Exemple 4.1.5 On reprend les programmes de l'exemple 4.1.4. L'ensemble des substitutions de réponse obtenu de toutes les réfutations de $P_1 \cup \forall x(\neg List(x))$ est

$$\{x \to nil\}, \{x \to cons(x_1, nil)\}, \{x \to cons(x_1, cons(x_2, nil))\}, \ldots$$

Exercice 111
Soit le programme P comme dans l'exemple 4.1.4. Donner l'ensemble des substitutions de réponse obtenu de toutes les réfutations de $P_2 \cup IncList(x)$.

4.1.4 SLD-résolution, et recherche de réfutations

Pour chercher une LD-réfutation comme dans la figure 4.1, il faut à chaque étape i faire deux choix :
- le choix d'un littéral $\neg P(\bar{s})$ dans L_i (à choisir parmi tous les littéraux de la clause L_i)
- le choix d'une clause de programme $P(\bar{t}) \leftarrow K$ (à choisir parmi toutes les clauses du programmes qui ont P à la tête).

Nous allons maintenant montrer que le premier choix peut être fait aléatoirement.

Definition 4.1.6 *Une* fonction de sélection *est une fonction qui, à chaque LD-dérivation $L_1, \ldots, L_n$ avec $L_n \neq \Box$, associe un littéral $\neg B \in L_n$.*

Soit P un programme logique, R une requête, et s une fonction de sélection. Une SLD-dérivation *à partir de $P \cup \{\neg R\}$ via s est une LD-dérivation $L_1, \ldots$ à partir de $P \cup \{\neg R\}$ où à chaque étape i l'atome $s(L_1, \ldots, L_i)$ est sélectionné.*

Une SLD-réfutation *via s est une* SLD-dérivation *via s qui se termine sur $\Box$.*

Le nom SLD-résolution vient de l'anglais *Selection rule-driven Linear resolution of Definite clauses.*

Théorème 4.1.3 *(Complétude réfutationelle de la résolution SLD pour les programmes logiques) Soit P un programme logique, $R = \tilde{\exists}(A_1 \wedge \ldots \wedge A_n)$ une requête et σ une substitution close. Les énoncés suivants sont équivalents :*

1. *$P \models (A_1 \wedge \ldots \wedge A_n)\sigma$*
2. *Il existe une LD-réfutation de $P \cup \{\neg R\}$ avec substitution de réponse θ et il existe une substitution close τ telles que $\sigma = \theta\tau$.*
3. *Pour toute fonction de sélection s il existe une SLD-réfutation via s de $P \cup \{\neg R\}$ avec substitution de réponse θ et il existe une substitution close τ telles que $\sigma = \theta\tau$.*

Les implications (1) $\Rightarrow$ (2) et de (3) $\Rightarrow$ (1) sont une conséquence du théorème 4.1.2 (puisque toute SLD-réfutation est aussi une LD-réfutation). Avant de démontrer l'implication (2) $\Rightarrow$ (3) nous avons besoin d'un petit lemme qui nous permettra de permuter l'ordre d'application des pas de résolution :

Lemme 4.1.3 *Soient*

- *$\bar{s}_1, \bar{s}_2, \bar{t}_1, \bar{t}_2$ des tuples de termes avec $Var(\bar{s}_1) \cap Var(\bar{t}_1, \bar{t}_2) = \emptyset$, $Var(\bar{s}_2) \cap Var(\bar{t}_1, \bar{t}_2) = \emptyset$, et $Var(\bar{s}1) \cap Var(\bar{s}_2) = \emptyset$.*
- *μ_1 un mgu de $\bar{s}_1 \stackrel{?}{=} \bar{t}_1$ et μ_2 un mgu de $\bar{s}_2 \stackrel{?}{=} \bar{t}_2$*
- *ν_2 un mgu de $\bar{s}_2 \stackrel{?}{=} \bar{t}_2\mu_1$, et ν_1 un mgu de $\bar{s}_1 \stackrel{?}{=} \bar{t}_1\mu_2$.*

Alors, $\nu_2 \circ \mu_1 = \nu_1 \circ \mu_2$ à renommage près.

Exercice 112
Démontrer le lemme 4.1.3.

Preuve:
(de la direction (2) $\Rightarrow$ (3) du théorème 4.1.3)

On montre d'abord qu'on peut, dans une LD-réfutation, commuter deux pas de résolution quand le littéral sélectionné dans le deuxième pas était déjà présent dans la clause de départ. Supposons que le programme contient deux clauses

$$\begin{array}{rcl} P_1(\bar{s}_1) & \leftarrow & K_1 \\ P_2(\bar{s}_2) & \leftarrow & K_2 \end{array}$$

et qu'il y a une réfutation

$$\begin{array}{l} \vdots \\ \leftarrow L_1, P_1(\bar{t}_1), L_2, P_2(\bar{t}_2), L_3 \\ (\leftarrow L_1, K_1, L_2, P_2(\bar{t}_2), L_3)\mu_1 \\ ((\leftarrow L_1, K_1, L_2, K_2, L_3)\mu_1)\nu_2 \\ \vdots \end{array}$$

$\bar{s}_1 \stackrel{?}{=} \bar{t}_1$ a pour mgu μ_1 et $\bar{s}_2 \stackrel{?}{=} \bar{t}_2\mu_1$ a pour mgu ν_2. Mais on suppose que $\mathrm{Var}(\bar{s}_2) \cap \mathrm{Var}(\bar{t}_2) = \emptyset$ et on a donc aussi $\bar{s}_2\mu_1\nu_2 = \bar{t}_2\mu_1\nu_2$. $\bar{s}_2$ et $\bar{t}_2$ sont donc unifiables. Soit μ_2 un mgu de $\bar{s}_2$ et $\bar{t}_2$. Il existe alors une substitution τ telle que $\mu_1\nu_2 = \mu_2\tau$. Alors, $\bar{s}_1\mu_1 = \bar{t}_1\mu_1$, donc $\bar{s}_1\mu_1\nu_2 = \bar{t}_1\mu_1\nu_2$ et donc $\bar{s}_1\tau = \bar{s}_1\mu_2\tau = \bar{t}_1\mu_2\tau$. Il en résulte que $\bar{s}_1 \stackrel{?}{=} \bar{t}_1\mu_2$ possède un mgu ν_1. D'après le lemme 4.1.3, on a alors $\mu_1\nu_2 = \mu_2\nu_1$ et on peut réarranger la réfutation en

$$\begin{array}{l} \vdots \\ \leftarrow L_1, P_1(\bar{t}_1), L_2, P_2(\bar{t}_2), L_3 \\ (\leftarrow L_1, P_1(\bar{t}_1), L_2, K_2, L_3)\mu_2 \\ ((\leftarrow L_1, K_1, L_2, K_2, L_3)\mu_2)\nu_1 \\ \vdots \end{array}$$

On peut ensuite continuer la réfutation comme dans la réfutation de départ. De plus, la longueur de la réfutation, et la substitution de réponse sont les mêmes.

Finalement, pour toute LD-réfutation $S = L_1, \ldots, L_n$ et fonction de sélection s soit $f_s(S) = n$ si S est une SLD-réfutation via s, et sinon

$$f_s(S) = min\{i \mid \text{le littéral sélectionné à l'étape } i \text{ n'est pas } s(L_1, \ldots, L_i)\}$$

On montre par récurrence sur $n - f_s(S)$ qu'on peut transformer toute LD-réfutation S en une SLD-réfutation via s et la même substitution de réponse.

Si $n - f_s(S) = 0$ alors S est déjà une SLD-réfutation.

Si $n-f_s(S) = m+1$: Soit $S = L_1, \ldots, L_n$. L'étape $n-(m+1)$ est la première étape où la sélection n'a pas été faite selon la fonction s. Puisque $L_n = \Box$, il y a une étape $j > n - (m + 1)$ où une instance de $s(L_1, \ldots, L_{n-(m+1)})$ est sélectionnée. En utilisant $j - (n - (m + 1))$-fois la commutation expliquée au-dessus, on obtient une LD-réfutation S' de la même longueur et avec la même substitution de réponse, et $f_s(S') \leq m$. $\Box$

Exercice 113
Montrer par un exemple, que les points du théorème ne sont pas équivalents à

4. Pour toute substitution close τ, $P \models (A_1 \wedge \ldots \wedge A_n)\theta\tau$

La définition d'une fonction de sélection donnée dans la définition 4.1.6 permet des critères de sélection qui rendent compte de toute l'histoire d'une dérivation. Par exemple, on peut définir une fonction de sélection qui choisit à chaque étape le littéral le plus « ancien » (s'il y en a plusieurs, on fais un choix selon un critère quelconque). Une telle fonction de sélection garantie une stratégie *équitable* (angl. : *fair*) : même dans une dérivation infinie, aucun littéral n'est persistant.

Les fonctions de sélection utilisées en pratique ne dépendent normalement pas de toute l'histoire d'une dérivation. Dans ce cas, on peut simplement écrire $s(\leftarrow B_1, \ldots, B_n)$ au lieu de $s(N_1, \ldots, N_m, s(\leftarrow B_1, \ldots, B_n))$.

- La fonction de sélection de PROLOG (voir Section 4.2) est $s_{Prolog}(\leftarrow B_1, \ldots, B_n) = B_1$. En conséquence, dans la construction d'une réfutation le but actuel peut être représenté comme une *pile* (évidemment, il faut toujours gérer les substitutions qui sont appliquées à toutes les littéraux dans le but). On peut obtenir une implantation efficace (*WAM : Warren Abstract Machine*), et un modèle simple pour le programmeur. En fait, le programmeur peut influencer l'ordre de sélection en choisissant l'ordre des littéraux dans les clauses du programme.
- ANDORRA PROLOG utilise une fonction de sélection plus intelligente : Un littéral $\neg P(\bar{t})$ est prioritaire pour la sélection s'il y a *exactement une* clause du programme avec tête $P(\bar{s}) \leftarrow L$ telle que le problème d'unification $\bar{s} \stackrel{?}{=} \bar{t}$ possède un unificateur. Dans ce cas là, il est certain que toute réfutation doit passer par la résolution avec cette clause. Remarquez que l'exécution de ce pas de résolution applique une substitution à toutes les littéraux restant dans le but. Il est donc possible qu'après un pas de résolution il y a des nouveaux littéraux prioritaires.

Étant donné un programme logique, une requête R et une fonction de sélection, on peut représenter toutes les SLD-dérivations à partir de $P \cup \{\neg R\}$ dans un arbre :

Definition 4.1.7 *Soit P un programme logique, R une requête, et s une fonction de sélection. Le* SLD-arbre *de $P \cup \{\neg R\}$ via s est un arbre (éventuellement infini) où*

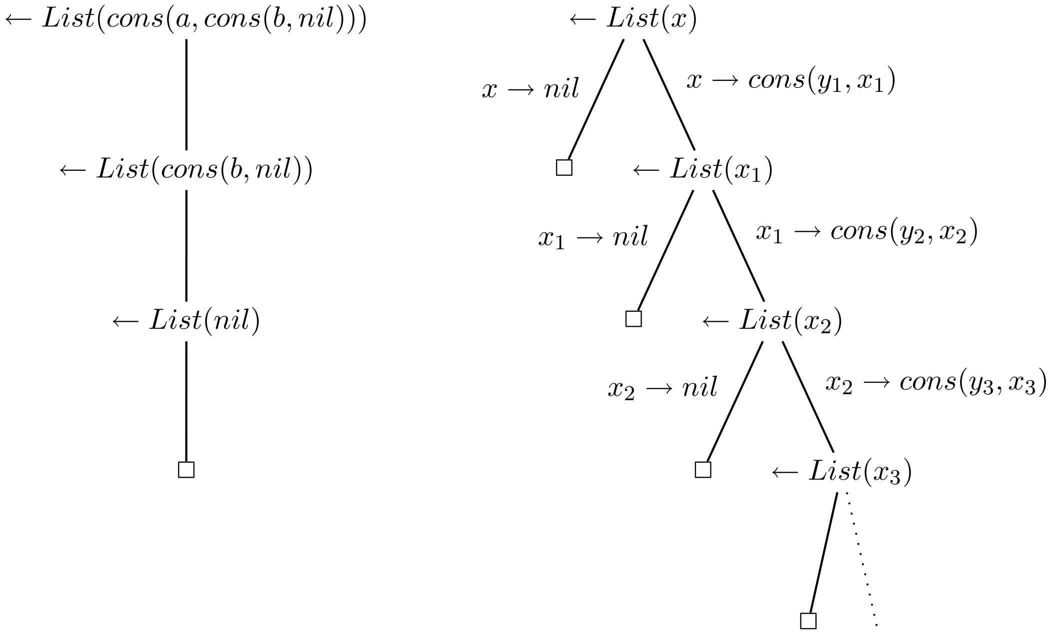

FIG. 4.2 – Les SLD-arbres pour les requêtes $List(cons(a, cons(b, nil)))$ et $\exists x List(x)$. On ne note que la partie de l'unificateur qui s'applique à des variables du but.

- *tout noeud est étiqueté par une clause de but,*
- *la racine est étiquetée par* $\neg R$*,*
- *si un noeud* n *est étiqueté par une clause but* L*, et si* $L_1, \ldots, L_m$ *sont toutes les clauses qu'on peut obtenir à partir de* L *par un pas de SLD-résolution via* s*, alors* n *a exactement* m *fils, qui sont étiquetés par* $L_1, \ldots, L_m$*.*

Un noeud de succès *d'un SLD-arbre est une feuille étiquetée par* $\Box$*. Un* noeud d'échec *d'un SLD-arbre est une feuille avec une étiquette différente de* $\Box$*. Un SLD-arbre est* réussi *s'il contient un noeud de succès.*

Donc, les chemins dans un SLD-arbre via s sont exactement les SLD-dérivations via s, et les chemins qui se terminent sur une feuille étiquetée par $\Box$ sont les SLD-réfutations via s.

Exemple 4.1.6 Reprennons le programme P_1 de l'exemple 4.1.4. Les SLD-arbres pour les requêtes $List(cons(a, cons(b, nil)))$ et $\exists x List(x)$ sont donnés dans figure 4.2. En fait, dans cet exemple la fonction de sélection n'a pas d'importance puisque aucun but n'a plus qu'un littéral.

Exercice 114
Soit le programme logique suivant :

$$\begin{aligned} Append(nil, x, x) &\leftarrow \\ Append(cons(h, x), y, cons(h, z)) &\leftarrow Append(x, y, z) \end{aligned}$$

Donner le SLD-arbre pour la requête $Append(x, y, cons(a, cons(b, nil)))$.

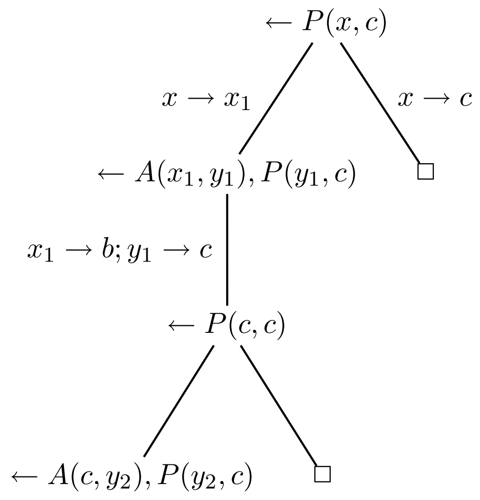

FIG. 4.3 – Un SLD-arbre via sélection du littéral le plus à gauche.

Le théorème 4.1.3 nous dit que deux SLD-arbres qui ne diffèrent que par la fonction de sélection utilisée contiennent essentiellement les mêmes LD-réfutations. Donc, pour toute branche du premier arbre qui se termine sur $\square$ on peut trouver une branche correspondante dans l'autre arbre, et inversement. Par contre, le théorème ne dit rien sur les branches infinies dans les SLD-arbres, ni sur les branches qui mènent vers un noeud d'échec. En fait, il est possible que pour un programme et une requête donné, le SLD-arbre via une fonction de sélection est fini, tandis que le SLD-arbre via une autre fonction de sélection est infini.

Exemple 4.1.7 Soit le programme logique suivant :

$$\begin{array}{rcl} P(x,z) & \leftarrow & A(x,y), P(y,z) \\ P(x,x) & \leftarrow & \\ A(b,c) & \leftarrow & \end{array}$$

Deux SLD-arbres pour la requête $\exists x P(x,c)$ via des fonctions de sélection différentes sont donnés sur les figures 4.3 et 4.4.

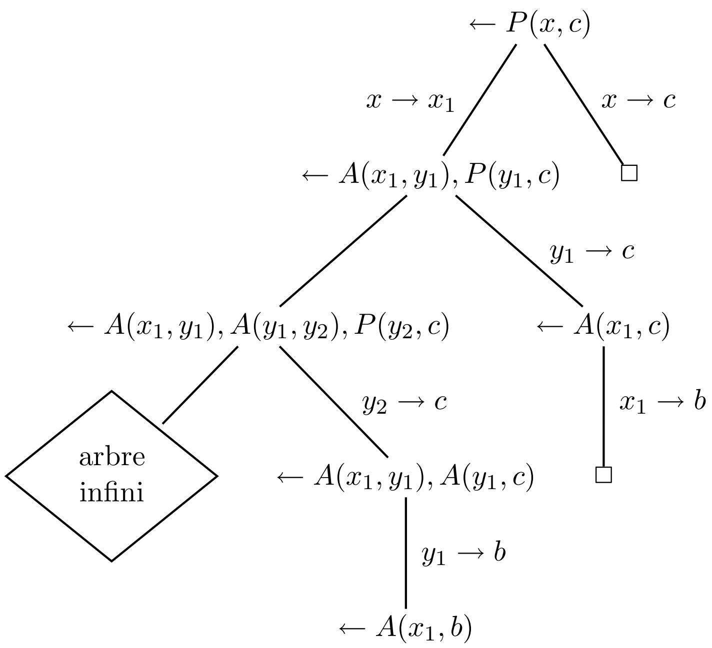

FIG. 4.4 – Un SLD-arbre via sélection du littéral le plus à droite.

4.2 Prolog

4.2.1 La recherche d'une réfutation en Prolog

Étant donné un programme logique P et une requête R, Prolog essaye de construire un SLD-arbre de réfutation de $P \cup \{\neg R\}$ via la fonction de sélection qui sélectionne dans un but le littéral le plus à gauche. Les clauses du programme sont utilisées dans l'ordre du programme. L'arbre est construit en profondeur d'abord (angl. : *depth first*). Donc, s'il y a une branche infinie dans le SLD-arbre de dérivation alors Prolog ne va jamais explorer les parties de l'arbre de dérivation qui sont « à droite » de cette branche infinie. Il est possible de demander à Prolog de calculer toutes les substitutions de réponse (aussi appelé *solution*), ou de s'arrêter après la première solution.

Pour un programme donné, le programmeur peut influencer la construction d'un SLD-arbre

1. en choisissant l'ordre des clauses dans le programme. L'ordre des clauses n'a pas de conséquence pour la structure du SLD-arbre même (à isomorphisme près). Donc, si on s'intéresse à toutes les substitutions de réponse alors l'ordre des clauses n'importe pas.

   Par contre, l'ordre des clauses est important pour l'ordre dans lequel les branches du SLD-arbre sont explorées. Puisque la stratégie de sélection de Prolog n'est pas équitable, un mauvais ordre des clause peut avoir le résultat qu'une certaine solution n'est jamais trouvée dans le cas d'un SLD-arbre infini.
2. en choisissant l'ordre des littéraux dans les clauses du programme et dans la requête. Ce choix peut avoir des conséquence importantes pour la structure du SLD-arbre, comme vu dans l'exemple 4.1.7 (on peut adapter cet exemple à la stratégie de sélection de Prolog en permuttant l'ordre des littéraux dans la première clause du programme).

Exercice 115
Soit le programme suivant :

$$\begin{array}{rcl} Permutation(x,x) & \leftarrow & List(x) \\ Permutation(x,y) & \leftarrow & ElemPerm(x,z),\ Permutation(z,y) \\[1ex] ElemPerm(cons(x,y),cons(x,z)) & \leftarrow & ElemPerm(y,z) \\ ElemPerm(cons(x,cons(y,z)),cons(y,cons(x,z))) & \leftarrow & \end{array}$$

Une requête $Permutation(t,s)$ est impliquée par ce programme ssi s est une liste (close ou pas), et t est une permutation de s.

Quelles sont les réponses trouvées par Prolog pour les requêtes suivantes :

1. $Permutation(cons(a,cons(b,cons(c,nil))),x)$
2. $Permutation(x,cons(a,cons(b,cons(c,nil))))$
3. $Permutation(cons(a,cons(b,cons(c,nil))),cons(c,cons(c,cons(c,nil))))$.

Exercice 116
Écrire un programme Prolog définissant un prédicat *select* tel que $select(t_1, t_2, t_3)$ est impliquée par le programme si t_3 est une liste, t_1 un élément de cette liste, et t_2 la liste t_3 privée de l'élément t_1. Prolog doit terminer sur une requête $select(s_1, s_2, s_3)$ quand s_2 ou s_3 est une liste.

Exercice 117
Écrire un nouveau programme Prolog pour *permutations* tel que Prolog termine sur les requêtes de l'exercice 115.

4.2.2 Arithmétique

Jusqu'ici, toutes les données dans des programmes Prolog étaient des termes. Il est en principe possible d'exprimer des autres types de donnée, comme par exemple les entiers, comme terme (voir l'exemple 4.1.3). Par contre, une telle représentation des entiers est très inefficace. C'est la raison pourquoi l'univers de Prolog contient aussi les entiers comme données.

De plus, il y a un nombre d'opérateurs comme $+$, $*$, $/$, etc. qui sont interprétées comme des fonctions sur les entiers, et des prédicats de comparaisons entre valeurs entières $==$, $\leq$, etc. Les expression entières sont partiellement intégrées dans le mécanisme d'unification. Un littéral de la forme $e_1 == e_2$ réussit si et seulement si toutes les variables dans les expressions e_1 et e_2 sont instanciées par des valeurs entières, et si les valeurs obtenues par évaluation des expressions e_1 et e_2 sont égales (analogue pour les autres opérateurs de comparaison $\leq$ etc.) De plus, un littéral $e_1 := e_2$ réussit si e_2 s'évalue vers un entier n, et

– soit e_1 s'évalue vers le même entier n
– soit e_2 est une variable x, et dans ce cas-là x est liée à la valeur n.

L'unification est donc très imparfaitement réalisée pour les expressions entières (ce problème a été résolu par la *programmation logique contrainte*).

Exercice 118
Écrire un programme définissant un prédicat *range* tel que, pour toute *valeur* entière non-négative n, la requête $range(x, n)$ termine avec la seule réponse

$$x \rightarrow cons(1, cons(2, \ldots, cons(n, nil) \ldots))$$

Un schéma de programmation très fréquent en Prolog (ou, plus généralement en programmation logique) est le *generate and test*. Il s'agit d'un schéma pour trouver une solution à un problème parmi les éléments d'un ensemble qui peut être énuméré comme les réponses d'une requête Prolog. Le schéma est simplement

$$\textit{generate-and-test(x)} \leftarrow generate(x), test(x).$$

Exercice 119
Écrire un programme définissant un prédicat *queens* tel que pour toute valeur entière positive n, les réponses obtenues pour la requête $queens(x, n)$ sont exactement les solutions du problèmes de n reines (le i-ème élément de la liste donne la colonne sur laquelle est placée la reine de la i-ème ligne).

Il se trouve qu'une réalisation naïve du schéma *generate and test* est souvent trop inefficace. En fait, on peut souvent « pousser le test dans le générateur », et couper un sous-arbre de l'arbre de recherche dès qu'on peut déterminer qu'une dérivation ne peut pas mener vers un succès. Dans l'exemple du problème des n reines, on peut échouer dès qu'il y a deux reines dans une configuration partielle qui se menacent.

Exercice 120
Modifier le programme obtenu dans l'exercice 119 tel qu'une solution partielle est rejetée dès qu'il y a deux reines qui se menacent.

Étant donné un prédicat $test(x)$, voici comment faire Prolog énumérer toutes les valeurs entières à partir d'une valeur de départ qui satisfont $test$:

$$\begin{array}{lcl} \textit{énumérer}(x, x) & \leftarrow & test(x) \\ \textit{énumérer}(x, y) & \leftarrow & z := y + 1, \textit{énumérer}(x, z) \end{array}$$

Il suffit de demander à la Prolog toutes les réponses à la requête $\textit{énumérer}(x, startvalue)$. On peut utiliser ce schéma pour obtenir toutes les solutions du problème des n reines pour toutes les valeurs de n (dans la syntaxe de GNU Prolog) :

La requête $allqueens(X, 1, N)$ va afficher toutes les réponses X pour toutes les valeurs de N.

4.2.3 Unification

Dans la plupart des des systèmes PROLOG, l'unification n'est pas implantée correctement : Pour des raisons d'efficacité, souvent le test de cycle (*occur check*) n'est pas effectué. Dans ce cas, la conséquence est que l'unification peut boucler. C'est notamment le cas avec GNU Prolog : Avec le programme

```
p(X,f(X)).
```

la requête `p(Y,Y)` fait boucler l'interpréteur GNU Prolog ! L'excuse est qu'un tel cas ne se produit pas en pratique.

4.2.4 Le *coupe-choix*

Le *coupe-choix* (angl. : *cut*), noté « ! », permet de couper une partie du SLD-arbre qui n'a pas encore été exploitée. Plus exactement, un littéral « ! » réussit toujours, et a l'effet que toutes les alternatives depuis l'introduction de ce coupe-choix sont coupées de l'arbre de recherche (voir Figure 4.5). Les effets d'un coupe-choix dans un programme Prolog ne sont pas toujours faciles à prévoir ; cette construction est donc à utiliser avec beaucoup de précaution. Une règle générale est que l'introduction d'un coupe-choix dans un programme ne doit pas changer la sémantique déclarative du programme.

Exemple 4.2.1 Soit le programme suivant pour calculer le minimum de deux entiers :

$$\begin{array}{lcl} minimum(x, y, x) & \leftarrow & x \leq y \\ minimum(x, y, y) & \leftarrow & x > y \end{array}$$

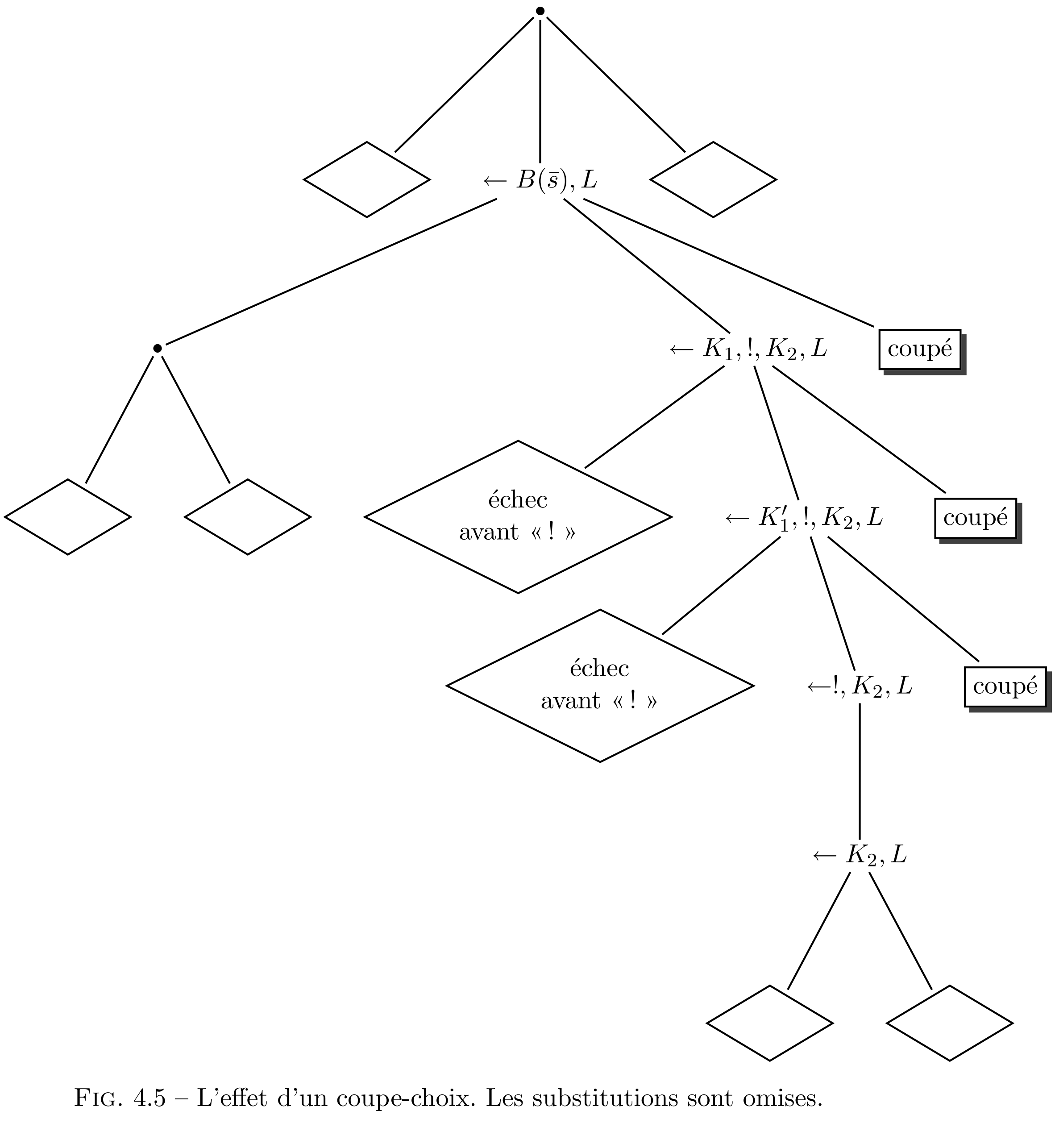

FIG. 4.5 – L'effet d'un coupe-choix. Les substitutions sont omises.

Avec un coupe-choix on peut éviter le test $x > y$ quand la première clause a réussi :

$$\begin{array}{rcl} minimum(x,y,x) & \leftarrow & x \leq y, ! \\ minimum(x,y,y) & \leftarrow & x > y \end{array}$$

Maintenant, on peut être tenté de supprimer le deuxième test, avec la justification que $x > y$ doit forcement être vrai quand la première clause a échoué :

$$\begin{array}{rcl} minimum(x,y,x) & \leftarrow & x \leq y, ! \\ minimum(x,y,y) & \leftarrow & \end{array}$$

Ça marche pour des requêtes de la forme $minimum(n_1, n_2, x)$, où n_1 et n_2 sont des valeurs entières, et x une variable. Le problème est que aussi la requête $minimum(1, 2, 2)$ réussit. La raison est qu'il y a en vérité deux tests qui peuvent échouer dans la première clause, le premier test (implicite) étant le test d'égalité entre le premier et le dernier argument. On peut donc « sauver » le programme en poussant le test d'égalité de la première clause derrière le coupe-choix :

$$\begin{array}{rcl} minimum(x,y,z) & \leftarrow & x \leq y, !, egal(x,z) \\ minimum(x,y,y) & \leftarrow & \\ egal(x,x) & \leftarrow & \end{array}$$

L'exemple 4.2.1 montre une utilisation du coupe-choix pour exprimer un algorithme déterministe. Une autre utilisation du coupe-choix est pour exprimer un non-déterminisme *don't care*, c'est-à-dire un non-déterminisme où le choix n'a pas de conséquence pour le résultat.

Exemple 4.2.2 Soit le programme suivant pour le tri à bulles :

$$\begin{array}{rcl} BubbleSort(x,x) & \leftarrow & SortedList(x) \\ BubbleSort(x,y) & \leftarrow & Append(z_1, cons(v, cons(w, z_2)), x), v > w, !, \\ & & Append(z_1, cons(w, cons(v, z_2)), y'), BubbleSort(y', z) \end{array}$$

(voir l'exercice 107 pour la définition de *SortedList*, et l'exercice 114 pour la définition de *Append*.) Dans cet exemple, le coupe-choix est justifié par le fait que le choix de la paire v, w avec $v > w$ n'importe pas.

4.2.5 Négation

Le coupe-choix permet d'exprimer une forme faible de la négation : Si L est une clause de but, alors $not(L)$ est considéré comme un littéral. Pour exécuter un tel littéral $not(L)$, Prolog essaye de construire le SLD-arbre pour L selon sa stratégie de sélection.

1. Si cet arbre est un arbre d'échec fini, alors $not(L)$ réussit. Aucune substitution n'est calculée pour contribuer à la réponse.
2. Si, pendant la construction de cet arbre, Prolog trouve un noeud de succès alors l'évaluation de $not(L)$ échoue.

3. Sinon l'évaluation de $not(x)$ ne termine pas.

Utilisant le fait qu'en Prolog les littéraux ne sont syntaxiquement pas différents des termes, le prédicat *not* peut être programmé en Prolog comme suit :

$$\begin{array}{rcl} not(x) & \leftarrow & x, !, fail \\ not(x) & \leftarrow & \end{array}$$

Ici, *fail* est un prédicat pour lequel le programme ne contient aucune clause, et qui donc échoue toujours.

Souvent, le prédicat *not* est utilisé avec des buts clos, c'est-à-dire dans une situation où la stratégie de Prolog nous garantie que toutes les variables du but sont complètement instanciées. Sinon, le fait que l'évaluation d'un *not* ne donne aucune substitution est un peu contre l'intuition. Il y a quand-même des cas où le *not* avec un but non-clos peut servir. Par exemple, le programme suivant défini un prédicat *disjoint* qui est vrai si et seulement si les deux listes sont disjointes :

$$\begin{array}{rcl} element(x, s) & \leftarrow & select(x, r, s) \\ disjoint(s_1, s_2) & \leftarrow & not(element(x, s_1), element(x, s_2)) \end{array}$$

Exercice 121
Donner un exemple de programme P et requête close R telle que $\neg R$ est dans le plus petit modèle de Herbrand, mais $not(R)$ n'est pas inféré. (Autrement dit, un témoin de l'incomplétude de la procédure décrite ci-dessus pour inférer la négation).

4.2.6 Méta-prédicats

Il y a en Prolog un nombre de prédicats, appelés des *méta-prédicats*, qui n'ont pas de dénotation logique. Par exemple, le prédicat $var(t)$ réussit quand t est un terme qui est une variable. Remarquez que, avec ce prédicat, il n'est plus possible de commuter les pas de résolution comme dans la preuve du théorème 4.1.3. En général, les méta-prédicats ne peuvent être compris que dans un sens purement opérationnel.

Exemple 4.2.3 [O'Keefe] Soit le programme suivant (*bidon* est une constante, x,y sont des variables).

$$\begin{array}{rcl} same\_var(bidon, y) & \leftarrow & var(y), !, fail \\ same\_var(x, y) & \leftarrow & var(x), var(y) \end{array}$$

Exercice 122
Montrer pour le programme de l'exemple 4.2.3 : Pour tous les termes t_1 et t_2, le but $\leftarrow same\_var(t_1, t_2)$ réussit si et seulement si t_1 et t_2 sont syntaxiquement la même variable.