
Programmation, logique et calcul

Présentation d'un enseignement de maîtrise d'informatique

Catherine Belleannée — Olivier Ridoux

IFSIC, Université de Rennes 1, Campus Universitaire de Beaulieu
F-35042 RENNES cedex
{belleannee,ridoux}@irisa.fr

RÉSUMÉ. Nous présentons un enseignement donné en Maîtrise d'informatique à l'université de Rennes 1. Cet enseignement examine des relations entre logique et calcul, et parmi celles-ci la recherche d'une preuve considérée comme un calcul, d'où une initiation à la programmation logique. Cet enseignement n'est pas qu'un enseignement de programmation logique, mais c'est le seul que recevront sur le sujet les étudiants concernés. Même si cet enseignement privilégie les aspects fondamentaux, il comporte une série de travaux pratiques où on insiste sur l'usage effectif d'un environnement de programmation logique, ex. test et outil de trace.

ABSTRACT. We present a course given at the "Maîtrise d'informatique" level at Université of Rennes 1. This course studies relations between logic and computation, among which proof-searching as computing, hence a first course in logic programming. This course is not limited to logic programming, but it is the only one that the student will get on logic programming. Though this course focuses on theoretical matters, there is a series of practical works where we insist on the effective use of an actual logic programming environment, e.g., test and debugging.

MOTS-CLÉS : Pédagogie, programmation, logique, calcul, programmation logique, PROLOG

KEYWORDS: Pedagogy, programming, logic, computation, logic programming, PROLOG

1. Introduction

La Maîtrise d'informatique de l'IFSIC¹ comporte des enseignements de tronc commun en anglais, en génie logiciel et programmation, en compilation, en système d'exploitation, et en réseau, plus un enseignement au choix en « Mathématiques pour l'informatique » et en « Domaines de recherche ». Le premier enseignement au choix est le contexte de cet article, tandis que le second consiste à présenter des domaines de recherches représentés à l'IRISA².

La Maîtrise d'informatique compte environ 100 étudiants, dont 90 environ viennent de la Licence d'informatique de l'IFSIC. Cette Licence comporte des enseignements en anglais, en architecture des machines et système d'exploitation, en compilation, algorithmique et programmation, en base de données, et en logique et langages formels. Le cours de logique conduit les étudiants jusqu'au calcul des prédicats et à la résolution. L'enseignement de programmation recouvre entre autre un cours de programmation fonctionnelle.

L'enseignement « Mathématiques pour l'informatique » de la Maîtrise proposait jusqu'en 2001 le choix entre un cours « Automates et réseaux » sur la calculabilité et les réseaux de Petri, « Modélisation par files d'attente » sur la modélisation stochastique des systèmes informatiques, et « Programmation logique et par contraintes ».

Beaucoup d'étudiants choisissaient la troisième option pour échapper aux mathématiques, et de fait cette option n'en comportait presque pas. Elle consistait essentiellement en un cours de programmation PLC. Plus de la moitié des étudiants, dont les moins motivés, et les moins compétents, choisissaient cette option. Pour ceux-ci, la programmation logique n'était qu'un pis-aller. L'enseignant en charge de ce cours se plaignait souvent des conséquences de ce choix par défaut. Nous avons donc proposé de reprendre ce cours avec des objectifs nouveaux.

1.1. *Ce que nous voulions faire*

Inscrire PROLOG dans un cadre général pas nécessairement limité à la programmation en clauses de Horn. On suivra pour cela les idées développées par Dale Miller [MIL 91] sur les preuves uniformes. Pour Miller, la recherche de preuves uniformes est l'essence de la programmation logique. Ce cadre a l'avantage de pouvoir s'instancier de façons très diverses, ex. en logique linéaire [GIR 87, HOD 91, MIL 94], et en logique intuitionniste (voir λ PROLOG [MIL 86, BEL 99], mais aussi PROLOG).

Inscrire la programmation logique dans un cadre général de relation entre calcul et logique. L'ouvrage de Pierre Wagner [WAG 98] est une bonne référence

1. L'IFSIC (Institut de formation supérieur en informatique et communication) est l'UFR d'informatique de l'université de Rennes 1

2. L'IRISA (Institut de recherche en informatique et systèmes aléatoires) est une UMR (unité mixte de recherche) qui associe l'université de Rennes 1, l'INRIA, le CNRS et l'INSA de Rennes. C'est le débouché le plus proche pour les étudiants de l'IFSIC qui se dirigent vers la recherche.

sur ce thème. Il s'agit d'une étude sur l'émergence de la notion de machine en logique. Elle se présente comme un traité de philosophie, et est techniquement précise. Pour un usage plus technique, on peut avantageusement compléter cet ouvrage par le livre de René Lalement [LAL 90]. Nous retiendrons deux relations entre logique et calcul : la recherche de preuve considérée comme un calcul, qui conduit à la programmation logique et s'exprime bien dans le calcul des séquents, et la normalisation de preuve considérée comme un calcul, qui s'exprime bien en déduction naturelle et conduit à la programmation fonctionnelle *via* l'isomorphisme de Curry-Howard [HOW 80, BAR 91] et le λ -calcul. La programmation fonctionnelle ne sera pas développée en travaux dirigés et travaux pratiques puisqu'elle a fait l'objet d'un cours étendu en Licence, mais elle pourrait l'être dans un autre contexte.

Un enseignement de PROLOG « pratique ». Même si l'enseignement de PROLOG n'est pas le seul objectif de ce cours, nous voulions qu'il permette aux étudiants d'utiliser les dispositifs les plus concrets qu'offrent les systèmes PROLOG. Au nombre de ceux-ci, le traceur nous semble très important. Il sera donc présenté en cours, et les travaux pratiques insisteront sur son usage. Il ne s'agit pas de vanter tel ou tel outil, mais de mettre les étudiants en situation de bien utiliser l'outil qui leur est offert. En l'occurrence, il s'agit de SICSTUS PROLOG. Le fait que ce système respecte le standard PROLOG [DER 96] nous semble un plus dans l'optique d'une éventuelle utilisation future de PROLOG par nos étudiants.

Les lectures recommandées pour cet enseignement seront donc les ouvrages de René Lalement, Pierre Wagner et Richard A. O'Keefe [LAL 90, WAG 98, O'K 90].

1.2. Ce que nous ne voulions pas faire

Un cours « langage », fût-il PROLOG. Cela ne convient ni aux objectifs de la Maîtrise d'informatique, ni à ceux de l'option « Mathématiques pour l'informatique ». D'une façon générale, les étudiants ont déjà trop tendance à se focaliser sur les outils utilisés en travaux pratiques, et à oublier le contenu fondamental des enseignements. C'est ainsi qu'un cours de base de données peut facilement devenir un cours « ORACLE », et un cours de systèmes d'exploitation, un cours « UNIX ». Nous tenons donc à ce que le contenu fondamental de l'enseignement ne puisse pas être réduit à l'apprentissage d'un langage de programmation.

Partir de la résolution, même si c'est l'approche la plus souvent utilisée dans la présentation de PROLOG, même si c'est aussi une approche presque historiquement correcte, et même si l'article de Robinson [ROB 65] est passionnant.

Le procédé induit par le principe de résolution n'est *jamais* vraiment utilisé en programmation. Il suggère de

- 1) prendre une formule du calcul des prédicats,
- 2) y adjoindre la négation d'une question,
- 3) mettre le tout sous forme normale conjonctive,

4) appliquer le principe de résolution jusqu'à obtenir la clause vide, ou s'apercevoir qu'elle ne pourra pas être obtenue,

alors que la programmation en PROLOG commence directement en (3) en ne notant pas les clauses par des disjonctions, mais par des implications. Le fait que l'étape (2) ne fait pas partie du procédé réel de la programmation rend en plus le discours sur la polarité des littéraux illisible. En effet, la présentation des clauses sous forme d'implications rend tous les littéraux d'une clause de Horn positifs. Comment comprendre alors que permettre la négation dans les corps de clause pose problème car les clauses ont alors plus d'un littéral positif ?

La présentation de la programmation logique par la théorie de la résolution a de plus l'inconvénient de masquer le fait que la programmation logique ne se réduit pas à la programmation en clauses de Horn comprise sous la déduction classique.

2. Ce que nous avons fait

Les enseignements de l'option « Mathématiques pour l'informatique » disposent d'un volume de 18 heures de cours, 18 de travaux dirigés, et 12 de travaux pratiques.

2.1. Cours

2.1.1. Démarche

Comme indiqué plus haut, le cours présente les rapports entre preuves et programmes. Il comporte deux parties. L'une est consacrée à la recherche de preuve considérée comme un calcul, l'autre est consacrée à la normalisation de preuve considérée comme un calcul. La première conduit à la programmation logique, et la seconde à la programmation fonctionnelle. Nous ne développons ici que le contenu de la première partie. Celle-ci représente les deux tiers du cours.

2.1.2. Contenu

Ayant montré qu'une fonction f peut être représentée par une relation F , évaluer f en un point a revient à prouver $\exists x.F(a, x)$ en exhibant un x . Cette preuve doit bien sûr se faire dans le contexte d'un programme P , lui-même représenté par des formules logiques. On recherche donc une preuve de $P \models \exists x.F(a, x)$. Tout le problème est désormais d'étudier sous quelles conditions la recherche d'une telle preuve revient bien au calcul de $f(a)$. À partir de là, un « cahier des charges » plus précis est passé en revue. Il va amener à caractériser progressivement le type des preuves capables de rendre le service demandé.

Constructivité : la preuve de $P \models \exists x.F(a, x)$ doit permettre de déterminer une valeur pour x , qui tiendra lieu de résultat. Cette valeur est appelée un *témoin* de la preuve de l'existentielle. La preuve d'une disjonction fournit aussi un témoin lorsqu'elle permet de déterminer laquelle des branches a une preuve. On dit qu'une

logique dont les preuves fournissent toujours des témoins est *constructive*. Toutes les preuves ne sont pas constructives (ex. les preuves par l'absurde), et tous les systèmes logiques n'assurent pas que tous les théorèmes ont au moins une preuve constructive (ex. le calcul des prédicats classique, qui est un système logique dans lequel $a \vee \neg a$ est un théorème). Le calcul des prédicats classique ne répond donc pas au cahier des charges. Par contre, ses fragments intuitionnistes sont constructifs.

Pour avancer dans la discussion, nous décidons de présenter les preuves dans le cadre du calcul des séquents. Celui-ci présente l'avantage de fournir une représentation syntaxique des preuves, où le contexte du programme est en permanence visible. La figure 1 présente les règles logiques et la règle *Axiome* du calcul des séquents *LJ*. Ce système est correct et complet pour la logique des prédicats intuitionnistes.

Un séquent intuitionniste est une structure $g_1, g_2, \dots \vdash d$ qui exprime le jugement que la conjonction des g_i entraîne d . On appelle les g_1, g_2, \dots l'antécédent du séquent, et le d son conséquent. Une règle de déduction représente une étape de déduction. Le séquent sous la règle est sa conclusion, et les séquents qui sont au dessus sont ses prémisses. Les règles se lisent de haut en bas pour leur contenu déductif, et de bas en haut pour leur contenu opérationnel. Considérons plus concrètement les règles \wedge_G^i et \wedge_D . Les règles \wedge_G^i expriment que si un jugement $A, \Gamma \vdash D$ est un théorème, le jugement $A \wedge B, \Gamma \vdash D$ l'est aussi. Mais elles expriment aussi que pour chercher une preuve de $A \wedge B, \Gamma \vdash D$ on peut essayer une des règles \wedge_G^i . Le fait que plusieurs règles soient éligibles pour rechercher une preuve introduit un espace de recherche de preuve. La règle \wedge_D exprime que si les jugements $\Gamma \vdash A$ et $\Gamma \vdash B$ sont des théorèmes, le jugement $\Gamma \vdash A \wedge B$ en est un aussi. Le fait que cette règle possède plusieurs prémisses donne une structure d'arbre aux preuves. Dans le cadre de la recherche de preuve considérée comme un calcul, les antécédents sont le programme, et les conséquents sont des buts. Rechercher une preuve d'un séquent revient à composer un arbre dont la racine est le séquent à prouver, les feuilles sont des instances de la règle *Axiome*, et chaque nœud interne est une instance d'une règle de déduction.

Le système de règles présenté ici est amputé de ses règles « structurelles » qui expriment que les antécédents peuvent être dupliqués et permutés. On s'en est affranchi en considérant que la virgule qui sépare les antécédents est un constructeur d'ensemble. Rien n'oblige à ce que les antécédents soient disjoints ; il est même souvent nécessaire de considérer que dans « $A, \Gamma \vdash \dots$ » A est un élément de Γ . Par exemple, dans l'utilisation de la règle \forall_G , $\forall x(A)$ est toujours un élément de Γ . Le cours commence par le système *LK* du calcul des séquents classique, avec ses règles structurelles. Nous ne faisons ce raccourci ici que pour aller plus vite.

Une autre règle du système *LJ* manque à la figure 1 ; c'est la règle de coupure :

$$\frac{\Gamma \vdash A \quad A, \Delta \vdash F}{\Gamma, \Delta \vdash F} \text{ Coupure}$$

Elle présente l'inconvénient que tout séquent est une instance de sa conclusion, et que rien ne guide l'émergence du A . Elle n'est donc pas propice à une recherche de preuve guidée par le but. Cependant, le *Hauptsatz* de Gentzen [LAL 90] montre que cette

$$\begin{array}{c}
\frac{}{A, \Gamma \vdash A} \textit{Axiome} \\
\\
\frac{A, \Gamma \vdash D}{A \wedge B, \Gamma \vdash D} \wedge_G^1 \quad \frac{B, \Gamma \vdash D}{A \wedge B, \Gamma \vdash D} \wedge_G^2 \quad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge_D \\
\\
\frac{B, \Gamma \vdash D \quad \Gamma \vdash A}{A \Rightarrow B, \Gamma \vdash D} \Rightarrow_G \quad \frac{A, \Gamma \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow_D \\
\\
\frac{A, \Gamma \vdash D \quad B, \Gamma \vdash D}{A \vee B, \Gamma \vdash D} \vee_G \quad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee_D^1 \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee_D^2 \\
\\
\frac{\Gamma \vdash A}{\neg A, \Gamma \vdash} \neg_G \quad \frac{A, \Gamma \vdash}{\Gamma \vdash \neg A} \neg_D \\
\\
\frac{\Gamma, A[x \leftarrow t] \vdash D}{\Gamma, \forall x(A) \vdash D} \forall_G \quad \text{où } t \text{ est un terme quelconque} \quad \frac{\Gamma \vdash A[x \leftarrow c]}{\Gamma \vdash \forall x(A)} \forall_D \quad \text{où } c \text{ est un symbole qui n'apparaît ni dans } \Gamma, \text{ ni dans } D \\
\\
\frac{\Gamma, A[x \leftarrow c] \vdash D}{\Gamma, \exists x(A) \vdash D} \exists_G \quad \text{où } c \text{ est un symbole qui n'apparaît ni dans } \Gamma, \text{ ni dans } D \quad \frac{\Gamma \vdash A[x \leftarrow t]}{\Gamma \vdash \exists x(A)} \exists_D \quad \text{où } t \text{ est un terme quelconque}
\end{array}$$

Figure 1. Calcul des séquents intuitionnistes

règle est redondante. On peut donc s'en passer. Il est cependant important d'en parler dans un cours de « Programmation, logique et calcul » car elle joue un rôle important. En effet, la recherche de preuve sans coupure est le prototype de la programmation logique, l'élimination des coupures d'une preuve qui peut en comporter est le prototype de la programmation fonctionnelle, et le principe de résolution consiste à ne plus utiliser que la règle de coupure. Elle est donc à la fois redondante et essentielle. Dans le cadre de cet article, on ne recherche que des preuves sans coupure.

Pilotage par la conclusion : la recherche de preuve intuitionniste sans coupure reste un problème complexe où il n'est pas facile de prévoir la forme que prendra la recherche de preuve au vu du programme et du but. Notamment, la recherche d'une preuve laisse beaucoup de choix ouverts à chaque étape. En chaque nœud de la preuve, il y a essentiellement autant de règles applicables que de formules dans le séquent à prouver. Parmi celles-ci, certaines conduisent à des impasses car elles ne sont pas liées

au but. D'autres conduisent à des preuves redondantes qui ne sont que des permutations les unes des autres. Si on doit considérer la recherche d'une preuve comme un calcul, il faut que le choix des règles de déduction découle de façon prévisible du programme et du but. Il faut donc limiter le choix des règles à appliquer, et faire en sorte qu'il dépende du but. Ainsi, les règles de déduction deviennent les instructions d'une machine, et le but devient un programme qui contrôle l'activation de ces instructions.

La stratégie de preuve dite *uniforme* consiste à toujours sélectionner une règle à droite quand une s'applique (et en ce cas, une seule s'applique car le conséquent est un singleton), et dans le cas contraire sélectionner une règle à gauche qui concerne une formule en lien avec le but. Cette stratégie permet de considérer les règles de déduction à droite comme les instructions d'une machine, et le but comme le compteur ordinal de la machine. La recherche d'une preuve acquiert alors un contenu opérationnel évident.

Cette stratégie ne permet pas de prouver tous les théorèmes du calcul des prédicats intuitionnistes. Cela n'est pas acceptable car l'échec de la recherche d'une preuve peut maintenant avoir deux causes : le but n'est pas un théorème du calcul des prédicats intuitionnistes, ou le but est un théorème qui n'a pas de preuve uniforme. Cependant, il existe des fragments syntaxiques du calcul des prédicats intuitionnistes où tous les théorèmes ont des preuves uniformes. Par exemple, le fragment restreint aux connecteurs \Rightarrow , \wedge et \forall a cette propriété. En fait, on va employer un autre fragment encore plus restreint qui est celui des clauses de Horn :

$$\begin{array}{l}
\text{Programme} \rightarrow \text{Programme} \wedge \text{Programme} \\
\text{Programme} \rightarrow \text{Clause} \\
\text{Clause} \rightarrow \forall x \text{ Clause} \\
\text{Clause} \rightarrow \text{But} \Rightarrow \text{Atome} \\
\text{Clause} \rightarrow \text{Atome} \\
\text{But} \rightarrow \text{But} \wedge \text{But} \\
\text{But} \rightarrow \text{Atome} \\
\text{Atome} \rightarrow \text{Prédictat} (\text{Terme}^*)
\end{array}$$

Ce choix vient de ce que l'on souhaite atteindre PROLOG, mais il faut bien noter que le premier fragment peut aussi constituer la base d'un langage de programmation logique. En fait, ce langage existe et s'appelle λ PROLOG [MIL 86, BEL 99].

Exemple 1 (concaténation de liste) Soit P le programme formé des clauses

$$C_1 : \forall x [\text{concat}(\text{nil}, x, x)]$$

$$\text{et } C_2 : \forall e \forall x \forall y \forall z [\text{concat}(x, y, z) \Rightarrow \text{concat}(\text{cons}(e, x), y, \text{cons}(e, z))],$$

soit Q le but $\text{concat}(\text{cons}(1, \text{nil}), \text{cons}(2, \text{nil}), \text{cons}(1, \text{cons}(2, \text{nil})))$, une preuve uniforme de $P \vdash Q$ est comme suit :

$$\frac{\frac{\frac{\frac{}{Ax.}}{P, \text{concat}(\text{nil}, \text{cons}(2, \text{nil}), \text{cons}(2, \text{nil}))}{} \text{Ax.}}{\vdash \text{concat}(\text{nil}, \text{cons}(2, \text{nil}), \text{cons}(2, \text{nil}))}}{P \vdash \text{concat}(\text{nil}, \text{cons}(2, \text{nil}), \text{cons}(2, \text{nil}))} \forall_G \quad \frac{}{P, Q \vdash Q} \text{Ax.}}{P, \text{concat}(\text{nil}, \text{cons}(2, \text{nil}), \text{cons}(2, \text{nil})) \Rightarrow Q \vdash Q} \Rightarrow_G}{P \vdash Q} \forall_G^4$$

La règle étiquetée \forall_G^A est un raccourci pour 4 applications de la règle \forall_G à la clause C_2 , dans laquelle les variables universellement quantifiées e, x, y et z sont remplacées par $1, nil, cons(2, nil)$ et $cons(2, nil)$. La règle étiquetée \forall_G est appliquée à la clause C_1 , dans laquelle la variable x est remplacée par $cons(2, nil)$.

Malgré toutes les restrictions posées (constructivité, uniformité), le fragment des clauses de Horn est calculatoirement complet [AND 76, TÄR 77]. Notons aussi qu'à ce stade, la logique n'est pas dénaturée car les règles de déduction retenues viennent directement du système de départ LJ .

Stratégies de recherche et de preuve : Certaines règles ont la même conclusion (ex. \wedge_G^1 et \wedge_G^2), d'autres ont plusieurs prémisses. Les premières conduisent à plusieurs recherches de preuve, alors que les autres conduisent à des preuves arborescentes. Dans tous les cas, le système logique retenu, un fragment de LJ ne spécifie absolument pas dans quel ordre se font les différentes recherches de preuves, ou dans quel ordre les sous-preuves d'une preuve sont développées. PROLOG résulte de l'application de stratégies précises au problème de recherche de preuve.

Nous avons fait de cette question un point important du cours et des travaux dirigés, car c'est là que se fait la charnière entre la lecture logique des programmes PROLOG et leur lecture opérationnelle. Il nous semble important que les étudiants maîtrisent également bien les deux lectures.

Élimination des quantificateurs : Selon le fragment uniforme retenu, un sous-ensemble des règles de LJ est utilisé. Par exemple, pour le fragment des clauses de Horn, seules les règles *Axiome*, \wedge_D , \wedge_G^i , \forall_G et \Rightarrow_G sont utilisées. Elles ont toutes un contenu opérationnel évident sauf la règle d'élimination d'un quantificateur à gauche qui suggère de « deviner » quel t permettra de poursuivre la recherche de preuve (l'étape \forall_G^A de l'exemple 1 illustre bien cette divination). En fait, le t doit être choisi de telle sorte que la formule obtenue (ou ses sous-formules) puisse être utilisée dans une règle *Axiome*. D'où l'idée que le choix du t n'est pas fait lorsque la règle \forall_G est appliquée, mais seulement paresseusement lorsque la règle *Axiome* est employée. Dans ce cas, le jugement à prouver par la règle *Axiome* a la forme $A', \Gamma \vdash A$, et on recherche une substitution θ telle que $\theta(A') = \theta(A)$. La recherche d'une telle substitution est le problème de l'*unification*. Élimination des quantificateurs ne se fait donc plus par divination d'une valeur à substituer à une variable quantifiée, mais par la substitution à celle-ci d'une *inconnue*. Dans le cours, nous utilisons systématiquement « variable » pour le phénomène syntaxique de la quantification, et « inconnue » pour l'intermédiaire opérationnel qui permet une élimination de quantificateur paresseuse.

Cette reconstitution de PROLOG ne reflète pas l'historique de sa création, ni les habitudes des chercheurs du domaine. Nous ne croyons pas que cela soit un problème car notre objectif est ailleurs : présenter PROLOG dans un cadre qui permet de le comparer à autre chose. En effet, à toutes les étapes de la reconstitution on se pose des questions qui peuvent avoir d'autres réponses que celle retenue par PROLOG. L'approche traditionnelle basée sur la résolution ne permet pas ces connexions vers d'autres objets.

Un second avantage est qu'elle rend obligatoire de bien articuler la lecture opérationnelle et la lecture logique, car celles-ci ne se ressemblent pas du tout. Par exemple, la structure de la preuve n'est jamais construite dans la lecture opérationnelle, elle n'est que parcourue. Cependant, un avantage inattendu de cette reconstruction est que les deux lectures se superposent bien. Par exemple, on peut observer que le modèle des boîtes de Byrd [BYR 80] consiste essentiellement à superposer des boîtes aux règles de déduction. Un autre avantage est qu'une opération très administrative comme le renommage des variables des clauses, qui prend une place qui nous semble indue dans la présentation classique de PROLOG, est éliminée au profit de règles de déduction de bon aloi comme les règles \forall_G et \exists_G .

2.2. Travaux dirigés

Pour les travaux dirigés et pratiques, l'objectif a été d'étudier PROLOG en tant que langage de programmation pratique. Ce n'est donc pas là que se situe la plus grande spécificité de cet enseignement. L'utilisation pratique de PROLOG dans le cadre des travaux pratiques donne l'occasion de faire sentir la réalité du conflit entre théorie et pratique, et d'appréhender ce tiraillement entre la volonté de se conformer à un cadre formel et la nécessité de tenir compte de réalités plus pragmatiques (efficacité, entrées-sorties, etc.). Les travaux dirigés sont le lieu où s'observent les deux approches et où l'on étudie la charnière entre les deux.

Quand on observe les règles de déduction qui servent de base à la réalisation des preuves en PROLOG, la formule $a \wedge b \Rightarrow c$ est équivalente à la formule $b \wedge a \Rightarrow c$, et les ensembles de preuves obtenues en utilisant l'une ou l'autre des formules sont les mêmes. Pour ce système de règles, l'ordre des formules dans les antécédents d'implication, ainsi que l'ordre d'énoncé des formules dans le programme, sont neutres. Par contre, la stratégie d'utilisation des règles de déduction mise en oeuvre en PROLOG est contrainte : recherche en profondeur d'abord, exploration gauche-droite des corps de clauses, et ordre d'utilisation des clauses également figé : de haut en bas. Ainsi l'ordre d'exploration de l'arbre de recherche est complètement déterminé. De ce fait, les clauses $c:-a, b$ et $c:-b, a$ ne sont pas équivalentes au regard de cette stratégie. Cet aspect, évidemment bien connu de tout programmeur PROLOG, apporte un certain trouble au novice qui découvre PROLOG par le biais de la logique, car cette réalité est alors fortement contre intuitive, et décevante.

C'est d'ailleurs une des raisons pour lesquelles certains pédagogues préfèrent présenter PROLOG de façon opérationnelle dans un premier temps et n'introduire ses fondements en terme de preuve et de logique qu'a posteriori. Notre approche dans ce cours étant d'explorer longuement la théorie de la preuve pour observer le lien entre preuve et calcul, il y a un certain prix à payer au moment du basculement vers la présentation de PROLOG en tant que langage de programmation effectif. Le système de règles obtenu par la reconstruction (preuves intuitionnistes uniformes dans le fragment des clauses de Horn, avec élimination paresseuse des quantificateurs) satisfaisait apparemment le « cahier des charges ». Les contraintes ajoutées pour rendre Prolog

opérationnel peuvent sembler dénaturer la belle reconstruction. « Tout ça pour ça ? ». On échappe en partie à cette difficulté en commençant l'enseignement de PROLOG en travaux dirigés dès les premières séances ; c'est à dire alors que seul le cahier des charges de la recherche de preuve est posé.

Nous nous sommes attachés à montrer comment la stratégie de recherche détermine la sémantique opérationnelle de PROLOG, mais aussi comment on peut s'en affranchir par programmation et offrir d'autres types de recherche : en profondeur bornée ou en largeur. La sémantique opérationnelle est déterminée par les points suivants :

- lecture logique des clauses/lecture opérationnelle ;
- arbre de recherche, importance de l'ordre (des clauses, des buts), problème de la récursivité à gauche ;
- non déterminisme (recherche potentielle de tous les témoins de la preuve) ;
- la coupure ou le choix entre efficacité et logique ;
- l'unification vue comme l'opérateur de manipulation de termes (comparaison, affectation, construction) ;
- unification réversible et notion de relation, mais limites de la réversibilité (arithmétique, entrées-sorties).

Le dernier point introduit la notion de mode que nous utilisons systématiquement pour commenter les programmes. Il conduit aussi à explorer avec le point qui précède des usages originaux de l'unification comme les listes en différences.

2.3. Travaux pratiques

Nous insistons dans les travaux pratiques sur un usage effectif de PROLOG et de tous les outils d'un environnement PROLOG normal.

2.3.1. Objectifs

- Apprendre aux étudiants à bien utiliser l'outil à leur disposition, ici SICSTUS PROLOG. Notamment savoir utiliser le traceur nous semble très important.
- Le fait que par nature PROLOG comporte peu de garde-fous (pas de typage, absence de déclaration des objets) est aussi l'occasion de sensibiliser les étudiants aux vertus de certaines pratiques : noms d'identificateurs parlants ; indication de la spécification des prédicats (relations entre les paramètres, modes, ...).

2.3.2. Description des sujets

Les travaux pratiques sont répartis sur six séances de deux heures. Les deux premières séances ont un but d'initiation et sont consacrées à la réalisation de petits prédicats relativement indépendants. Les quatre suivantes proposent la réalisation de deux applications plus ambitieuses, occupant chacune deux séances.

2.3.2.1. Initiation

Les différents exercices proposés lors des deux premières séances permettent aux étudiants de s'approprier l'environnement de programmation de SICSTUS PROLOG, ainsi que de se familiariser avec les principales notions de PROLOG que sont le non-déterminisme, l'unification, la réversibilité des prédicats, et les structures de données (listes et autres termes construits), sans oublier les problèmes posés par la récursivité à gauche, l'arithmétique et la coupure. Les prédicats sont testés sur de petits programmes PROLOG de type base de données ou sur des listes. La nécessité de passer en revue un nombre important de notions différentes en peu de temps explique le découpage de ces séances en petites unités. Cet apprentissage de PROLOG doit conduire à une meilleure autonomie pour la réalisation des deux applications suivantes, de plus grande envergure et dont le déroulement est beaucoup moins guidé.

2.3.2.2. Analyse syntaxique et transformation de formules

La première des deux applications a pour objet l'analyse puis la transformation de formules logiques. Les formules considérées sont les formules du calcul des propositions. La première étape de la réalisation est consacrée à l'analyse des formules reçues sous forme de données brutes (des chaînes de caractères). Elle se décompose en une analyse lexicale, qui transforme la chaîne d'entrée en la liste de ses unités lexicales, suivie d'une analyse syntaxique écrite au moyen des DCG. L'analyse syntaxique produit un terme représentant l'arbre de syntaxe abstraite de la formule analysée.

La deuxième étape consiste à effectuer la mise sous forme normale conjonctive de la formule représentée par son arbre de syntaxe abstraite.

2.3.2.3. Apprentissage d'automates

La dernière application propose de mettre en œuvre un algorithme d'inférence grammaticale. Nous utiliserons l'algorithme RPNI, dont Jacques Nicolas, dans [NIC 99], propose une mise en œuvre en PROLOG. Étant donné un ensemble d'exemples et un ensemble de contre-exemples de mots d'un langage, il s'agit d'inférer un langage régulier compatible avec les exemples et les contre-exemples, et s'appuyant sur la structure des exemples pour en faire la généralisation. Pour cet algorithme, un langage est représenté par un automate fini. De façon un peu plus précise, l'automate appris est un automate qui généralise les exemples, mais seulement dans la mesure où la généralisation reste motivée par les exemples. Pour cela, la généralisation procède par fusions successives d'états de l'automate, tant que ces fusions ne conduisent pas à l'acceptation de contre-exemples du langage.

La mise en œuvre s'articule selon trois parties :

- l'application reçoit en entrée un ensemble d'exemples, un ensemble de contre-exemples et un vocabulaire. À partir de là, elle doit commencer par construire un automate de départ reconnaissant exactement tous les exemples et refusant exactement tous les contre-exemples. Cet automate a des états finaux positifs (état d'acceptation) et des états finaux négatifs (état de rejet).

– l’algorithme d’apprentissage lui-même. Sa mise en œuvre nécessite une bonne structuration du problème (calcul de niveaux de l’automate et parcours de l’automate, comportant éventuellement des boucles, pour fusionner en priorité les états de même niveau, etc.).

– à l’autre extrémité de l’application se présente l’affichage de l’automate produit.

Bien entendu, à chaque niveau de la conception devront être conçus des tests adaptés.

Les modules d’acquisition des exemples et contre-exemples, de construction de l’automate de départ, et d’affichage d’automates étaient fournis aux étudiants. Il leur restait à faire les modules de calcul de niveaux d’un automate, et de fusion d’états.

3. Évaluation

L’évaluation de cet enseignement, qui n’a été donné qu’une fois (au premier semestre de l’année universitaire 2002-2003), est assez délicate. Nous proposons de l’évaluer par l’attitude et les productions des étudiants en travaux pratiques, et par la réussite à l’examen. C’est bien sûr criticable, car le premier point est assez subjectif, et le second fait craindre une rétro-action perverse entre la satisfaction des étudiants, celle de l’enseignant, et les notes d’examen.

Un autre point semble totalement objectif ; les étudiants se sont répartis naturellement et équitablement entre les options. On n’a pas observé le choix par défaut mentionné plus haut (section 1). Cependant, il semble curieux de prendre pour signe positif une moindre popularité.

Comme il est en principe obligatoire de le faire, les étudiants évaluent les enseignements, mais ils le font sur une base annuelle qui ne permet pas d’avoir de l’information avant la fin de l’année universitaire.

3.1. Travaux pratiques

Les étudiants ont eu une attitude très positive en travaux pratiques. Beaucoup sont arrivés au bout de leur travail, et avec des résultats satisfaisants.

Les difficultés ne sont pas arrivées là où on les attendaient. La réalisation du programme d’apprentissage n’a pas posé de problème aux étudiants qui suivaient les consignes. Seul le test des programmes a posé problème. Les étudiants ne savent pas spontanément utiliser les procédures disponibles pour contruire un contexte de test complexe, ou pour interpréter les résultats des tests. Ici, il convenait d’utiliser le module de construction d’automate, et celui d’affichage, même pour le test unitaire. Ces difficultés sont celles de l’enseignement de la programmation, en général.

C’est le programme de manipulation de formules qui a posé le plus de problèmes. Écrire un analyseur lexical s’est révélé insurmontable pour beaucoup. Écrire une grammaire non-récursive à gauche n’est pas facile non-plus, mais n’est pas insur-

montable. Appliquer des identités logiques pour normaliser des formules est délicat, et c'est la seule difficulté que nous attendions.

3.2. Examens

Une première partie de l'examen (d'une durée totale de 3 heures) comportait des questions de manipulation de programmes synthétiques (ex. $p: -q, r$) pour en déterminer les arbres de recherche, et en prévoir le comportement. Une seconde partie était constituée de questions de recherche de preuve en calcul des séquents, ou en déduction naturelle, et d'une question sur la non-prouvabilité d'un séquent particulier. Enfin, une troisième partie était constituée d'un problème de programmation de recherche de chemins particuliers dans un graphe coloré.

La partie logique a révélé que si la plupart des étudiants étaient capables d'effectuer une recherche de preuve de séquent, ils n'en percevaient pas vraiment le sens. Ainsi, tous les étudiants qui ont traité la question de montrer que le séquent $\vdash \exists y.P(x, y) \Rightarrow \forall x.P(x, y)$ n'a pas de preuve ont montré qu'une recherche de preuve échouait pour conclure qu'aucune ne réussissait. Il faut donc être prudent sur la partie logique.

Pour la partie programmation, l'examen met surtout en évidence que les étudiants qui n'utilisent pas les principes de programmation donnés en cours, travaux dirigés et travaux pratiques n'arrivent pas à répondre aux questions que nous avons posées. Ceux-là essaient sans succès de construire des solutions *ad hoc* sans être guidés par une démarche. Cela illustre une difficulté plus générale de l'enseignement de la programmation ; l'important n'est pas seulement de maîtriser la syntaxe et la sémantique des langages de programmation, mais aussi de s'appropriier les idiomes propres à chaque langage (ex. listes en différence et DCG).

4. Conclusion et travaux futurs

Cet enseignement s'est déroulé de façon satisfaisante pour les enseignants, et semble l'avoir été aussi pour les étudiants. Faute d'évaluation moins subjective, on ne peut pas défendre notre approche pour son efficacité pédagogique, qui reste inconnue, mais plutôt pour sa largeur de spectre. Selon le temps disponible et le contexte pédagogique, on peut imaginer de faire plus ou moins de programmation, logique ou fonctionnelle, plus de travaux pratiques ou plus de théorie. Concernant la partie programmation logique, celle-ci peut s'instancier dans plusieurs langages concrets, selon les objectifs, le niveau des élèves, et le temps disponible. Un polycopié n'a pas pu être finalisé pour la fin de la première édition de ce cours, mais devra l'être pour le début de la seconde. Ce polycopié reprend à parts égales la recherche et la normalisation de preuve. Il peut donc servir de support à plusieurs variantes de cet enseignement.

5. Bibliographie

- [AND 76] ANDRÉKA H., NÉMETI I., « The Generalised Completeness of Horn Predicate-Logic as a Programming Language », DAI Research Report n° 21, 1976, University of Edinburgh.
- [BAR 91] BARENDREGT H., « Introduction to Generalized Type Systems », *J. Functional Programming*, vol. 1, n° 2, 1991, p. 125–154.
- [BEL 99] BELLEANNÉE C., BRISSET P., RIDOUX O., « A Pragmatic Reconstruction of λ Prolog », *J. Logic Programming*, vol. 41, n° 1, 1999, p. 67–102.
- [BYR 80] BYRD L., « Understanding the control flow of PROLOG programs », TÄRNLUND S.-Å., Ed., *Proceedings of the Logic Programming Workshop*, 1980, p. 127–138.
- [DER 96] DERANSART P., ED-DBALI A., CERVONI L., *Prolog: The Standard*, Springer, 1996.
- [GIR 87] GIRARD J.-Y., « Linear Logic », *Theoretical Computer Science*, vol. 50, 1987, p. 1–102.
- [HOD 91] HODAS J., MILLER D., « Logic Programming in a Fragment of Intuitionistic Linear Logic », KAHN G., Ed., *Symp. Logic in Computer Science*, Amsterdam, The Netherlands, 1991, p. 32–42.
- [HOW 80] HOWARD W., « The Formulae-as-types Notion of Construction », SELDIN J., HINDLEY J., Eds., *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, p. 479–490, Academic Press, London, 1980.
- [LAL 90] LALEMENT R., *Logique, réduction, résolution*, Études et recherches en informatique, Masson, 1990.
- [MIL 86] MILLER D., NADATHUR G., « Higher-Order Logic Programming », SHAPIRO E., Ed., *3rd Int. Conf. Logic Programming*, LNCS 225, Springer, 1986, p. 448–462.
- [MIL 87] MILLER D., NADATHUR G., SCEDROV A., « Hereditary Harrop Formulas and Uniform Proof Systems », GRIES D., Ed., *2nd Symp. Logic in Computer Science*, Ithaca, NY, 1987, p. 98–105, Revised version in [MIL 91].
- [MIL 91] MILLER D., NADATHUR G., PFENNING F., SCEDROV A., « Uniform Proofs as a Foundation for Logic Programming », *Annals of Pure and Applied Logic*, vol. 51, 1991, p. 125–157, Revised version of [MIL 87].
- [MIL 94] MILLER D., « A Multiple-Conclusion Meta-Logic », *Symp. Logic in Computer Science*, 1994, p. 272–281.
- [NIC 99] NICOLAS J., « Grammatical Inference as Unification », Rapport de recherche n° 3632, 1999, Inria.
- [O’K 90] O’KEEFE R., *The Craft of Prolog*, MIT Press, 1990.
- [ROB 65] ROBINSON J., « A Machine-Oriented Logic Based on the Resolution Principle », *J. ACM*, vol. 12, n° 1, 1965, p. 23–41.
- [TÄR 77] TÄRNLUND S.-Å., « Horn Clause Computability », *BIT*, vol. 17, 1977, p. 215–226.
- [WAG 98] WAGNER P., *La machine en logique*, Science, histoire et société, Presse Universitaire de France, 1998.