre partiel r sur 2^A, la borne supérieure de p, notée $\mathsf{lub}(p)$, et la borne inférieure de p, notée $\mathsf{glb}(p)$, par :

```
Definition sup : PPA -> RA -> PA -> Prop :=
[p:PPA][r:RA][s:PA]
( ((p1:PA)(p p1) -> (r p1 s)) /\
  ((p2:PA) ((p3:PA)(p p3)->(r p3 p2)) -> (r s p2))).

Definition inf : PPA -> RA -> PA -> Prop :=
[p:PPA][r:RA][i:PA]
( ((p1:PA)(p p1) -> (r i p1)) /\
  ((p2:PA) ((p3:PA)(p p3)->(r p2 p3)) -> (r p2 i))).
```

Un ensemble partiellement ordonné est un treillis complet si pour chacune de ses parties p, $\mathsf{lub}(p)$ et $\mathsf{glb}(p)$ existent. On montre facilement que 2^A est un treillis complet. En effet, pour toute partie p de 2^A (i.e. un élément de type 2^{2^A}), on a (voir tableau A.1):

$$\mathsf{lub}(p) = \bigcup_{p(p_0)} p_0 \qquad \mathsf{glb}(p) = \bigcap_{p(p_0)} p_0$$

Le type des opérateurs sur 2^A est défini par :

```
Definition OP := PA -> PA.
```

Points fixes d'opérateurs monotones

Un opérateur φ sur 2^A est monotone s'il vérifie le prédicat :

```
Definition monoton : OP -> Prop :=
[o:OP]((p1,p2:PA)(INCLUDE p1 p2) -> (INCLUDE (o p1) (o p2))).
```

Les deux prédicats caractérisant les parties Φ_φ-closes et Φ_φ-denses, relativement à un opérateur φ, sont définis par :

```
Definition pclos:OP->PA->Prop:=[f:OP][p:PA](INCLUDE (f p) p).

Definition pdense:OP->PA->Prop:=[f:OP][p:PA](INCLUDE p (f p)).
```

Les deux ensembles $\mathsf{Ind}(\varphi)$ et $\mathsf{CoInd}(\varphi)$ sont alors définis par les prédicats :

```
Definition IND:OP->PA:=[f:OP](PI [p:PA](pclos f p)).

Definition COIND:OP->PA:=[f:OP](PU [p:PA](pdense f p)).
```

Enfin, les notions de points fixes suivantes sont définies :

```
Definition PF:OP->PA->Prop:=[f:OP][p:PA](EQP p (f p)).

Definition PPPF:OP->PA->Prop:=
[f:OP][p:PA]((PF f p)/\((p0:PA)(PF f p0)->(INCLUDE p p0))).

Definition PGPF:OP->PA->Prop:=
[f:OP][p:PA]((PF f p)/\((p0:PA)(PF f p0)->(INCLUDE p0 p))).
```

Ces définitions permettent de formaliser les théorèmes 1.1 et 1.3 :

```
Lemma Tarski : (f:OP)(monoton f) -> (PPPF f (IND f)).

Lemma Down_Tarski : (f:OP)(monoton f) -> (PGPF f (COIND f)).
```

Points fixes d'opérateurs continus

Afin de définir la notion d'opérateurs continus, nous introduisons le type des séquences infinies de parties de A par [36] :

```
CoInductive Type Stream:=si:PA->Stream->Stream.
```

Deux destructeurs et un observateur sont définis sur ce type :

```
Definition head_stream : Stream -> PA :=
      [s:Stream]<PA>Case s of
                         [p:PA][s0:Stream]p
                    end.

Definition tail_stream : Stream -> Stream :=
      [s:Stream]<Stream>Case s of
                             [p:PA][s0:Stream]s0
                        end.

Fixpoint nth [s:Stream;n:nat] : PA :=
 <PA>Case n of
          (head_stream s)
          [k:nat](nth (tail_stream s) k)
     end.
```

Une séquence infinie $(p_n)_{n\geq 0}$ de parties de A est dite :

- croissante si $p_0 \subseteq p_1 \subseteq \ldots \subseteq p_i \subseteq \ldots$

```
CoInductive SCPA:Stream->Prop:=
```

$\bigcup\limits_{n\geq 0} p_n$	`Definition UStream:Stream->PA:=` `[s:Stream][a:A](Ex [n:nat]((nth s n) a)).`
$\bigcap\limits_{n\geq 0} p_n$	`Definition IStream:Stream->PA:=` `[s:Stream][a:A]((n:nat)((nth s n) a)).`

$(\varphi(p_n))_{n\geq 0}$	`CoFixpoint OPS:OP->Stream->Stream:=`
	`[f:OP][s:Stream]<Stream>Case s of`
	`  [p:PA][s0:Stream](si (f p) (OPS f s0)) end.`

TAB. A.2 – *Opérations sur* $(p_n)_{n\geq 0}$

```
scpa_si:(s:Stream)
        ((INCLUDE (head_stream s)
                  (head_stream (tail_stream s))) /\
        (SCPA (tail_stream s))) -> (SCPA s).
```

- décroissante si $p_0 \supseteq p_1 \supseteq \ldots \supseteq p_i \supseteq \ldots$

```
CoInductive SDPA:Stream->Prop:=
sdpa_si:(s:Stream)
        ((INCLUDE (head_stream (tail_stream s))
                  (head_stream s)) /\
        (SDPA (tail_stream s))) -> (SDPA s).
```

Les principales opérations sur les séquences infinies de parties de A sont définies dans le tableau A.2. Un opérateur φ sur 2^A est dit :

- finitaire, si pour toute séquence croissante de parties de A, $(p_n)_{n\geq 0}$, on a :

$$\varphi(\bigcup_{n\geq 0} p_n) \subseteq \bigcup_{n\geq 0} \varphi(p_n)$$

```
Definition finitary:OP->Prop:= [f:OP]
((s:Stream)(SCPA s)->(INCLUDE (f (UStream s))
                              (UStream (OPS f s)))).
```

- $\uparrow$-continu, si pour toute séquence croissante de parties de A, $(p_n)_{n\geq 0}$, on a :

$$\varphi(\bigcup_{n\geq 0} p_n) = \bigcup_{n\geq 0} \varphi(p_n)$$

```
Definition CONTINU:OP->Prop:=[f:OP]
((s:Stream)(SCPA s)->(EQP (f (UStream s))
                          (UStream (OPS f s)))).
```

- $\downarrow$-continu, si pour toute séquence décroissante de parties de A, $(p_n)_{n\geq 0}$, on a :

$$\varphi(\bigcap_{n\geq 0} p_n) = \bigcap_{n\geq 0} \varphi(p_n)$$

```
Definition Down_CONTINU:OP->Prop:=
[f:OP]((s:Stream)(SDPA s)->(EQP (f (IStream s))
                                (IStream (OPS f s)))).
```

Afin d'introduire la notion de puissances ordinales d'un opérateur, nous définissons, de manière classique [63, 97], les nombres ordinaux par :

```
Inductive ORD : Set :=
 Oo : ORD | So : ORD -> ORD | Lo : (nat -> ORD) -> ORD.
```

Oo et So correspondent respectivement au zéro et au successeur d'un ordinal. Le constructeur Lo, appliqué à une séquence d'ordinaux, correspond à la borne supérieure de cette séquence. Le principe d'induction transfinie engendré par cette définition est :

```
ORD_rec : (P:ORD->Set)
 (P Oo) ->
   ((o:ORD)(P o)->(P (So o))) ->
      ((o:nat->ORD)((n:nat)(P (o n)))->(P (Lo o)))
         ->(o:ORD)(P o).
```

Nous pouvons à présent définir les puissances ordinales d'un opérateur φ sur 2^A.

$$\begin{array}{rcl} \varphi^{\uparrow 0} = \varnothing & & \varphi^{\downarrow 0} = 2^A \\ \varphi^{\uparrow \alpha+1} = \varphi(\varphi^{\uparrow \alpha}) & \alpha+1 \text{ est un successeur} & \varphi^{\downarrow \alpha+1} = \varphi(\varphi^{\downarrow \alpha}) \\ \varphi^{\uparrow \alpha} = \bigcup_{n\geq 0} \varphi^{\uparrow \alpha(n)} & \alpha \text{ est un ordinal limite} & \varphi^{\downarrow \alpha} = \bigcap_{n\geq 0} \varphi^{\downarrow \alpha(n)} \end{array}$$

```
Fixpoint POF [f:OP;o:ORD] : PA := <PA>Case o of
         [a:A]False
         [o0:ORD](f (POF f o0))
         [l:(nat -> ORD)]([a:A](Ex [n:nat]((POF f (l n)) a)))
                                     end.

Fixpoint Down_POF [f:OP;o:ORD] : PA := <PA>Case o of
         [a:A]True
         [o0:ORD](f (Down_POF f o0))
         [l:(nat -> ORD)]([a:A]((n:nat)((Down_POF f (l n)) a)))
                                          end.
```

L'ordinal limite ω est défini par :

```
Fixpoint ORD_FIRST [n:nat] : ORD :=
<ORD>Case n of
          Oo
          [p:nat](So (ORD_FIRST p))
     end.

Definition OMEGA : ORD := (Lo ORD_FIRST).
```

Ces définitions permettent de prouver formellement les résultats suivants :

Lemme 1.4.2 :

```
Lemma continu_is_monoton:(f:OP)(CONTINU f)->(monoton f).
```

Lemme 1.8.2 :

```
Lemma down_cont_is_monoton:(f:OP)(Down_CONTINU f)->(monoton f).
```

Lemme 1.5 :

```
Lemma is_continuous:(f:OP)(monoton f)->(finitary f)
                     ->(CONTINU f).
```

Lemme 1.4.1 :

```
Lemma S_UP:(f:OP)(o:ORD)(monoton f)
                         ->(INCLUDE (POF f o) (POF f (So o))).
```

Lemme 1.8.1 :

```
Lemma S_DOWN:(f:OP)(n:nat)(Down_CONTINU f)->
      (INCLUDE (Down_POF f (ORD_FIRST (S n)))
               (Down_POF f (ORD_FIRST n))).
```

Théorème 1.2 :

```
Lemma KLN:(f:OP)(CONTINU f)->(PPPF f (POF f OMEGA)).
```

Théorème 1.4 :

```
Lemma Down_KLN:(f:OP)(Down_CONTINU f)
                      ->(PGPF f (Down_POF f OMEGA)).
```

Notations, définitions

Φ ensemble de règles de la forme $e \leftarrow E$

T_Φ opérateur monotone associé à Φ (défini en (1.1), page 9)

φ opérateur monotone

$\varphi^{\uparrow\alpha}$, $\varphi^{\downarrow\alpha}$ puissances ordinales de φ (définies pages 11 et 14)

Φ_φ ensemble de règles associé à φ (défini page 10)

$\mathsf{lfp}(\varphi)$ plus petit point fixe de φ

$\mathsf{gfp}(\varphi)$ plus grand point fixe de φ

$\mathsf{lub}(E)$ borne inférieure de E

$\mathsf{glb}(E)$ borne supérieure de E

$\mathsf{Ind}(\Phi)$ ensemble défini inductivement à partir de l'ensemble de règles Φ (définition 1.1, page 8)

$\mathsf{Ind}(\varphi)$ ensemble défini inductivement à partir de l'opérateur monotone φ (définition 1.1, page 8)

$\mathsf{CoInd}(\Phi)$ ensemble défini co-inductivement à partir de l'ensemble de règles Φ (définition 1.2, page 13)

$\mathsf{CoInd}(\varphi)$ ensemble défini co-inductivement à partir de l'opérateur monotone φ (définition 1.2, page 13)

X ensemble de variables

A^X applications de X vers A (valuations)

Σ signature fonctionnelle

f_h^0 constante de Herbrand (symbole de fonction d'arité nulle)

Π signature relationnelle

$\mathbb{B}$ booléens ($\{\mathsf{true},\mathsf{false}\}$)

$\wedge_b$, $\vee_b$, $\neg_b$ connecteurs logiques sur $\mathbb{B}$

Prop propositions

$\wedge$, $\vee$, $\neg$ connecteurs logiques sur Prop

ar fonction d'arité ($\Sigma \cup \Pi \to \mathbb{N}$)

$T_\Sigma[X]$ ensemble des termes finis sur $\Sigma \cup X$ (définition 2.2, page 39)

$T_\Sigma^\infty[X]$ ensemble des termes finis et infinis sur $\Sigma \cup X$

$T_\Sigma[\varnothing]$ ensemble des termes finis fermés (i.e. sans variables) sur Σ

$L_{n,\Sigma}[X]$ ensemble des listes de termes de longueur n (définition 2.2, page 39)

$L_{n,\Sigma}[\varnothing]$ ensemble des listes de termes fermés de longueur n

$Q_\Sigma[X]$ ensemble des quasi-termes finis sur $\Sigma \cup X$ (définition 2.1, page 33)

P^T, P^L, $\mathcal{P}^T$, $\mathcal{P}^L_n$ prédicats caractérisant les quasi-termes compatibles avec les termes, les listes de termes (définis pages 41 et 47)

$Q^T_\Sigma[X]$, $Q^L_\Sigma[X]$ ensemble des quasi-termes compatibles avec les termes, les listes de termes

$\equiv_T$, $\equiv_{L_n}$ relations d'équivalence caractérisant les bijections entre $Q^T_\Sigma[X]$ et $T_\Sigma[X]$ et entre les éléments de $Q^L_\Sigma[X]$ de longueur n et $L_{n,\Sigma}[X]$ (définies page 43)

τ^t_q, τ^l_q fonctions des quasi-termes compatibles avec les termes, les listes de termes vers les termes, les listes de termes (définies pages 46 et 48)

τ^q_t, τ^q_l fonctions des termes, des listes de termes non vides vers les quasi-termes (définies page 49)

$S[X]$ substitutions sur les termes (applications de X vers $T_\Sigma[X]$)

$S[\varnothing]$ substitutions fermées (applications de X vers $T_\Sigma[\varnothing]$)

s_{id} substitution identité

$S_R[X]$ substitutions de renommage (applications de X vers X)

P^R prédicat sur $S_R[X]$ (défini page 73)

P_{RS} prédicat sur $S_R[X] \times S[X]$ (défini page 74)

$S_Q[X]$ quasi-substitutions, substitutions sur les quasi-termes (applications de X vers $Q_\Sigma[X]$)

P^S, $\mathcal{P}^S$ prédicats caractérisant les quasi-substitutions compatibles avec les termes (définis page 50)

$S^T_Q[X]$ quasi-substitutions compatibles avec les termes

$\equiv_S$ relation d'équivalence caractérisant la bijection entre $S^T_Q[X]$ et $S[X]$ (définie page 51)

τ^t_s fonction de $S^T_Q[X]$ vers $S[X]$ (définie page 51)

τ^q_s fonction de $S[X]$ vers $S^T_Q[X]$ (définie page 52)

$dom(s)$ domaine d'une substitution, d'une substitution de renommage, d'une quasi-substitution

$range(s)$ image d'une substitution, d'une substitution de renommage, d'une quasi-substitution

$At_{\Sigma,\Pi}[X]$ ensemble des atomes finis sur $\Pi \cup T_\Sigma[X]$ (défini page 71)

$At_{\Sigma,\Pi}[\varnothing]$ ensemble des atomes finis fermés sur $\Pi \cup T_\Sigma[\varnothing]$

$At^C_{\Sigma,\Pi}[\varnothing]$ ensemble des atomes fermés finis et infinis sur $\Pi \cup T^\infty_\Sigma[\varnothing]$ (complétion métrique définie page 117)

$At^C_{\Sigma_\perp,\Pi}[\varnothing]$ ensemble des atomes fermés finis et infinis sur $\Pi \cup T^\infty_{\Sigma\cup\perp}[\varnothing]$ (complétion définie page 134)

$At^C_{\Sigma,\Pi}[X]$ ensemble des atomes finis et infinis sur $\Pi \cup T^\infty_\Sigma[X]$ (complétion par idéaux définie page 126)

$[A]$ ensemble des instances fermées dans $At_{\Sigma,\Pi}[\varnothing]$ de l'atome A

$[\![A]\!]$ ensemble des instances fermées dans $At^C_{\Sigma,\Pi}[\varnothing]$ de l'atome A

$\approx$ relation d'équivalence (à un renommage près) définie page 147

$R_{\Sigma,\Pi}[X]$ ensemble des requêtes (défini page 71)

$r_{/n,a}$, $r_{/n}$ $n+1$-ième atome d'une requête (défini page 72)

$r[n \leftarrow r']$ requête r dans laquelle le $n+1$-ième atome a été remplacé par la requête r' (défini page 72)

$\Box$, $r_\varnothing$ requête vide

P_{RR} prédicat sur $S_R[X] \times R_{\Sigma,\Pi}[X]$ (défini page 74)

$[E]$ instances fermées finies des atomes apparaissant dans E

$\lceil E \rceil$ instances finies des atomes apparaissant dans E

$C_{\Sigma,\Pi}[X]$ ensemble des clauses (défini page 71)

C^+ tête de la clause C (défini page 71)

C^- corps de la clause C (défini page 71)

P_{RC} prédicat sur $S_R[X] \times C_{\Sigma,\Pi}[X]$ (défini page 74)

$P_{\Sigma,\Pi}[X]$ ensemble des programmes définis (défini page 72)

$[P]$ ensemble des instances fermées finies des clauses du programme défini P

$[\![P]\!]$ ensemble des instances fermées finies et infinies des clauses du programme défini P

$\lceil P \rceil$ ensemble des instances finies des clauses du programme défini P (définition 4.9, page 148)

$\lceil P \rceil^+$ sous-ensemble de $\lceil P \rceil$ (définition 4.16, page 159)

$H_{\Sigma,\Pi}[X]$ ensemble des clauses de Horn (défini page 72)

$var(E)$ liste des variables apparaissant dans E (terme, requête, clause, ...)

lg longueur :

- d'un quasi-terme (définie page 41)
- d'une requête (définie page 72)

Γ, P^Γ, $\overset{n,r,C}{\rightarrow}_P$ transitions SLD entre états de résolution (définies page 74)

P_{TC} prédicat caractérisant les couples de transitions composables (défini page 77)

$\mathbb{D}_g$, $\mathbb{D}_d$, $\overset{*}{\rightarrow}_P$, $P^{\mathbb{D}}$, $P^{\mathbb{D}_d}$ SLD-dérivations (définies pages 77 et 78)

τ_d^g fonction de $\mathbb{D}_d$ vers $\mathbb{D}_g$ (définie page 77)

τ_g^d fonction de $\mathbb{D}_g$ vers $\mathbb{D}_d$ (définie page 77)

$\vartheta(d)$ liste des variables apparaissant dans les clauses utilisées dans la dérivation d (définie page 78)

$\mathbb{L}_n[A]$ listes de longueur n constituées d'éléments dans A (définies page 89)

$\mathbb{F}_n[A]$ applications de $\mathbb{L}_n[A]$ vers A

$\mathbb{P}_n[A]$ applications de $\mathbb{L}_n[A]$ vers $\mathbb{B}$

$I_\Sigma[D]$, $I_\Pi[D]$ interprétations de Σ, de Π (définies pages 89 et 90)

$\mathcal{H}$ interprétation de Herbrand de Σ (définie page 97)

$H(I)$ interprétation de Herbrand associée à une interprétation I (définie page 99)

$\pi(I)$ prédicat caractérisant les atomes qui admettent l'interprétation de Herbrand I pour modèle

o^I schéma d'interprétation de o (terme, atome, requête, clause, ...) selon l'interprétation I (défini pages 90 à 92)

$\models_I o$ I est un modèle de o (atome, requête, clause, ...) (défini page 92)

$H \models R$ relation de conséquence sémantique sur $H_{\Sigma,\Pi}[X] \times R_{\Sigma,\Pi}[X]$ (définie page 92)

$\mathcal{M}_P$ plus petit modèle de Herbrand du programme défini P (défini page 100)

$\mathcal{M}_P^{\mathcal{C}}$ plus petit $\mathcal{C}$-modèle du programme défini P (défini page 148)

$\mathcal{M}_P^{\mathcal{C}+}$ plus petit $\mathcal{C}^+$-modèle du programme défini P (défini page 158)

S_P ensemble des succès fermés du programme défini P (défini page 103)

$S_P^{\mathcal{C}}$ ensemble des succès du programme défini P (définition 4.9, page 149)

C_P ensemble des atomes calculables à l'infini à partir du programme défini P :

- approche métrique : définition 4.4, page 118
- définition 4.5, page 121
- approche par CPO : définition 4.6, page 135

T_P opérateur de «conséquence immédiate» associé au programme défini P :

- définition classique page 101
- approche complétion par idéaux : défini page 126
- approche par contraintes : défini page 131
- approche par CPO : défini page 135

$\mathcal{O}(t)$ domaine, ensemble des occurrences de l'arbre t (défini page 115)

$\tau_n(t)$ opérateur de troncature de l'arbre t à la profondeur n (défini page 116)

$\leqslant$ préordre sur :

- les substitutions et les quasi-substitutions (défini page 55)
- les termes finis et infinis sur $\Sigma \cup \bot$ (défini page 133)
- les atomes de $At^{C}_{\Sigma_\bot,\Pi}[\varnothing]$ (défini page 134)

$\bowtie_l$ concaténation sur les listes de termes

$\bowtie_r$ concaténation sur les requêtes (définie page 72)

$\bowtie_v$ concaténation sur les listes de variables

$\bowtie_g$ concaténation sur $\mathbb{D}_g$ (définie page 77)

$\bowtie_d$ concaténation sur $\mathbb{D}_d$ (définie page 77)

`False_rec` axiome correspondant à la règle d'élimination du «faux» en logique intuitionniste (défini en (1.12), page 26)

arbre de preuve, arbre de preuve partiel définition 4.11, page 150

Bibliographie

[1] M. Abadi, L. Cardelli, P.L. Curien, and J.J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] M.A. Nait Abdallah. On the interpretation of infinite computations in logic programming. In J. Paredaens, editor, *11th International Colloquium on Automata, Languages and Programming, ICALP'84*, volume 172 of *Lecture Notes in Computer Science*, pages 358–370. Springer-Verlag, 1984.

[3] M.A. Nait Abdallah and M.H. van Emden. Top-down semantics of fair computations of logic programs. *Journal of Logic Programming*, 2(1):67–76, 1985.

[4] P. Aczel. An introduction to inductive definitions. In K.J. Barwise, editor, *Handbook of Mathematical Logic*, Studies in Logic and Foundations of Mathematics. North Holland, 1977.

[5] K.R. Apt. Logic programming. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 10, pages 493–574. The MIT Press, New York, 1990.

[6] K.R. Apt and R.N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19-20:9–72, 1994.

[7] K.R. Apt and M.H. Van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.

[8] H.P. Barendregt. Lambda calculi with types. In D.M. Gabbai Samson Abramski and T.S.E. Maiboum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

[9] B. Barras. Coq en Coq. Rapport de Recherche 3026, INRIA, 1996.

[10] M. Bellia, P.G. Bosco, E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. A two-level approach to logic plus functional programming integration. In A.J. Nijman, J.W. de Bakker, and P.C. Treleaven, editors, *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE). Vol. I: Parallel Architectures*, volume 258 of *Lecture Notes in Computer Science*, pages 374–393. Springer-Verlag, 1987.

[11] M. Bezem. Strong termination of logic programs. *Journal of Logic Programming*, 15(1-2):79–97, 1993.

[12] R.N. Bol, K.R. Apt, and J.W. Klop. An analysis of loop checking mechanisms for logic programs. *Theoretical Computer Science*, 86(1):35–79, 1991.

[13] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The *s*-semantics approach: Theory and applications. *Journal of Logic Programming*, 19-20:149–197, 1994.

[14] J. Chomicki and T. Imielinski. Relational specifications of infinite query answers. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 18(2):174–183, 1989.

[15] K. Clark. Predicate logic as a computational formalism. Research Report 79/59, Dept. of Computing, Imperial College, 1979.

[16] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer Verlag, New York, 1981.

[17] A. Colmerauer. PROLOG II, manuel de Référence et modèle théorique. Technical report, Groupe Intelligence Artificielle, Université Aix-Marseille II, 1982.

[18] A. Colmerauer, H. Kanoui, and M. Van Caneghem. PROLOG, bases théoriques et développements actuels. *Technique et Science Informatique*, 2(4), 1983.

[19] Project COQ. *The Coq Proof Assistant Reference Manual Version 6.2.* INRIA-Rocquencourt, 1998.

[20] T. Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Selected Papers of the 1st International Workshop on Types for Proofs and Programs, TYPES'93*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 1994.

[21] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988.

[22] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, 1983.

[23] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–94, 1992.

[24] B.A. Davey and H.A. Priestly. *Introduction to Lattices and Order.* Cambridge Mathematical Textbooks, 1990.

[25] F.S. de Boer, A. Di Pierro, and C. Palamidessi. Nondeterminism and infinite computations in constraint programming. *Theoretical Computer Science*, 151(1):37–78, 1995.

[26] N. Dershowitz, S. Kaplan, and D.A. Plaisted. Rewrite, rewrite, rewrite, rewrite, rewrite. *Theoretical Computer Science*, 83(1):71–96, 1991.

[27] C. Dubois and V. Viguié Donzeau-Gouge. A step towards the mechanization of partial functions: domains as inductive predicates. In *Workshop on mechanization of partial functions, CADE 15*, July 1998.

[28] E. Eder. Properties of substitutions and unifications. *Journal of Symbolic Computation*, 1(1):31–46, 1985.

[29] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A new declarative semantics for logic languages. In Kowalski and Bowen [61], pages 993–1005.

[30] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A model-theoretic reconstruction of the operational semantics of logic programs. *Information and Computation*, 103(1):86–113, 1993.

[31] M. Falaschi, G. Levi, C. Palamidessi, and M. Martelli. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[32] M. Fitting. *Computability Theory, Semantics, and Logic Programming.* Oxford University Press, New York, 1987.

[33] M. Fitting. Metric methods three examples and a theorem. *Journal of Logic Programming*, 21(3):113–127, 1994.

[34] J.H. Gallier. *Logic for computer science, foundations of automatic theorem proving.* Harper and Row Publishers, 1986.

[35] J.H. Gallier. On the correspondence between proofs and λ-terms. In P. de Groote, editor, *The Curry-Howard Isomorphism*, Cahiers du Centre de Logique, No. 8, pages 55–138. Université Catholique du Louvain, 1995.

[36] C.E. Giménez. *Un calcul des constructions infinies et son application à la vérification de systèmes communicants.* Thèse de doctorat, Ecole Normale Supérieure de Lyon, 1996.

[37] J.Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1989.

[38] W.G. Golson. Toward a declarative semantics for infinite objects in logic programming. *Journal of Logic Programming*, 5(2):151–164, 1988.

[39] M.J.C. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A mechanised logic of computation.* Springer-Verlag, 1979.

[40] M. Hagiya and T. Sakurai. Foundation of logic programming based on inductive definition. *New Generation Computing*, 2(1):59–77, 1984.

[41] L. Hallnäs. Partial inductive definitions. *Theoretical Computer Science*, 87(1):115–142, 1991.

[42] L. Hallnäs and P. Schroeder-Heister. A proof-theoretic approach to logic programming. I. Clauses as rules. *Journal of Logic and Computation*, 1(2):261–283, 1990.

[43] L. Hallnäs and P. Schroeder-Heister. A proof-theoretic approach to logic programming. II. Programs as definitions. *Journal of Logic and Computation*, 1(5):635–660, 1990.

[44] J. Harrison. Formalizing basic first order model theory. In J. Grundy and M. Newey, editors, *11th International Conference on Theorem Proving in Higher Order Logics TPHOLs'98*, volume 1479 of *Lecture Notes in Computer Science*, pages 153–170. Springer Verlag, 1998.

[45] J. Hein. Completions of perpetual logic programs. *Theoretical Computer Science*, 99(1):65–78, 1992.

[46] J. Herbrand. Recherches sur la théorie de la Démonstration (thesis 1930). *Logical Writings*, 1971.

[47] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.

[48] J. Jaffar and P.J. Stuckey. Canonical greatest fixpoints of logic programs. Technical Report 55, Dept. Computer Science, Monash University, 1985.

[49] J. Jaffar and P.J. Stuckey. Canonical logic programs. *Journal of Logic Programming*, 3(2):143–155, 1986.

[50] J. Jaffar and P.J. Stuckey. Semantics of infinite tree logic programming. *Theoretical Computer Science*, 46(2-3):141–158, 1986.

[51] M. Jaume. Formalization of SLD-resolution in the calculus of inductive constructions. Research Report 96-67, ENPC-CERMICS, 1996.

[52] M. Jaume. Unification des termes du premier ordre dans le calcul des constructions inductives. Research Report 96-58, ENPC-CERMICS, 1996.

[53] M. Jaume. Explicit proofs of well-known results in logic programming. Research Report 97-104, ENPC-CERMICS, 1997.

[54] M. Jaume. Formalisation de la SLD-résolution dans le calcul des constructions inductives. In *Actes des 6èmes Journées Francophones de Programmation Logique et programmation par Contraintes, JFPLC'97*, pages 277–291, Orléans, 1997. Hermès.

[55] M. Jaume. Unification : a Case Study in Transposition of Formal Properties. In E.L. Gunter and A. Felty, editors, *Supplementary Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics: Poster session TPHOLs'97*, pages 79–93, Murray Hill, N.J., 1997.

[56] M. Jaume. A full formalisation of SLD-resolution in the calculus of inductive constructions. *To appear in Journal of Automated Reasoning, Special Issue on Formal Proof*, 1998.

[57] M. Jaume. Logic programming and co-inductive definitions. Research Report 98-140, ENPC-CERMICS, 1998.

[58] R.S. Kemp and G.A. Ringwood. Reynolds and Heyting models of logic programs. In *ICLP94 Workshop on Proof Theoretical extensions of logic programming*, 1994.

[59] J.R. Kennaway, J.W. Klop, M.R. Sleep, and F.J. de Vries. Infinitary lambda calculus. *Theoretical Computer Science*, 175(1):93–125, 1997.

[60] H.P. Ko and M.E. Nadel. Substitution and refutation revisited. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 679–692. The MIT Press, 1991.

[61] R.A. Kowalski and K.A. Bowen, editors. *Proceedings of the Fifth International Conference and Symposium on Logic Programming*. ALP, IEEE, The MIT Press, 1988.

[62] J.L. Krivine. Une preuve formelle et intuitionniste du théorème de complétude de la logique classique. *The Bulletin of Symbolic Logic*, 2(4):405–421, December 1996.

[63] R. Lalement. *Computation as Logic*. Prentice Hall International Series in Computer Science, 1993.

[64] J.L. Lassez, M. Maher, and K. Mariott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. M. Kaufman, Los Altos, California, 1987.

[65] J.L. Lassez and M.J. Maher. Closures and fairness in the semantics of programming logic. *Theoretical Computer Science*, 29(1-2):167–184, 1984.

[66] G. Levi and C. Palamidessi. Contributions to the semantics of logic perpetual processes. *Acta Informatica*, 25(6):691–711, 1988.

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

[68] J.W. Lloyd and J.C. Sheperdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3-4):217–242, 1991.

[69] J.A. Makowsky. Why Horn formulas matter in computer science: Initial structures and generic examples. *Journal of Computer and System Sciences*, 34:266–292, 1987.

[70] Z. Manna and R.J. Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1(1-2):5–48, 1981.

[71] Z. Manna and R.J. Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, 1992.

[72] P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis, Napoli, 1984.

[73] P. Martin-Löf. Mathematics of infinity. In P. Martin-Löf and G. Mints, editors, *Proceedings, International Conference on Computer Logic, COLOG-88*, volume 417 of *Lecture Notes in Computer Science*, pages 146–197. Springer-Verlag, 1990.

[74] G. Nadathur and D. Miller. An overview of λ-PROLOG. In Kowalski and Bowen [61], pages 810–827.

[75] N.J. Nilsson. *Principles of Artificial Intelligence.* Symbolic Computation. Springer-Verlag, 1982.

[76] U. Nilsson and J. Maluszynski. *Logic Programming and Prolog.* Wiley, 1990.

[77] C. Palamidessi. Algebraic properties of idempotent substitutions. In M.S. Paterson, editor, *Proceedings, 17th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 443 of *Lecture Notes in Computer Science*, pages 386–399. Springer-Verlag, 1990.

[78] C. Palamidessi, G. Levi, and M. Falaschi. The formal semantics of processes and streams in logic programming. In *Colloquia Mathematica Societatis Janos Bolyai*, 42, pages 363–377, 1985.

[79] C. Paulin-Mohring. Inductive definitions in the system COQ. Rules and properties. In M. Bezem and J.F. Groote, editors, *Proceedings of the* 1^{st} *International Conference on Typed Lambda Calculi and Applications, TCLA'93*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, 1993.

[80] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system COQ. *Journal of Symbolic Computation*, 15(5-6):607–640, 1993.

[81] L.C. Paulson. Verifying the unification algorithm in LCF. *Science of Computer Programming*, 5:143–169, 1985.

[82] L.C. Paulson and A.W. Smith. Logic programming, functional programming, and inductive definitions. In P. Schroeder-Heister, editor, *Proceedings of the International Workshop on Extensions of Logic Programming*, volume 475 of *Lecture Notes in Computer Science*, pages 283–310. Springer-Verlag, 1989.

[83] H. Persson. *Constructive completeness of Intuitionistic Predicate Logic: A formalization in Type theory.* Licenciate thesis, Department of Computing Science, Chalmers University of Technology and University of Goteborg, Sweden, 1996.

[84] A. Di Pierro. *Negation and Infinite computations in Logic programming.* Ph.D. thesis, Università di Pisa-Genova-Udine, 1994.

[85] D. Prawitz. *Natural Deduction: A Proof-Theoretic Study*, volume 3 of *Stockholm Studies in Philosophy.* Almqvist & Wiksell, Stockholm, 1965.

[86] A.J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1), 1965.

[87] A.J. Robinson. The generalized resolution principle. *Machine Intelligence*, 3:77–94, 1968.

[88] A.J. Robinson. *Logic: Form and Function. The Mechanization of Deductive Reasoning.* Edinburgh University Press, 1979.

[89] J. Rouyer. Développement de l'algorithme d'unification dans le calcul des constructions avec types inductifs. Research Report 1795, INRIA-Lorraine, 1992.

[90] V.A. Saraswat, M.C. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 333–352, 1991.

[91] E.Y. Shapiro. Alternation and the computational complexity of logic programs. *Journal of Logic Programming*, 1(1):19–33, 1984.

[92] J.C. Shepherdson. Negation in logic progamming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. M. Kaufmann, San Mateo, CA, 1988.

[93] J.C. Shepherdson. The role of standardising apart in logic programming. *Theoretical Computer Science*, 129(1):143–166, 1994.

[94] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge, Ma, 1986.

[95] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[96] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[97] B. Werner. *Une théorie des constructions inductives*. Thèse de doctorat, Université Paris 7, 1994.