Belaïd Benhamou, Vincent Risch, Éric Würbel

ASP : un devenir de Prolog

Volume 5, nº 2-3 (2024), p. 177-202.

https://doi.org/10.5802/roia.78

La Revue Ouverte d'Intelligence Artificielle est membre du Centre Mersenne pour l'édition scientifique ouverte
www.centre-mersenne.org
e-ISSN : 2967-9672

Revue Ouverte d'Intelligence Artificielle
Volume 5, n° 2-3, 2024, 177-202

ASP : un devenir de Prolog

Belaïd Benhamou[a], Vincent Risch[a], Éric Würbel[a]

[a] Aix-Marseille Université, Laboratoire D'Informatique et systèmes, CNRS, UMR7020, Marseille (France)
E-mail : belaid.benhamou@lis-lab.fr, vincent.risch@lis-lab.fr, eric.wurbel@lis-lab.fr.

Résumé. — L'idée d'utiliser des formules logiques pour représenter de la connaissance et raisonner sur celle-ci remonte aux années 60, et en particulier à une proposition de McCarthy [23]. Une des directions de recherche émanant de cette proposition a conduit à la genèse du langage Prolog et au paradigme de la Programmation Logique [4]. Une évolution importante de prolog a été l'incorporation du mécanisme du *cut* qui a permis par la suite la mise en place de la *négation par échec*. Les recherches visant à établir une sémantique déclarative à la négation par échec ont débouché sur la notion de *modèle stable*, et sur un nouveau paradigme de la programmation logique succédant à Prolog et appelé ASP. Par la suite, des travaux se sont intéressés aux programmes logiquement consistants mais n'admettant pas de modèles stables. Un exemple de tels travaux est la *sémantique des extra-modèles*.

Mots-clés. — programmation logique, prolog, sémantique des modèles stables.

1. Introduction

L'idée d'utiliser des formules logiques pour représenter de la connaissance et raisonner sur celle-ci remonte aux années 60, et en particulier à une proposition de McCarthy [23]. Une des directions de recherche émanant de cette proposition a conduit à la genèse du langage Prolog et au paradigme de la Programmation Logique [4]. Au départ, Prolog a été conçu comme un outil générique de démonstration automatique portant sur un sous-ensemble « traitable » du Calcul des Prédicats. Cette première version, le Prolog *pur*, est complètement déclarative. La syntaxe restreinte du prolog pur, se limitant à un sous-ensemble des clauses de Horn, a permis d'obtenir un processus d'inférence efficace, tout en conservant une sémantique déclarative bien définie s'appuyant sur la notion d'inférence logique.

Prolog a évolué au cours du temps [5]. Une des évolutions les plus importantes a été l'incorporation du mécanisme du *cut*, conçu pour l'élimination autoritaire de modèles, mais permettant la mise en place de la *négation par l'échec* [2, 25]. Initialement, la sémantique de ce mécanisme a été définie dans Prolog de façon complètement procédurale, ce qui posait un certain nombre de problèmes quand à son utilisation dans le domaine de la représentation des connaissances. L'idée des travaux initiés à la fin des années 70 par Clark et Reiter était d'obtenir une sémantique déclarative de la négation par l'échec pour les programmes logiques.

Ce problème s'est avéré beaucoup plus difficile que prévu. Après une vingtaine d'années de travaux sur la question, la sémantique des modèles stables a enfin permis d'obtenir une sémantique déclarative satisfaisante qui conduit à la mise en place d'un nouveau paradigme de programmation logique succédant à Prolog et appelé ASP.

En particulier, le passage aux modèles stables a conduit à un basculement sur la manière de considérer « la » solution d'un programme logique : si en prolog, une solution se décrit en tant que dérivation obtenue à partir d'une requête, en ASP une solution se décrit comme un modèle « stable » du programme.

Parallèlement, les progrès effectués par les solveurs SAT ont permis le développement de solveurs ASP efficaces.

L'article est structuré de la façon suivante. La section 2 commence par une description des principes fondamentaux de Prolog : clauses définies, SLD-résolution et unification. L'aspect « langage de programmation » de Prolog est ensuite abordé, menant à la présentation du *cut* et de la négation par échec. Cette section s'achève en soulevant les problèmes posés par cette négation en Prolog. La section 3 présente l'answer set programming (ASP). En particulier, cette section met l'accent sur la prise en compte de la négation par échec en programmation logique au travers d'une sémantique bien définie, la sémantique des modèles stables. Par la suite, la section 4 donne un exemple de travaux visant à dépasser certains problèmes de la sémantique des modèles stables, notamment le fait que certains programmes logiquement consistants ne possèdent pas de modèle stable. Enfin, une conclusion est présentée en section 5.

2. Prolog

2.1. Principes fondamentaux

Le propos fondamental de Prolog est de mettre à la disposition d'un utilisateur un environnement lui permettant d'exprimer un ensemble de connaissances, et de raisonner (c'est-à-dire, calculer) sur celles-ci pour en tirer des conclusions valides. Concrètement, à l'idée de raisonnement/calcul est associé l'usage d'un outil de démonstration automatique, par le biais d'une représentation des connaissances considérées, effectuée sur un fragment de la logique du Premier Ordre qui soit précisément traitable par cet outil. La réalisation pratique d'un tel environnement est ainsi conditionnée par plusieurs résultats théoriques :

(1) la possibilité d'utiliser la notion de *clause définie* pour exprimer les connaissances ;
(2) la possibilité d'utiliser la *SLD-résolution* [4, 15, 21, 22] qui s'appuie sur
   - l'algorithme d'*unification* [27] pour effectuer un calcul simulant un raisonnement sur ces connaissances ;
   - l'usage du *théorème de Herbrand* pour conduire ce calcul à destination d'une conclusion valide.

Dans le détail, ces résultats s'expriment de la manière suivante :

Clause définie

Un *littéral* étant une formule atomique ou la négation logique d'une formule atomique (par exemple $P(x)$, $\neg Q(x, y)$, $R(z)$...), une *clause* étant une disjonction de littéraux (par exemple $P(x) \vee \neg Q(x, y) \vee R(z)$, et une *clause de Horn* une clause dont au plus un littéral est positif (par exemple $\neg Q(x, y)$, $P(x) \vee \neg Q(x, y)$...), une *clause définie* est une clause de Horn comprenant exactement un littéral positif (par exemple $P(x)$, $P(x) \vee \neg Q(x, y)$,...). Tout fragment de la logique du Premier Ordre ne comprenant que des clauses définies a pour propriété remarquable de toujours posséder un modèle minimal [9]. Ce résultat garantit qu'une conjonction de littéraux positifs dérive d'un ensemble de clauses définies si et seulement si chacun des littéraux est rendu vrai par le modèle minimal de cet ensemble de clauses définies.

SLD-résolution

La *SLD-résolution* (pour *Selected Linear Definite clause-resolution*) repose sur l'emploi unique de la règle d'inférence suivante, dont il aisé de montrer la validité :

$$\frac{\neg P \vee \neg Q_1 \vee \neg Q_2 \vee \cdots \vee Q_n \qquad P \vee \neg Q'_1 \vee \cdots \vee \neg Q'_m}{\neg Q_1 \vee \neg Q_2 \vee \cdots \vee Q_n \vee \neg Q'_1 \vee \cdots \vee \neg Q'_m}$$

Comme on le voit, à chaque étape d'une application de la règle, l'appariement du même littéral P, respectivement sous forme négative dans la première clause et sous forme positive dans la seconde clause, permet de produire une clause résolvante plus courte au sein de laquelle le littéral P est éliminé (on parle aussi d'*effacement* de ce littéral).

Comme l'indique son nom même, l'emploi de la règle de SLD-résolution conduit à la mise en place d'une stratégie linéaire de production des résolvantes pour peu qu'à chaque étape la dernière clause produite soit l'une des deux clauses mises en oeuvre à l'étape suivante (voir l'exemple considéré ci-après). D'autre part, la possibilité de produire une clause résolvante sur des littéraux prédicatifs construits à partir du langage du Premier Ordre — c'est-à-dire faisant intervenir des termes constitués de variables ou de fonctions intégrant ces variables — repose sur la possibilité d'*unifier* les termes en question, tel que l'a décrit Robinson dans un article fondateur [27]. Nous ne rentrons pas ici dans le détail de cet algorithme, mais pour en donner une idée naïve, il s'agit de tenter de rendre identiques les termes de deux mêmes prédicats par le biais d'un jeu de réécriture. Par exemple, considérons les deux occurences suivantes du même prédicat F :

$$F(x, 2, G(z)) \qquad F(G(1), y, G(4))$$

Il aisé de constater qu'il est possible de rendre identiques ces deux occurrences par le biais de la séquence d'affectations suivantes (appelée *Plus Grand Unifieur*) :

$$(x := G(1))(y := 2)(z := 4),$$

séquence qui permet de générer l'occurence commune $F(G(1), 2, G(4))$. Si ce prédicat F apparaît dans deux clauses définies, respectivement sous forme négative et

positive, la règle de SLD-résolution permet après unification (quand celle-ci est possible comme c'est le cas ici) de produire une clause résolvante dont F a été effacé. C'est ici qu'intervient un autre résultat fondamental, obtenu par Jacques Herbrand au début du vingtième siècle : une conjonction de littéraux est valide si et seulement si la clause vide est dérivable par réduction à l'absurde.

Autrement dit, afin de vérifier qu'une conjonction de littéraux est déductible d'un ensemble de clauses, il convient d'adjoindre la négation de cette conjonction (la disjonction des négations, obtenue par application des Lois de De Morgan) à l'ensemble initial de clauses, et de vérifier que la clause vide est dérivable. Comme on l'a vu, si l'ensemble initial de clauses n'est constitué que de clauses définies, l'application de la règle de SLD-résolution garantit la génération linéaire de clauses résolvantes de plus en plus courtes, jusqu'à obtention de la clause vide quand la conjonction initiale de littéraux est effectivement déductible de l'ensemble initial de clauses.

Afin d'illustrer la mise en œuvre de l'ensemble des mécanismes qui viennent d'être décrits, nous reprenons l'exemple historique proposé dans le premier ouvrage que l'équipe du Groupe d'Intelligence Artificielle a consacré à Prolog à l'époque. On considère un ensemble de divers mets (entrées, plats, desserts) que l'on souhaite combiner de façon à constituer un menu cohérent. On dispose d'un ensemble de connaissances de base sur les mets considérés, connaissances représentées par un ensemble de clauses définies unaires :

$[C_0]$ *hors-d'œuvre*(Artichauts Mélanie)	$[C_5]$ *poisson*(Bar aux algues)
$[C_1]$ *hors-d'œuvre*(Truffes sous le sel)	$[C_6]$ *poisson*(Sole meunière)
$[C_2]$ *hors-d'œuvre*(Cresson œuf-poché)	$[C_7]$ *dessert*(Sorbet aux poires)
$[C_3]$ *viande*(Grillade de bœuf)	$[C_8]$ *dessert*(Fraises chantilly)
$[C_4]$ *viande*(Poulet au tilleul)	$[C_9]$ *dessert*(Melon en surprise)

On dispose aussi d'un ensemble de connaissances structurées par les connections existantes entre prédicats. Ainsi, on sait qu'une viande est un plat, qu'un poisson est un plat, et aussi qu'un repas est constitué d'une entrée, d'un plat et d'un dessert, ce qu'énoncent les formules du Premier Ordre suivantes :

$$(\forall p)(viande(p) \Rightarrow plat(p)) \quad (\forall p)(poisson(p) \Rightarrow plat(p))$$

$$(\forall e)(\forall p)(\forall d)(hors\text{-}d'œuvre(e) \wedge plat(p) \wedge dessert(d) \Rightarrow repas(e, p, d))$$

Ces formules se traduisent à leur tour sous la forme de clauses définies :

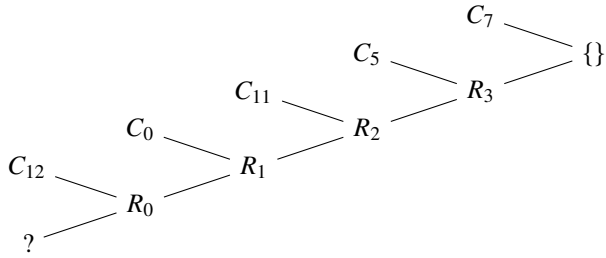
$$[C_{10}]\ \neg viande(p) \vee plat(p) \quad [C_{11}]\ \neg poisson(p) \vee plat(p)$$

$$[C_{12}]\ \neg hors\text{-}d'œuvre(e) \vee \neg plat(p) \vee \neg dessert(d) \vee repas(e, p, d)$$

On peut alors se poser la question, par exemple, de vérifier que *repas*(Artichauts Mélanie, Bar aux algues, Sorbet aux poires) est dérivable de notre ensemble de connaissances. Appliquant le théorème de Herbrand, on cherche donc à vérifier que la clause

$$[?]\ \neg repas(\text{Artichauts Mélanie, Bar aux algues, Sorbet aux poires})$$

conduit à la dérivation de la clause vide par SLD-résolution. On obtient effectivement par unifications successives une séquence linéaire de résolvantes partant de la question [?] et se terminant par la clause vide :



2.2. Prolog comme langage de programmation

Afin de faire de l'environnement de démonstration automatique qui a été décrit un outil de programmation, un pas supplémentaire est accompli en considérant l'ensemble des clauses utilisées sous l'angle de la notion abstraite de *programme logique*. En particulier, on définit ici et pour commencer la notion de *programme logique pur*, vu comme un ensemble fini de *règles* de la forme

$$A_0 \leftarrow A_1, \ldots, A_n$$

où $A_0, A_1, \ldots, A_n$ sont des atomes (autrement dit des littéraux positifs) et

- $A_0 \leftarrow A_1, \ldots, A_n$ est interprétée comme une *définition* de A_0, associée à la clause définie $A_0 \vee \neg A_1 \vee \cdots \vee \neg A_n$;
- $A_0 \leftarrow$ est interprétée comme un *fait* (écrit aussi $A_0 \leftarrow \top$), associé à la clause définie unaire A_0.
- A_0 est identifié comme la *tête* de règle, et la séquence $A_1 \ldots A_n$ comme le *corps* de la règle.

Intuitivement la flèche $\leftarrow$ correspond à l'implication logique (prise à l'envers) $\Leftarrow$ et correspond ainsi effectivement pour chaque règle à la clause définie associée. Par ailleurs, la syntaxe décrite dans cette définition se veut abstraite, alors qu'historiquement plusieurs syntaxes ont concrètement été définies pour Prolog en tant que véritable langage de programmation, parmi lesquelles historiquement la syntaxe « Marseille » et la syntaxe « Edinburgh ». En particulier,

- la règle $p(x) \leftarrow q(y)\ r(z)$ est codée par :

Syntaxe « Marseille »	Syntaxe « Edinburgh »
`p(x) -> q(y) r(z);`	`p(X) :- q(Y), r(Z).`

- Le fait $p(x) \leftarrow$ est codé par :

Syntaxe « Marseille »	Syntaxe « Edinburgh »
`p(x) -> ;`	`p(X).`

Alors que la syntaxe « Marseille » établit un lien extraordinairement élégant avec le concept de grammaire formelle par le biais des *grammaires de métamorphose* [3], la syntaxe « Edinburgh » met, elle, l'accent sur *l'effacement* des têtes de règles vues comme des *buts* dans un cadre de planification. Au delà, ce qui fait de Prolog un véritable langage de programmation a trait notamment à (1) l'intégration de structures faiblement typées (tuples et listes), (2) la généralisation du mécanisme d'unification, étendu à des équations/inéquations sur les arbres, (3) l'intégration d'un mécanisme d'élagage des modèles mis à disposition des programmeurs sous la forme d'un prédicat appelée *cut*. C'est à ce mécanisme, aussi intéressant que problématique, que nous consacrons le prochain paragraphe.

2.3. Cut et Négation par échec

Rapidement a été intégré aux implémentations existantes de Prolog un prédicat spécifique, noté « / » (Marseille) ou « ! » (Edinburgh), qui permet de supprimer des têtes de règles en attente de traitement. Par exemple, si en Prolog II (Marseille), on souhaite obtenir une énumération des plats disponibles dans notre menu, il suffit d'écrire

```
? plat(p)
```

pour obtenir l'énumération

```
> p = grillade_de_boeuf
> p = poulet_au_tilleul
> p = bar_aux_algues
> p = sole_meuniere
```

L'ajout du *cut* à la requête va conduire à l'élimination de toutes les affectations en attente après la première trouvée (c'est-à-dire de tous les modèles à l'exception du premier trouvé) :

```
? plat(p) /
> p = grillade_de_boeuf
```

Le prédicat *cut* a été initialement conçu par l'équipe du Groupe d'Intelligence Artificielle de Marseille comme un moyen commode de restreindre l'espace de recherche. Il n'est en effet pas rare dans l'écriture d'un programme symbolisant la connaissance que l'on a d'un sujet donné, que le premier modèle soit suffisant. Il se trouve qu'au delà de cette fonction utilitaire d'élagage, le prédicat *cut* prend un sens très particulier.

Considérons la question de constituer des menus constitués de mets plus ou moins légers. On peut par exemple décider qu'un fruit représente un dessert léger, à l'exception de tout autre dessert qui sera considéré plus lourd ; en Prolog II, sans usage du *cut*, ceci s'écrirait par exemple :

```
dessert(fruit, leger) -> ;
dessert(d, lourd) -> ;
```

Mais à la question de savoir à quel type de dessert appartient un fruit

```
? dessert(fruit, x)
```

le moteur d'inférence Prolog renvoie un résultat contre-intuitif :

```
> x = leger
> x = lourd
```

En effet, rien n'empêche d'unifier `d` avec `fruit` dans la seconde règle. Pour obtenir le résultat désiré, il est nécessaire d'employer un *cut* dans la première règle afin d'« oublier » volontairement qu'une unification restait en attente à partir de la seconde règle :

```
dessert(fruit, leger) ->  / ;
dessert(d, lourd) -> ;
```

Ce bout de programme renvoie cette fois à la question

```
? dessert(fruit,x)
```

le seul modèle

```
> x = leger
```

Autrement dit, « / » permet de préciser que *tout ce qui n'est pas précisément réputé léger est lourd*. On le voit, l'emploi du *cut* pose implicitement une façon particulière d'appréhender le sens d'un programme. Intuitivement, priver une solution de certains de ses modèles revient à accepter de l'affaiblir de telle façon que les modèles complémentaires des modèles retirés puissent potentiellement être intégrés à cette solution. Or le complémentaire du modèle d'un littéral est associé à la négation de ce littéral. Le *cut* introduit donc « par défaut » une forme de négation : jusqu'à quel point celle-ci calque t'elle effectivement la négation logique ? Consciente de cette question, l'équipe du Groupe d'Intelligence Artificielle a introduit avec ingéniosité l'expression explicite d'un prédicat appelé `not` doté de certaines des propriétés essentielles de la négation classique. Le prédicat en question s'écrit avec deux règles :

```
not(p) -> p / impasse ;
not(p) -> ;
```

où `impasse` est un prédicat qui n'existe pas dans le reste du programme. Le comportement de ces deux règles sur un éventuel fait `toto` donne en particulier que

- le prédicat `not(toto)` est vrai si `toto` est absent du programme ; en effet, dans ce cas aucune unification n'est possible dans le corps de la première règle entre `p` et `toto`, et le prédicat « / » n'étant pas atteint, l'unification se fait alors seulement dans la seconde règle, toujours entre `p` et `toto`, cette fois comme un fait ;
- le prédicat `not(toto)` est faux si `toto` est présent (et donc avéré) en tant que fait dans le programme ; en effet dans ce cas l'unification entre `p` et `toto` est possible dans le corps de la première règle, le prédicat « / » est atteint (et l'appel restant de la seconde règle, oublié), mais l'absence du prédicat `impasse` conduit à un échec.

Ainsi, on a bien que `not(toto)` est valide si `toto` ne l'est pas, et que `not(toto)` n'est pas valide si `toto` l'est. On peut, de la même façon, montrer aussi que `not(not(toto))` est valide si `toto` l'est. Le comportement du prédicat `not` semble donc correspondre à ce que l'on est en droit d'attendre d'une implémentation de la négation classique. Malheureusement, l'utilisation du *cut* dans un programme sort celui-ci du champs de la logique (et par conséquent `not` ne représente pas la négation logique). Pourquoi ? *cut* rend tout programme qui l'utilise sensible *à l'ordre d'évaluation des prédicats*. L'ordre d'écriture des règles devient pertinent (contrairement à ce qui se passe dans un programme logique pur), et l'ordre des requêtes aussi. Imaginons, toujours dans le cadre de notre exemple, qu'un restaurant garde en mémoire les préférences de ses clients les plus fidèles. Ainsi, Pierre aime tout autant bar aux algues et poulet au tilleul :

```
aime(pierre, bar_aux_algues) -> ;
aime(pierre, poulet_au_tilleul) -> ;
```

Toutefois, aujourd'hui, Pierre préfèrerait ne pas manger de poisson : quels sont alors les plats qu'il aime qui ne comportent pas de poisson ? A la requête « qu'est-ce qu'aime Pierre et qui n'est pas du poisson ? », autrement dit

```
? aime(pierre, x) not(poisson(x))
```

le moteur Prolog répond comme on peut s'y attendre, et à propos,

```
> x = poulet_au_tilleul -> ;
```

Mais à la requête théoriquement équivalente « qu'est-ce qui n'est pas du poisson et qu'aime Pierre ? », autrement dit

```
? not(poisson(x)) aime(pierre, x)
```

le moteur Prolog renvoie cette fois un échec ! Ainsi, en Prolog avec *cut*, le traitement (astucieux) de la négation reste inachevé. Le type de négation qui est représenté à partir du *cut* définit bien une négation différente, appelée *négation par échec*, caractérisée par le fait qu'au sein d'un domaine fini de connaissances, pour tout prédicat p qui n'est pas asserté, $not(p)$ est vrai.

En résumé, la négation par échec

- *n'est pas* la négation logique ;
- exprimerait l'*absence d'un prédicat* dans les conséquences d'un programme logique ;
- à ce titre, correspondrait à l'assertion d'une forme d'*incompatibilité entre prédicats absents ou présents dans un programme logique* ;
- est imparfaitement simulée par l'*élimination de modèles* que permet le *cut*.

C'est à la sémantique de la négation par échec dans le cadre de la programmation logique que sont consacrés les paragraphes suivants.

Pour terminer, nous présentons un exemple d'utilisation de la négation par échec qui est problématique pour Prolog, et dont nous verrons par la suite une solution dans le cadre de la sémantique des modèles stables définie dans le cadre de l'*answer set programming*.

Exemple 2.1. — Cet exemple définit les deux notions de « salade vinaigrée » et « salade citronnée » appliquée à une salade `x` :

```
salade(scarole) -> ;
vinaigree(x) -> salade(x) not(citronnee(x));
citronnee(x) -> salade(x) not(vinaigree(x));
```

La définition des prédicats `vinaigree(x)` et `citronnee(x)` stipule qu'une salade est vinaigrée si elle n'est pas citronnée, et inversement qu'une salade est citronnée si elle n'est pas vinaigrée.

On s'attendrait à ce que qu'une requête comme `vinaigree(x)` fournisse la réponse

```
vinaigree(scarole) ->;
```

mais il n'en est rien. En effet, une unification `x=scarole` va être réalisée pour effacer le sous-but `salade(x)`, puis Prolog va chercher à prouver que `citronnee(scarole)` n'est pas dérivable. Pour cela, le premier sous-but `salade(scarole)` est effacé, puis Prolog cherche à prouver que `vinaigree(scarole)` n'est pas dérivable. Cette dérivation circulaire sans fin conduit dans les faits à un dépassement de la pile servant à l'empilement des sous-buts à effacer. Notons que la requête `citronnee(x)` conduit à la même absence de résultat.

3. Answer Set Programming

ASP (Answer Set Programming) permet de décrire formellement le traitement de la négation par échec dans les programmes logiques en définissant la notion de *modèle stable* [12]. De façon à se concentrer uniquement sur cette notion, nous commencerons par l'aborder dans le cadre de programme définis sur des atomes propositionnels.

Une fois la notion de modèle stable définie, nous verrons comment ASP permet tout de même de traiter des programmes contenant des variables, quoi que d'une façon très différente de Prolog.

3.1. Syntaxe

Un *programme logique général* est un ensemble de règles de la forme :

$$A_0 \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n \tag{3.1}$$

où $A_0, \ldots, A_m, A_{m+1}, \ldots, A_n$ sont des atomes d'un langage propositionnel $\mathcal{L}_A$. *not* représente la négation par échec et la virgule la conjonction.

La notation $A_0 \leftarrow$ correspond à $A_0 \leftarrow \top$. Ces règles constituent les faits du programme. La notation $\leftarrow A_1, \ldots, A_m$ correspond à $\bot \leftarrow A_1, \ldots, A_m$. Ces règles sont appelées *contraintes d'intégrité*, nous en reparlerons dans la section 3.3.

Étant donnée une règle r de la forme (3.1), on définit :

$$\begin{aligned} head(r) &= \{A_0\} \\ body^+(r) &= \{A_1, \ldots, A_m\} \\ body^-(r) &= \{A_{m+1}, \ldots, A_n\} \\ body(r) &= body^+(r) \cup body^-(r) \end{aligned}$$

Un programme logique $\mathcal{P}$ est *positif* ou *pur* si $\forall r \in \mathcal{P}$, $body^-(r) = \emptyset$.

L'interprétation intuitive d'une règle de la forme (3.1) est : si les $A_1, \ldots, A_m$ sont valides, et qu'aucun des atomes $A_{m+1}, \ldots, A_n$ n'est prouvé, alors on peut inférer A_0.

Exemple 3.1. — Pour illustrer l'interprétation intuitive des règles d'un programme, considérons le programme suivant :

$$p \leftarrow \tag{3.2}$$

$$q \leftarrow p, not\ r \tag{3.3}$$

$$r \leftarrow p, not\ q \tag{3.4}$$

Ce qu'on peut dire de ce programme est :

- qu'en utilisant les règles (3.2) et (3.3), les atomes p et q sont dérivables ensemble.
- qu'en utilisant les règles (3.2) et (3.4), les atomes p et r sont dérivables ensemble.
- qu'en utilisant les règles (3.3) et (3.4), les atomes q et r (avec p qui est toujours dérivable) ne sont pas dérivables ensemble.

Dans la section suivante, nous allons expliciter formellement la sémantique des programmes logiques ainsi définis.

3.2. Sémantique

Définition 3.2. — *Étant donné $\mathcal{P}$, un programme logique positif :*

- *Soit X un ensemble d'atomes. X est* clos *pour $\mathcal{P}$ si, pour chaque règle $r \in \mathcal{P}$, $head(r) \in X$ si $body(r) \subseteq X$.*
- *L'ensemble des* conséquences *de $\mathcal{P}$, noté $Cn(\mathcal{P})$, est le plus petit ensemble d'atomes logiquement clos, et clos pour $\mathcal{P}$.*

Définition 3.3 (modèle stable d'un programme positif). — *Étant donné $\mathcal{P}$, un programme logique positif. $Cn(\mathcal{P})$ est l'unique modèle stable de $\mathcal{P}$.*

C'est aussi le modèle minimal de Herbrand que considère Prolog. On retombe alors sur le Prolog pur, sans le *cut*. Voyons maintenant comment étendre cette définition d'un modèle stable aux programmes généraux.

Définition 3.4 (réduit d'un programme). — *Soit $\mathcal{P}$ un programme logique général et X un ensemble d'atomes. On appelle* réduit de $\mathcal{P}$ relativement à X *l'ensemble*

$$\mathcal{P}^X = \{head(r) \leftarrow body^+(r) \ \mid \ r \in \mathcal{P}, body^-(r) \cap X = \emptyset\}$$

En d'autres termes, $\mathcal{P}^X$ est le programme positif obtenu à partir de $\mathcal{P}$:

- en supprimant chaque règle r telle que $body^-(r) \cap X \neq \emptyset$;
- en supprimant $body^-(r)$ des règles restantes.

Définition 3.5 (modèle stable d'un programme général [12]). — *Soit $\mathcal{P}$ un programme logique et X un ensemble d'atomes. X est un* modèle stable *de $\mathcal{P}$ si et seulement si $Cn(\mathcal{P}^X) = X$.*

Exemple 3.6. — Reprenons le programme $\mathcal{P}$ de l'exemple 3.1. On commence par considérer $X = \{p, q\}$. On obtient alors $\mathcal{P}^X = \{p \leftarrow,\ q \leftarrow p\}$. De plus, $Cn(\mathcal{P}^X) = \{p, q\}$. Donc $Cn(\mathcal{P}^X) = X$, et donc $\{p, q\}$ est un modèle stable de $\mathcal{P}$.

Si on considère maintenant $X = \{p, r\}$, on obtient $\mathcal{P}^X = \{p \leftarrow,\ r \leftarrow p\}$. On a alors $Cn(\mathcal{P}) = \{p, r\}$, donc $Cn(\mathcal{P}^X) = X$, et donc $\{p, r\}$ est un modèle stable de $\mathcal{P}$.

$\{p, q\}$ et $\{p, r\}$ sont les deux seuls modèles stables de $\mathcal{P}$. Pour les autres ensembles d'atomes envisageables, on obtient :

X	$\mathcal{P}^X$	$Cn(\mathcal{P}^X)$	$Cn(\mathcal{P}^X) = X$?
$\{q, r\}$	$\{p \leftarrow\}$	$\{p\}$	non
$\emptyset$	$\{p \leftarrow,\ q \leftarrow p,\ r \leftarrow p\}$	$\{p, q, r\}$	non
$\{p\}$	$\{p \leftarrow,\ q \leftarrow p,\ r \leftarrow p\}$	$\{p, q, r\}$	non
$\{q\}$	$\{p \leftarrow,\ q \leftarrow p\}$	$\{p, q\}$	non
$\{r\}$	$\{p \leftarrow,\ r \leftarrow p\}$	$\{p, r\}$	non
$\{p, q, r\}$	$\{p \leftarrow\}$	$\{p\}$	non

On peut d'ores et déjà noter quelques différences de comportement avec Prolog, illustrées par les exemples suivants.

Exemple 3.7. — Soit le programme $\mathcal{P} = \{p \leftarrow not\, q,\ q \leftarrow not\, p\}$. Sur une requête p ou q, Prolog *ne termine pas*, alors qu'ASP trouve deux modèles stables : $\{p\}$ et $\{q\}$.

Exemple 3.8. — Soit le programme

$$p \leftarrow q$$
$$p \leftarrow r$$
$$q \leftarrow not\ r$$
$$r \leftarrow not\ q$$

En Prolog, p n'est pas dérivable. En revanche, ASP renvoie deux modèles stables $\{p, q\}$ et $\{p, r\}$.

Propriété 3.9. — *Soit $\mathcal{P}$ un programme général. $Cn(\mathcal{P})$ est le plus petit modèle de $\mathcal{P}^X$ (au sens de l'inclusion), donc tout modèle stable M de $\mathcal{P}$ est un modèle* minimal *de $\mathcal{P}$ (au sens de l'inclusion).*

Propriété 3.10 (Non-cumulativité). — *Soit $\mathcal{P}$ un programme général. Si p est dans tout modèle stable de $\mathcal{P}$, il n'en découle pas que $\mathcal{P}$ et $\mathcal{P} \cup \{p \leftarrow\}$ ont les mêmes modèles stables.*

Notons qu'ASP est un formalisme non monotone, alors que la programmation logique au sens de Prolog est monotone.

La propriété suivante est une conséquence de la définition du réduit :

Propriété 3.11 (Programme sans modèle stable). — *Le programme $\{p \leftarrow not\ p\}$ n'a pas de modèle stable.*

D'une façon plus générale, si on considère un programme $\mathcal{P}_n$ comprenant $n + 1$ règles

$$p_i \leftarrow not\ p_{i+1} \quad (1 \leqslant i \leqslant n)$$
$$p_{i+1} \leftarrow not\ p_1$$

Si n est pair, $\mathcal{P}_n$ ne possède aucun modèle stable. Si n est impair, $\mathcal{P}_n$ possède exactement deux modèles stables : $\{p_{2i+1} \mid i \in \{0, \ldots, n/2\}\}$ et $\{p_{2i} \mid i \in \{0, \ldots, n/2\}\}$. Notez que le cas $n = 0$ correspond à la propriété 3.11, et le cas $n = 1$ correspond à l'exemple 3.7.

3.3. Contraintes d'intégrité

Nous avons vu avec la propriété 3.11 que l'ajout de la règle $p \leftarrow not\ p$ à un programme logique $\mathcal{P}$ «tue» tout modèle stable de $\mathcal{P}$. On peut étendre ce principe. En ajoutant à un programme $\mathcal{P}$ une règle $p \leftarrow q_1, \ldots, q_m, not\ r_1, \ldots, not\ r_n, not\ p$, on élimine tous les modèles stables de $\mathcal{P}$ qui contiennent $q_1, \ldots, q_m$ et ne contiennent pas $r_1, \ldots, r_n$. Ce type de construction permet de *filtrer* les modèles stables. L'utilisation des contraintes soutend la méthodologie dominante d'élaboration d'un programme logique répondant à un problème : la méthode *générer et tester*, qui divise un programme en deux parties : l'ensemble des règles qui va permettre de générer un ensemble de solutions potentielles (l'espace de recherche), et l'ensemble des contraintes qui va filtrer cet espace de recherche pour ne conserver que les solutions au problème.

Cette construction syntaxique, appelée *contrainte d'intégrité*, est si employée qu'elle fait l'objet dans tous les solveurs d'un raccourci syntaxique. Ainsi, la contrainte $p \leftarrow q_1, \ldots, q_m, not\ r_1, \ldots, not\ r_n, not\ p$ s'écrira tout simplement

$$\leftarrow q_1, \ldots, q_m, not\ r_1, \ldots, not\ r_n.$$

3.4. Variables et instanciation

Dans la section 2, nous avons vu que les atomes d'un programme Prolog peuvent contenir des variables, dont le traitement se base sur l'unification. Dans tout ce qui a été exposé précédemment à propos des programmes ASP, les atomes considérés sont des atomes propositionnels. Néanmoins, les outils de programmation ASP offrent la possibilité d'utiliser des variables dans les atomes. Ces variables ne sont pas traitées par un mécanisme d'unification. Un programme logique ASP avec variables est *instancié* au préalable. Le résultat de cette instanciation est un programme *fondé* (*ground*) ne contenant plus de variables et traitable par un solveur.

L'introduction du traitement des variables permet d'exprimer facilement les problèmes nécessitant une représentation des connaissances. Ces programmes sont ensuite instanciés pour pouvoir être traité par un solveur.

Un programme logique ASP avec variables se construit à partir :

- de *termes* qui se construisent à partir :
  - de symboles de constante $a, b, c, \ldots$;
  - de symboles de fonction $f, g, \ldots$;
  - de constantes numériques entières $1, 2, \ldots$;
  - de symboles de variables $X, Y, \ldots$;
  - de parenthèses $(,)$.

  Par exemple $f(3, c, Z), g((3, c), X), a$ sont des termes.
- d'atomes, qui sont constitués :
  - d'un symbole de prédicat $p, q, \ldots$;
  - d'arguments spécifiés entre parenthèses, chaque argument étant un terme.

  Par exemple, $p(f(3, c, Z), g(Y)), q()$ sont des atomes ($q()$ s'écrit q).
- de litéraux, un litéral étant un atome ou la négation par échec d'un atome.

Exemple 3.12 (Programme avec variables). — Le programme suivant est un programme avec variables :

$$\mathcal{P} = \{r(a, b) \leftarrow, r(b, c) \leftarrow, t(X, Y) \leftarrow r(X, Y)\}$$

Pour pouvoir être traité par un solveur, un programme avec variables doit être instancié. Pour cela, on définit les ensembles suivants :

- L'ensemble $\mathcal{T}$ des termes sans variables (l'univers de Herbrand du programme).
- l'ensemble $\mathcal{A}$ des atomes sans variables (la base de Herbrand du programme).

Une *instance fondée* (*ground instance*) d'une règle r d'un programme $\mathcal{P}$ est l'ensemble des règles sans variables obtenu en remplaçant toutes les variables de r par des éléments de $\mathcal{T}$:

$$ground(r) = \{r\theta \mid \theta : var(r) \rightarrow \mathcal{T} \text{ et } var(r\theta) = \emptyset\}$$

où $var(r)$ représente l'ensemble de toutes les variables apparaissant dans r, et θ est une substitution fondée.

L'instanciation fondée d'un programme $\mathcal{P}$ est alors définie par

$$ground(\mathcal{P}) = \bigcup_{r \in P} ground(r)$$

Exemple 3.13 (Instanciation fondée d'un programme). — Si on reprend l'exemple 3.12, on a :

- $\mathcal{T} = \{a, b, c\}$ (univers de Herbrand)
- $\mathcal{A} = \begin{Bmatrix} r(a,a), r(a,b), r(a,c), r(b,a), r(b,b), r(b,c), r(c,a), r(c,b), r(c,c), \\ t(a,a), t(a,b), t(a,c), t(b,a), t(b,b), t(b,c), t(c,a), t(c,b), t(c,c) \end{Bmatrix}$
- $ground(\mathcal{P}) = \begin{Bmatrix} r(a,b) \leftarrow, \\ r(b,c) \leftarrow, \\ t(a,a) \leftarrow r(a,a), t(b,a) \leftarrow r(b,a), t(c,a) \leftarrow r(c,a), \\ t(a,b) \leftarrow r(a,b), t(b,b) \leftarrow r(b,b), t(c,b) \leftarrow r(c,b), \\ t(a,c) \leftarrow r(a,c), t(b,c) \leftarrow r(b,c), t(c,c) \leftarrow r(c,c) \end{Bmatrix}$

Notons que les instancieurs, c'est-à-dire les programmes chargés de construire $ground(\mathcal{P})$ pour un programme $\mathcal{P}$ donné, procèdent à des optimisations visant à éliminer les instanciations inutiles dans le calcul des modèles stables du programme.

Par exemple, dans l'instanciation ci-dessus, l'instancieur détecterait le fait que la règle $t(a,a) \leftarrow r(a,a)$ est inutile : $t(a,a)$ n'est jamais dérivable puisque $r(a,a)$ ne l'est pas non plus. Par ailleurs si on considère la règle $t(a,b) \leftarrow r(a,b)$, il est possible de la réécrire comme un fait $t(a,b) \leftarrow$ puisque le fait $r(a,b) \leftarrow$ est un fait présent dans le programme. Au final, un instancieur asticieux produira l'instanciation fondée suivante :

$$ground(\mathcal{P}) = \{r(a,b) \leftarrow, r(b,c) \leftarrow, t(a,b) \leftarrow, t(b,c) \leftarrow\}.$$

Notez que le calcul du modèle stable de ce programme devient trivial.

Définition 3.14 (Modèles stables d'un programme avec variables). — *Soit $\mathcal{P}$ un programme logique général avec variables. Un ensemble X d'atomes fondés est un modèle stable de $\mathcal{P}$ si et seulement si $Cn(ground(\mathcal{P})^X) = X$*

Exemple 3.15. — Pour reprendre l'exemple 2.1 présenté dans la section 2.3 consacrée à la négation par échec en Prolog, le programme ASP $\mathcal{P}$ avec variables correspondant est

$$\begin{aligned} &salade(scarole) \leftarrow . \\ &vinaigree(X) \leftarrow salade(X), not\ citronnee(X). \\ &citronnee(X) \leftarrow salade(X), not\ vinaigree(X). \end{aligned}$$

L'instanciation $ground(\mathcal{P})$ est

$$salade(scarole).$$
$$vinaigree(scarole) \leftarrow salade(scarole), not\ citronnee(scarole).$$
$$citronnee(scarole) \leftarrow salade(scarole), not\ vinaigree(scarole).$$

Et les modèles stables de $\mathcal{P}$ sont $\{salade(scarole), vinaigree(scarole)\}$ et $\{salade(scarole), citronnee(scarole)\}$. Ainsi que nous l'avons vu précédemment, Les requêtes prolog `vinaigree(x)` et `citronnee(x)` n'ont pas de solution.

Cet exemple illustre le fait que la notion de modèle stable répond bien à l'intuition donnée par les règles du programme (ici l'intuition d'exclusion mutuelle de la dérivation des têtes des seconde et troisième règle).

Pour finir, on peut noter qu'en règle générale, les solveurs imposent que l'utilisation des variables dans les règles soit *sûre*. La propriété de sûreté dit que toute variable apparaissant dans une règle r doit apparaître dans son corps positif $body^+(r)$.

4. Extensions des sémantiques de point fixe dans l'ASP

Ici, nous nous intéressons à la nouvelle sémantique (sémantique des extra-modèles) introduite dans [1] pour donner un sens à une classe plus large de programmes logiques. Cette sémantique est différente de toutes les sémantiques itératives basées sur l'idée du point fixe. Dans cette nouvelle approche, les programmes logiques sont exprimés à l'aide de formules CNF (ensembles de clauses) d'une logique propositionnelle pour laquelle est définie une notion d'extension.

Il a été démontré dans cette sémantique [1] que chaque formule CNF consistante admet au moins une extension. En programmation logique, chacune de ces extensions exprime un modèle du programme représenté sous forme logique. Ce modèle pourrait être un modèle stable où un extra-modèle. En effet, pour chaque modèle stable donné d'un programme logique, il existe une extension de sa formule CNF correspondante qui l'implique logiquement. D'autre part, il a été montré que certaines extensions de la forme logique du programme logique considéré n'impliquent aucun modèle stable ; dans ce cas, une condition simple qui permet de reconnaître de telles extensions a été donnée. Ces extensions sont des extra-extensions qui représentent des extra-modèles. Elles sont très importantes car en l'absence de modèles stables, ces dernières garantissent au programme logique considéré une sémantique, grâce aux extra-modèles qu'elles expriment.

Les modèles représentés par ces extra-extensions ne sont pas capturés par la sémantique des modèles stables, qui échoue à donner un sens à certains programmes logiques. En effet, plusieurs programmes logiques consistants demeurent sans modèles stables et représentent un manque considérable pour la sémantique des modèles stables. La sémantique des extra-modèles étend celle des modèles stables dans le sens où elle garantit toujours la présence d'extra-modèles pour tout programme logique consistant n'admettant pas de modèles stables. Les modèles stables représentent une sous-classe des modèles de la sémantique des extra-modèles [1]. Une caractérisation complète de

cette sous-classe des modèles stables d'un programme logique au moyen des extensions de son codage logique CNF a été donnée dans [1]. Elle est caractérisée par l'ensemble des extensions vérifiant une condition simple dite descriminante. Enfin, la sémantique des extra-modèles ouvre une alternative nouvelle et importante pour la conception de solveurs ASP. C'est en ce sens qu'une première méthode de recherche de modèles stables et d'extra-modèles basée sur cette sémantique a été proposée dans [14].

4.1. Limites de la sémantique des modèles stables

Il existe des programmes logiques consistants qui n'admettent pas de modèles stables. La sémantique des modèles stables ne donne aucun sens à ces programmes, elle est donc insuffisante pour interpréter tous les programmes logiques consistants. L'exemple le plus frappant est la sous-classe de programmes qui ont des complétions de Clark inconsistantes. La sémantique des modèles stables ne peut pas capturer le sens de ces programmes. Les programmes logiques consistants sans modèles stables sont très nombreux, nous prenons à titre d'exemple le programme suivant :

Exemple 4.1. — Soit le programme logique $\mathcal{P} = \begin{cases} a \leftarrow not\ a \\ p \leftarrow \end{cases}$

Il est clair que le programme $\mathcal{P}$ est consistant et que l'atome p doit être vrai dans $\mathcal{P}$. Mais, Il n'existe pas de modèles stable pour $\mathcal{P}$. La sémantique des modèles stables ne capture pas le sens du programme $\mathcal{P}$.

D'un autre coté, certains programmes contiennent des modèles stables qui semblent manquer de sens logique. En effet, il est parfois difficile de trouver au vu des règles formant les programmes logiques que ces modèles sont censés interpréter, une intuition sur l'implication de certains atomes qui constituent ces modèles et l'absence d'autres atomes dans ces derniers. Cet aspect pourait se voir dans l'exemple suivant :

Exemple 4.2. — Soit $\mathcal{P} = \begin{cases} q \leftarrow not\ r \\ r \leftarrow not\ q \\ p \leftarrow not\ p \\ p \leftarrow not\ r \end{cases}$

Ce programme contient un seul modèle stable $M_1 = \{p, q\}$. Cependant au vu des règles constituant ce programme, on a aucune intuition logique du fait que p, q doivent être vrais et r faux dans M_1. On peut penser que des modèles du programme où r devrait être vrai manquent à la sémantique des modèles stables. Cet aspect peut être retrouvé dans plusieurs autres situations liées à différents programmes logiques.

4.2. Sémantique des extra-modèles

Cette sémantique suit une approche logique basée sur un langage propositionnel L où on distingue deux types de variables propositionnelles : un sous ensemble de variables classiques $V = \{A_i : A_i \in L\}$ et un autre sous-ensemble $nV = \{not\ A_i : not\ A_i \in L\}$. Pour chaque variable dans $A_i \in V$ il existe une variable correspondante

not $A_i \in nV$. L'ensemble total des variables de L est $V \cup nV$. On utilise le symbole *not* A_i pour exprimer un genre de négation par échec affaiblie proposée à la place de la négation par échec classique utilisée dans les programmes logiques. À ce niveau, les deux types de variables A_i et *not* A_i sont indépendants. C'est-à-dire, il n y a pas de relation sémantique entre les deux types de variables.

Depuis l'introduction de Prolog en 1972 et sa négation par échec, le but de la programmation logique est de définir un lien entre les deux types de variables. Particulierement, la logique des défauts [26], ou la logique des hypothèses [28, 29] représentent deux façons de formaliser un tel lien. Généralement, toutes les logiques non-monotonnes œuvrent dans ce sens.

Ici, la sémantique étudiée donne un lien entre les deux types de variables dans le cadre de la programmation par ensemble réponse (ASP). Inspiré de ce qui a été fait au niveau supérieur dans le cadre de la logique des hypothèses [28, 29], ce lien est exprimé par l'ajout au langage propositionnel L de l'axiome exprimant l'exclusion mutuelle entre A_i et *not* A_i, représenté par l'ensemble de clauses négatives :

$$ME = \{(\neg A_i \vee \neg not\ A_i) : A_i \in V\}$$

Remarque 4.3. — Cet axiome est équivalent à chacun des ensembles de formules :

- $\{\neg(A_i \wedge not\ A_i) : A_i \in V\}$
- $\{(A_i \rightarrow \neg not\ A_i) : A_i \in V\}$
- $\{(not\ A_i \rightarrow \neg A_i) : A_i \in V\}$.

C'est un pseudo-axiome qui s'applique seulement aux variables A_i de V. L'ensemble de clauses ME ainsi généré exprime seulement les exclusions mutuelles entre chaque littéral $A_i \in V$ et son littéral négatif correspondant *not* $A_i \in nV$, mais n'établit pas l'équivalence entre $\neg A_i$ et *not* A_i. En effet, on peut avoir $\neg not\ A_i$ sans qu'on ait A_i. C'est une sorte de négation par échec dont les condition sont affaiblies.

Dans la suite nous allons nous intéresser à la classe des programmes logiques généraux où un programme $\mathcal{P}$ est un ensemble de règles de la forme :

$$r : A_0 \leftarrow A_1, A_2, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n, (0 \leqslant m < n).$$

L'ensemble $STB = \{not\ A_i : not\ A_i \in \pi\} \subset nV$ des littéraux positifs du type *not* A_i qui apparaissent dans $\mathcal{P}$ est très important et représente en quelque sorte ce que Bart Selman et al. ont appellé dans [31] le *strong backdoor*. Ici l'ensemble STB est le strong backdoor de $\mathcal{P}$, c'est à dire le sous ensemble de variables sur lequel portera essentiellement le processus d'énumération des modèles et en conséquence la mesure de la complexité.

Dans cette approche, chaque règle r de $\mathcal{P}$ est traduite par une clause propositionnelle :

$$C\ : A_0 \vee \neg A_1 \vee \neg A_2, \ldots, \neg \vee A_m \vee \neg not\ A_{m+1} \ldots \neg not\ A_n.$$

On ajoute à cet ensemble de clauses traduites, l'ensemble des clauses d'exclusion mutuelle :

$$ME = \{(\neg A_i \vee \neg not\ A_i) : A_i \in V\}.$$

Un programme logique général :

$$\mathcal{P} = \left\{ \bigcup_{r \in \mathcal{P}} r : A_0 \leftarrow A_1, A_2, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n \right\},\ (0 \leqslant m < n)$$

est donc traduit par un ensemble $L(\mathcal{P})$ de clauses de Horn propositionnelles qui représente son codage CNF dans le langage propositionnel L :

$$\text{Soit } L(\mathcal{P}) = \begin{cases} \bigcup_{r \in \mathcal{P}} (A_0 \vee \neg A_1 \vee, \ldots, \neg \vee A_m \vee \neg not\ A_{m+1} \ldots \neg not\ A_n), \\ \bigcup_{A_i \in V} (\neg A_i \vee \neg not\ A_i). \end{cases}$$

Soit donc $L(\mathcal{P})$ le codage en calcul propositionnel du programme logique $\mathcal{P}$ et $STB \subset nV$ son strong backdoor. Nous allons nous intéresser aux extensions du couple $(L(\mathcal{P}), STB)$. Une extension de $(L(\mathcal{P}), STB)$ est la théorie obtenue à partir de $L(\mathcal{P})$ en ajoutant un ensemble maximal consistant de littéraux $not\ A_i \in STB$ à $L(\mathcal{P})$ tel que la théorie obtenue soit consistante. Nous sommes maintenant en mesure de définir la notion d'extension pour les programmes logiques :

Définition 4.4. — *Soit $L(\mathcal{P})$ le codage logique du programme $\mathcal{P}$, STB son strong backdoor, et un sous-ensemble $S' \subset STB$ de son strong backdoor. Alors $E = L(\mathcal{P}) \cup S'$ est une extension de $(L(\mathcal{P}), STB)$ si les conditions suivantes sont vérifiées :*

(1) *E est consistant*
(2) *$\forall not\ A_i \in STB - S', E \cup \{not\ A_i\}$ est inconsistant.*

Proposition 4.5. — *Soient $\mathcal{P}$ un programme logique, $L(\mathcal{P})$ sa forme logique CNF et STB son strong backdoor, si $L(\mathcal{P})$ est consistante alors $(L(\mathcal{P}), STB)$ admet au moins une extension.*

Cette proposition est d'un intérêt capital car elle donne la garantie d'avoir un modèle pour tout programme logique consistant.

Exemple 4.6. — Soit le programme logique :

$$\mathcal{P} = \begin{cases} a \leftarrow c, not\ b \\ b \leftarrow a \\ c \leftarrow not\ d \\ a \leftarrow \end{cases}$$

et son codage logique :

$$L(\mathcal{P}) = \begin{cases} L(\mathcal{P}) - Rules : & L(\mathcal{P}) - ME : \\ a \vee \neg c \vee \neg not\ b & \neg a \vee \neg not\ a \\ b \vee \neg a & \neg b \vee \neg not\ b \\ c \vee \neg not\ d & \neg c \vee \neg not\ c \\ a & \neg d \vee \neg not\ d \end{cases}$$

Le strong backdoor de $\mathcal{P}$ est l'ensemble $STB = \{notb, notd\}$, et la paire $(L(\mathcal{P}), STB)$ a une seule extension $E = L(\mathcal{P}) \cup \{not\ d\}$. En effet E est maximalement consistante

sur l'ensemble STB. On peut remarquer que si on ajoute *not b* à E, l'ensemble obtenu deviendrait inconsistant.

Exemple 4.7. — On considère maintenant, le programme logique $\mathcal{P}$ de l'exemple 4.6 dans lequel on supprime la règle $a \leftarrow$. On obtient, alors le programme logique $\mathcal{P}'$:

$$\mathcal{P}' = \begin{cases} a \leftarrow c, not\ b \\ b \leftarrow a \\ c \leftarrow not\ d \end{cases}$$

Le programme $\mathcal{P}'$ a pour codage logique CNF $L(\mathcal{P}') = L(\mathcal{P}) - \{a\}$ et pour STB le même ensemble que celui du programme $\mathcal{P}$.

Dans cet exemple, on obtient deux extensions de $(L(\mathcal{P}'), STB)$: $E_1 = L(\mathcal{P}') \cup \{not\, d\}$ et $E_2 = L(\mathcal{P}') \cup \{not\, b\}$. Mais $\mathcal{P}'$ n'a pas de modèle stable. Les deux extensions en question sont des extra-extensions sur lesquelles on reviendra dans la suite. Nous allons maintenant donner un théorème qui stipule que tout modèle stable correspond à une extension.

Théorème 4.8 (Benhamou et Seigel 2012). — *Un ensemble d'atomes X est un modèle stable d'un programme logique $\mathcal{P}$, si et seulement si il existe une extension E de $(L(\mathcal{P}), STB)$ vérifiant la* condition discriminante *: $(\forall A_i \in V, E \models \neg not\, A_i \Rightarrow E \models A_i)$ telle que $X = \{A_i \in V : E \models A_i\}$.*

Exemple 4.9. — Soit le programme logique $\mathcal{P}$ de l'exemple 4.6 et son codage CNF $L(\mathcal{P})$. $\mathcal{P}$ a un seul modèle stable $X = \{a, b, c\}$ qui correspond à l'unique extension $E = L(\mathcal{P}) \cup \{not\, d\}$ de $(L(\mathcal{P}), STB)$. En effet, après l'affectation des variables du STB, on utilise la résolution unitaire pour déduire que $E \models \{a, b, c, \neg d, \neg not\, a, \neg not\, b, \neg not\, c\}$. On voit ici que E vérifie bien la condition discriminante $(\forall A_i \in V, E \models \neg not\ A_i \Rightarrow E \models A_i)$ et que $X = \{A_i \in V : E \models A_i\} = \{a, b, c\}$.

La proposition 4.5, affirme que $(L(\mathcal{P}), STB)$ a toujours une extension si $L(\mathcal{P})$ est consistant. Mais un programme logique $\mathcal{P}$ peut ne pas avoir de modèle stable. Il peut donc exister des extensions (extra-extensions) de $(L(\mathcal{P}), STB)$ qui ne correspondent à aucun modèle stable de $\mathcal{P}$. Ces extra-extensions sont exactement celles qui ne vérifient pas la condition discriminante. Ces extra-extensions, non prises en compte par la sémantique des modèle stables, sont très intéressantes pour les cas de programmes logiques consistants n'ayant pas de modèles stables. Elles sont également importantes pour la représentation des connaissances, mais on ne discutera pas de cet aspect dans cet article.

Par exemple, dans l'exemple 4.7, on a deux extensions de $(L(\mathcal{P}'), STB)$:
$E_1 = L(\mathcal{P}') \cup \{not\, d\}$ et $E_2 = L(\mathcal{P}') \cup \{not\, b\}$ qui ne correspondent pas à des modèles stables.

On peut facilement vérifier que $E_1 \models \{c, \neg d, \neg not\, b, \neg not\, c\}$ et que $E_2 \models \{\neg b, \neg a, \neg c, \neg not\, d\}$. Mais aucune de ces deux extensions ne vérifie la condition discriminante du Théorème 4.8. En effet, E_1 implique $\neg not\ b$ mais n'implique pas b et d'un autre

coté, E_2 implique $\neg not\, d$ mais n'implique pas d. Il s'agit bien de deux extra-extensions du programme logique $\mathcal{P}'$.

Ci dessous on donne deux exemples d'école très utilisés par la communauté logique des défauts et raisonnement non-monotone. On remarque que la sémantique des extra-modèles est valide pour les deux exemples.

Exemple 4.10. — Soit le programme $\mathcal{P} = \{a \leftarrow not\ a\}$ composé d'une seule règle. Ce programme n'a pas de modèle stable. Son codage CNF est $L(\mathcal{P}) = \{a \vee \neg not\, a, \neg a \vee \neg not\, a\}$ et son strong backdoor est $STB = \{not\, a\}$. La paire $(L(\mathcal{P}), STB)$ a une extra-extension $E = \{\neg not\, a\}$. En effet par résolution sur les deux clauses de $L(\mathcal{P})$ on infère la clause unitaire $\neg not\, a$ qui subsume les clauses de $L(\mathcal{P})$. Donc $E \models \neg not\, a$ et $E \not\models a$. L'extension E ne vérifie pas la condition discriminante. On prouve donc pour cet exemple que $(L(\mathcal{P}), STB)$ a une seule extension qui est une extra-extension. La sémantique des extra-modèles capture bien la sémantique classique des modèles stables pour cet exemple.

Exemple 4.11. — On reprend le programme logique donné dans l'exemple 4.10 auquel on ajoute une règle $a \leftarrow$. On obtient le programme $\mathcal{P} = \{a \leftarrow nota, a \leftarrow\}$ qui a un unique modèle stable $X = \{a\}$. Son codage CNF est $L(\mathcal{P}) = \{a \vee \neg nota, \neg a \vee \neg nota, a\}$ et son strong backdoor est $STB = \{not\ a\}$. La paire $(L(\mathcal{P}), STB)$ a une extension $E = L(\mathcal{P})$, telle que $E \models \neg not\ a$ et $E \models a$. Cette extension vérifie bien la condition discriminante et implique le modèle stable $X = \{a\}$ de $\mathcal{P}$. La sémantique des extra-modèles capture bien ici aussi les modèles stables du programme logique considéré.

On donne ci dessous deux autres exemples classiques qui viennent de la logique des défauts.

Exemple 4.12. — Soit le programme logique $\mathcal{P}$ composé de deux règles :

$$\mathcal{P} = \{a \leftarrow not\ b, b \leftarrow not\ a\}$$

Son codage CNF est l'ensemble de clauses $L(\mathcal{P}) = Rules \cup ME$:

$$Rules = \{a \vee \neg not\ b, b \vee \neg not\ a\}$$

$$ME = \{\neg a \vee \neg not\ a, \neg b \vee \neg not\ b\}$$

Son strong backdoor est l'ensemble $STB = \{not\ a, not\ b\}$. La paire $(L(\mathcal{P}), STB)$ a deux extensions $E_1 = L(\mathcal{P}) \cup \{not\, a\}$ and $E_2 = L(\mathcal{P}) \cup \{not\, b\}$. Par résolution unitaire on déduit que $E_1 \models \{b, \neg a, \neg not\ b\}$ and $E_2 \models \{a, \neg b, \neg not\ a\}$. Donc ces extensions vérifient la condition discriminante. À partir de E_1 resp. E_2 on obtient les ensembles de littéraux positifs impliqués $X_1 = \{b\}$ resp. $X_1 = \{a\}$ qui sont bien les deux modèles stables du programme $\mathcal{P}$.

Exemple 4.13. — Soit le programme logique $\mathcal{P}$:

$$\mathcal{P} = \{a \leftarrow not\ b, b \leftarrow not\ c, c \leftarrow not\ a\}$$

Son codage CNF est l'ensemble de clauses $L(\mathcal{P}) = Rules \cup ME$:

$$Rules = \{a \vee \neg not\ b, b \vee \neg not\ c, c \vee \neg not\ a\}$$

$$ME = \{\neg a \vee \neg not\ a, \neg b \vee \neg not\ b, \neg c \vee \neg not\ c\},$$

Son ensemble strong backdoor est $STB = \{not\ a, not\ b, not\ c\}$. La paire $(L(\mathcal{P}), STB)$ a trois extensions $E_1 = L(\mathcal{P}) \cup \{not\ a\}$, $E_2 = L(\mathcal{P}) \cup \{not\ b\}$ et $E_3 = L(\mathcal{P}) \cup \{not\ c\}$. Par résolution unitaire, on déduit que $E_1 \models \{\neg a, c, \neg not\ c, \neg not\ b\}$, $E_2 \models \{\neg b, a, \neg not\, a, \neg not\, c\}$ et $E_3 \models \{\neg c, b, \neg not\, b, \neg not\, a\}$. On voit que les trois extensions sont des extra-extensions qui ne vérifient pas la condition discriminante ($\forall A_i \in V, E \models \neg not\ A_i \Rightarrow E \models A_i$). Donc le théorème 4.8 dit que $\mathcal{P}$ n'a pas de modèles stables, ce qui est bien le cas.

Enfin nous reprenons le programme logique $\mathcal{P}$ de l'exemple 4.2 ayant pour unique modèle stable $X = \{p, q\}$. Dans la sémantique des extra-modèles, ce modèle correspond bien à l'extension $E_1 = L(\mathcal{P}) \cup \{not r\}$ qui satisfait bien la condition descriminante car $E_1 \models \{\neg not\ p, \neg not\ q, not\ r, p, q, \neg r\}$. On peut aussi remarquer que dans cette sémantique, ce programme logique admet une extra-extension $E_2 = L(\mathcal{P}) \cup \{not\, q\}$ telle que $E_2 \models \{\neg not\ p, not\ q, \neg not\ r, r, \neg q, \neg r\}$. E_2 ne vérifie pas la condition descriminate et infère l'extra-modèle $X' = \{r\}$ dans lequel l'atome r est vrai. La sémantique des extra-modèles règle en quelque sorte le problème d'intuition logique constaté dans la sémantique des modèles stable qui n'infère pas l'atome r. La sémantique des extra-modèles donne plus de sens à ce programme.

Le théoreme 4.8 montre que les modèles stables d'un programme logique $\mathcal{P}$ sont en bijection avec un sous ensemble des extensions de $(L(\mathcal{P}), STB)$. Les extra-extensions codant les extra-modèles ont été caractérisées par une condition simple à vérifier (la condition descriminante). D'autre part on a vu que tout programme logique consistant admet au moins une extension, donc un modèle. Cette propriété prouve que la définition d'extension donnée ici n'est pas une définition par point fixe. Il est donc possible de calculer les extensions de manière incrémentale : on ajoute progressivement des littéraux $not\, A_i$ du STB en vérifiant à chaque ajout la consistance de l'ensemble obtenu. On peut obtenir des algorithmes très simples de calcul d'extensions ou de modèles stables / extra-modèles. Si on ne s'intéresse pas aux extra-extensions, il suffit de les filtrer à la fin du calcul : dès qu'une extension a été trouvée on vérifie si elle infère un modèle stable en vérifiant la condition discriminante. On va donner dans la prochaine section une méthode de calcul d'extension et de modèles stables/extra-modèles basée sur la sémantique des extra-modèles.

4.3. Une méthode de recherche d'extensions / modèles et extra-modèles

On sait que tout modèle stable d'un programme $\mathcal{P}$ est un modèle de sa complétion $comp(\mathcal{P})$, mais que la réciproque est fausse dans le cas général . Fages [10] a montré que si un programme $\mathcal{P}$ est « tight » (sans boucle) alors l'ensemble des modèles stables est égal à l'ensemble des modèles de sa complétion $comp(\mathcal{P})$ [2]. Si la complétion d'un programme sans boucle est traduite par un ensemble de clauses Γ, alors on peut utiliser un solveur SAT Γ comme une boîte noire pour générer les modèles stables de $\mathcal{P}$. Lin et Zhao [20] ont montré que pour les programmes avec des boucles, les modèles de la complétion qui ne sont pas des modèles stables peuvent être éliminés en ajoutant à la complétion des *formules boucles*. Leur solveur ASSAT est basé sur cette technique et est plus performant que les solveurs ASP classiques tel que Smodels [24, 30] et

DLV [8] pour plusieurs instances. Néanmoins le solveur ASSAT a quelques défauts : il ne peut calculer qu'un seul modèle stable et la formule calculée peut exploser de manière combinatoire en espace. En prenant en compte le fait que chaque modèle stable d'un programme $\mathcal{P}$ est un modèle de sa complétion $comp(\mathcal{P})$, Guinchiglia et al. in [13] n'utilisent pas un solveur SAT comme boîte noire, mais implémentent une méthode basée sur la procédure DPLL [7] dans laquelle ils introduisent une fonction qui vérifie si un modèle généré est un modèle stable. Cette méthode a été implémentée dans le système Cmodels-2 [19] et a l'avantage de calculer $comp(\mathcal{P})$ sans introduire de variables supplémentaires, à part celles utilisées par la transformation en clauses de $comp(\mathcal{P})$. Elle fonctionne également pour les programmes avec boucles et sans boucles.

La méthode décrite ici est aussi énumérative, mais elle est différente de toutes les méthodes citées précédemment. Elle est basée sur la sémantique des extensions et extra-modèles décrite dans les sections précédentes. Pour un programme $\mathcal{P}$, la méthode travaille sur son encodage logique $L(\mathcal{P})$ et non pas sur sa complétion $comp(\mathcal{P})$. Si $\mathcal{P}$ est un programme général, $L(\mathcal{P})$ est un ensemble de clauses de Horn construit sur deux ensembles de variables propositionnelles V and nV. L'ensemble $STB \subset nV$ des variables qui apparaissent dans le corps des règles de $\mathcal{P}$ est le strong backdoor [31] sur lequel on effectue l'énumération pour obtenir les extensions de $(L(\mathcal{P}), STB)$ à partir desquelles on peut trouver les modèles stables et les extra-modèles de $\mathcal{P}$.

4.3.1. *Méthode*

À partir de ce qui précède on va donner ci dessous les étapes principales d'une nouvelle méthode de calcul des extensions / modèles stables et extra-modèles.

- Phase I : énumération
  (1) Soit $E = L(\mathcal{P})$, on affecte les variables du STB en utilisant une méthode d'énumération du type DPLL.
  (2) On vérifie à chaque affectation la consistance de l'ensemble clauses résultatnt. Comme on a un ensemble de clauses de Horn, une propagation par résolution unitaire assure la consistance.
  (3) On maintient également à chaque point de choix l'ensemble des littéraux $\{not\ A_i\}$ et $\{\neg not\ A_i\}$ impliquées par l'état courant. Les $\{not\ A_i\}$ sont générés par la propagation unitaire. Pour les $\{\neg not\ A_i\}$ une méthode très brutale est de faire k tests de consistance (k propagations unitaire) si k est le cardinal du STB (on peut être bien plus efficaces).
  (4) Pour la stratégie d'affectation des variables du STB à l'étape 1, une bonne approche est d'affecter en premier à Vrai les variables libres du STB. Puis on ne reviendra en arrière que sur les nœuds qui ont inféré au moins un nouveau littéral négatif du STB. Cette stratégie assure qu'à chaque point de choix consistant, en poursuivant l'énumération du STB, on obtiendra au moins une extension (la maximalité de l'extension est donc obtenue automatiquement).

- Pase II : complétion
  (1) Une fois l'affectation du STB effectuée totalement (on est sur une feuille de l'arbre de recherche), on obtient donc une interprétation partielle I_{STB} du STB qui étend E ($E = E \cup I_{STB}$)).
  (2) On affecte à Faux toutes les variables de E non encore affectées pour obtenir le modèle minimal M de E.
  (3) Si la condition ($\forall A_i \in V : E \models \neg not\ A_i \Rightarrow E \models A_i$) est vérifiée, la restriction aux littéraux positifs de E est alors un modèle stable de $\mathcal{P}$, sinon elle est un extra-modèle. Cette vérification est immédiate à partir du modèle minimal M obtenu.

L'algorithme est décrit ici de manière très générale, mais dans la pratique on peut le rendre bien plus efficace.

4.3.2. *Complexité*

Si n est le nombre de variables du codage $L(\mathcal{P})$ d'un programme $\mathcal{P}$, k est le cardinal du STB , m est le nombre de clauses de $L(\mathcal{P})$, l'étape 2 de l'algorithme peut être effectuée en $O((k/2)nm2^{k/2})$. Donc la complexité asymptotique dans le pire des cas est $O((k/2)nm2^{k/2})$ avec $k \leqslant n$. Cette estimation est très grossière et peut être vraiment améliorée. Dans le cas moyen la complexité est bien moindre. De plus, il est intéressant de remarquer que la première extension est trouvée à la première descente dans l'arbre sans retour en arrière, donc en $O((k/2)nm)$. Enfin la méthode a l'avantage de travailler sur un codage CNF $L(\mathcal{P})$ de $\mathcal{P}$ dont la taille est $taille(\mathcal{P}) + 2n$.

Cette méthode peut être utilisée pour des programmes avec boucles et sans boucles. Elle est capable de calculer tous les modèles stables d'un programme logique général $\mathcal{P}$. La méthode peut être mise en œuvre avec une modification mineure de la procédure DPLL. Ce travail est théorique mais tous les outils pour l'implémenter existent. Un premier solveur basée sur cette nouvelle sémantique a été conçu dans le cadre des travaux de thèse de Tarek Khaled [14] où il a été montré que cette approche est une très bonne alternative pour implémenter des solveurs ASP.

5. Conclusion

Prolog [5] a été un pionnier dans l'utilisation de la logique pour représenter des connaissances et raisonner sur celles-ci, permettant d'en tirer des conclusions valides. Nous avons présenté le fragment de la logique du Premier Ordre traitable et le mécanisme de résolution utilisé par Prolog, la SLD-résolution [4, 15, 21, 22], utilisant lui même le mécanisme de l'unification décrit précédemment par Robinson [27], et qui ont fait le succès de Prolog.

Nous avons ensuite présenté le mécanisme du *cut*, permettant d'élaguer des modèles, et sous-tendant une forme de négation, qui toutefois se distingue de la négation logique. Cette forme de négation, simulant un genre de négation par échec, va malheureusement sortir Prolog du champ de la logique, en rendant les programmes qui

l'utilisent sensibles à l'ordre d'écriture des règles qui les composent. Dès lors, tout un pan de la communauté IA va travailler à tenter de proposer une sémantique qui définisse proprement le sens de la négation par échec.

L'entreprise s'avérera plus compliquée que prévu, et au bout d'une vingtaine d'années, la sémantique des modèles stables a enfin permis d'obtenir une sémantique déclarative [12, 24], conduisant à un nouveau paradigme de programmation logique, ASP. Le succès d'ASP tient en une syntaxe expressive : utilisation de variables, contraintes d'intégrités, mais aussi règles de choix et règles de poids que nous n'avons pas abordées ici, traitées soit nativement [11, 30], soit en traduisant ces constructions en règles normales [19, 20]. Certains solveurs proposent aussi des directives d'optimisation permettant d'exprimer des préférences sur les modèles [11, 30]. Un autre élément de succès est l'existence de solveurs efficaces, qui ont bénéficié des progrès des solveurs SAT ces dernières années[11, 19, 20]. Si la plupart des solveurs s'appuient sur un processus en deux étapes visant dans une première étape à instancier le programme puis dans une seconde étape à faire appel au solveur pour le calcul des modèles stables, de nouveaux solveurs tels que GASP [6] et ASPeRiX [17, 18] réalisent l'instanciation au fur et à mesure de l'avancement du calcul des modèles. L'amélioration des performances des solveurs, en particulier en ce qui concerne le problème de l'énumération des modèles, reste toutefois un domaine de recherche actif, donnant lieu par exemple à de nouveaux schémas de codage des programmes logiques en un ensemble de clauses [14].

Toutefois, un angle mort persiste : c'est celui des programmes qui, considérés sous un angle logique, sont consistants, alors qu'ils n'admettent pas de modèle stable. Certaines sémantiques, comme la sémantique des extra-modèles présentée ici [1], répondent à ce problème. La sémantique des extra-modèles permet en outre de discriminer les modèles stables des extra-modèles. En conséquence, cette sémantique est à même de définir une approche alternative au calcul effectif des modèles stables[14].

Dans cet article, nous nous sommes attachés à présenter l'évolution de la programmation logique au travers de la problématique liée à l'introduction de la négation en Prolog, conduisant plus tard à l'ASP et à ses extensions, en particulier celles questionnant la notion de programme logique consistant n'admettant pas de modèle stable. En suivant ce chemin, il est bien évident que nous ne prétendions pas brosser un tableau d'ensemble de la programmation logique, qui est bien plus vaste que cela. Pour un tableau plus exhaustif, le lecteur pourra se référer au chapitre 4 de l'ouvrage « Panorama de l'intelligence artificielle, Volume 2 » [16].

Bibliographie

[1] B. Benhamou & P. Siegel, « A New Semantics for Logic Programs Capturing and Extending the Stable Model Semantics », in *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*, 2012, p. 572-579.

[2] K. L. Clark, « Negation as Failure », in *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977* (New York) (H. Gallaire & J. Minker, éds.), Advances in Data Base Theory, Plemum Press, 1977, p. 293-322.

[3] A. Colmerauer, « Metamorphosis Grammars », in *Natural Language Communication with Computers* (L. Bolc, éd.), Lecture Notes in Computer Science, vol. 63, Springer, 1978, p. 133-189.

[4] A. Colmerauer, H. Kanoui, P. Roussel & R. Pasero, « Un système de communication homme-machine en français », Tech. report, Groupe de recherche en Intelligence Artificielle, Université d'Aix-Marseille II, 1973.

[5] A. Colmerauer & P. Roussel, « The Birth of Prolog », p. 331–367, Association for Computing Machinery, New York, NY, USA, 1996.

[6] A. Dal Palù, A. Dovier, E. Pontelli & G. Rossi, « Answer Set Programming with Constraints Using Lazy Grounding », in *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings* (P. M. Hill & D. S. Warren, éds.), Lecture Notes in Computer Science, vol. 5649, Springer, 2009, p. 115-129.

[7] M. Davis, G. Logemann & D. Loveland, « A Machine Program for Theorem-Proving », *Commun. ACM* **5** (1962), nº 7, p. 394–397.

[8] T. Eiter, N. Leone, C. Mateis, G. Pfeifer & F. Scarcello, « The KR System dlv: Progress Report, Comparisons and Benchmarks », in *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), Trento, Italy, June2-5, 1998* (A. G. Cohn, L. K. Schubert & S. Shapiro, éds.), Morgan Kaufmann, 1998, p. 406-417.

[9] M. H. van Emden & R. A. Kowalski, « The Semantics of Predicate Logic as a Programming Language », *J. ACM* **23** (1976), nº 4, p. 733-742.

[10] F. Fages, « Consistency of Clark's completion and existence of stable models », *Thoery and Practice of Logic Programming* **1** (1994), p. 51-60.

[11] M. Gebser, B. Kaufmann & T. Schaub, « Conflict-driven answer set solving: From theory to practice », *Artif. Intell.* **187** (2012), p. 52-89.

[12] M. Gelfond & V. Lifschitz, « The stable model semantics for logic programming », in *Proceedings of International Logic Programming Conference and Symposium* (R. Kowalski & K. Bowen, éds.), MIT Press, 1988, p. 1070-1080.

[13] E. Giunchiglia, Y. Lierler & M. Maratea, « SAT-Based Answer Set Programming », in *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA* (D. L. McGuinness & G. Ferguson, éds.), AAAI Press / The MIT Press, 2004, p. 61-66.

[14] T. Khaled, B. Benhamou & P. Siegel, « A new method for computing stable models in logic programming », in *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, 2018, p. 800-807.

[15] R. A. Kowalski, « Predicate Logic as Programming Language », in *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974* (J. L. Rosenfeld, éd.), North-Holland, 1974, p. 569-574.

[16] A. Lallouet, Y. Moinard, P. Nicolas & I. Stéphan, « 4 : Programmation logique », in *Algorithmes pour l'Intelligence Artificielle* (P. Marquis, O. Papini & H. Prades, éds.), Panorama de l'Intelligence Artificielle, vol. 2, Cépaduès, 2014, p. 739-771.

[17] C. Lefèvre & P. Nicolas, « A First Order Forward Chaining Approach for Answer Set Computing », in *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings* (E. Erdem, F. Lin & T. Schaub, éds.), Lecture Notes in Computer Science, vol. 5753, Springer, 2009, p. 196-208.

[18] ———, « The First Version of a New ASP Solver: ASPeRiX », in *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings* (E. Erdem, F. Lin & T. Schaub, éds.), Lecture Notes in Computer Science, vol. 5753, Springer, 2009, p. 522-527.

[19] Y. Lierler & M. Maratea, « Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs », in *Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, January 6-8, 2004, Proceedings* (V. Lifschitz & I. Niemelä, éds.), Lecture Notes in Computer Science, vol. 2923, Springer, 2004, p. 346-350.

[20] F. Lin & Y. Zhao, « ASSAT: computing answer sets of a logic program by SAT solvers », *Artif. Intell.* **157** (2004), nº 1-2, p. 115-137.

[21] D. W. Loveland, « A linear format for resolution », in *Symposium on Automatic Demonstration* (Berlin, Heidelberg) (M. Laudet, D. Lacombe, L. Nolin & M. Schützenberger, éds.), Springer Berlin Heidelberg, 1970, p. 147-162.

[22] D. Luckham, « Refinement theorems in resolution theory », in *Symposium on Automatic Demonstration* (Berlin, Heidelberg) (M. Laudet, D. Lacombe, L. Nolin & M. Schützenberger, éds.), Springer Berlin Heidelberg, 1970, p. 163-190.

[23] J. McCarthy, « Programs with Common Sense », in *Proceedings of the Teddington Conference on the Mechanization of Thought Processes* (London), Her Majesty's Stationary Office, 1959, p. 75-91.

[24] I. Niemelä, « Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm », *Ann. Math. Artif. Intell.* **25** (1999), nº 3-4, p. 241-273.

[25] R. Reiter, « On Closed World Data Bases », in *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977* (H. Gallaire & J. Minker, éds.), Advances in Data Base Theory, Plemum Press, 1977, p. 55-76.

[26] ———, « A logic for default reasoning », *Artif. Intell.* **13** (1980), p. 81-132.

[27] J. A. Robinson, « A Machine-Oriented Logic Based on the Resolution Principle », *J. ACM* **12** (1965), nº 1, p. 23-41.

[28] C. Schwind & P. Siegel, « A Modal Logic for Hypothesis Theory », *Ann. Soc. Math. Pol., Ser. IV, Fundam. Inf.* **21** (1994), nº 1-2, p. 89-101.

[29] P. Siegel & C. Schwind, « Hypothesis theory for nonmonotonic reasoning », in *Workshop on Non-standard Queries and Answers, Toulouse, July 1991*, 1991.

[30] P. Simons, « Extending and implementing the stable model semantics », 2000, In doctoral dessertation, 305-316 pages.

[31] R. Williams, C. P. Gomes & B. Selman, « Backdoors To Typical Case Complexity », in *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003* (G. Gottlob & T. Walsh, éds.), Morgan Kaufmann, 2003, p. 1173-1178.

Abstract. — Using logic formulas in order to represent knowledge and reason about it is an idea dating back to the 60s, in particular to a proposition by McCarthy [23]. One of the axis of research stemming from this proposition leads to the genesis of Prolog, and to the Logic Programming paradigm [4]. An important evolution of Prolog was the incorporation of the *cut* mechanism, which subsequently allowed the development of *negation as failure*. Research aimed at establishing a declarative semantics of negation by failure led to the notion of *stable model*, and to a new paradigm of logic programming succeeding Prolog and called ASP. Subsequent work focused on programs which, while logically consistent, do not admit stable models. An example of such work is the *extra-model semantics*.

Keywords. — logic programing, prolog, stable models semantics.

Manuscrit reçu le 27 mai 2024, accepté le 12 juillet 2024.