UNIFICATION : ALGORITHMES ET APPLICATIONS

1. L'unification

1.1. **Termes, substitutions, unification.** On se donne un ensemble $\mathcal{L}$ de symboles de fonctions d'arité donnée et un ensemble infini de variables $\mathcal{V}$.

Définition 1. L'ensemble $\mathcal{T}$ des termes sur $\mathcal{L}$ est défini inductivement par : :

- $\mathcal{T}$ contient les variables ;
- si f est un symbole de fonction d'arité k et si $t_1, ..., t_k \in \mathcal{T}$, alors $f(t_1, ..., t_k) \in \mathcal{T}$.

Si t est un terme, on note $t[x_1, ..., x_n]$ si seules les variables $x_1, ..., x_n$ apparaissent dans t.

Définition 2. A tout terme correspond un unique arbre de dérivation. On appelle *position* dans un terme un noeud de cet arbre.

Définition 3. Une *substitution* est une fonction d'un ensemble fini de variables dans $\mathcal{T}$. Si σ est une substitution de domaine $x_1, ..., x_n$ et si $t[x_1, ..., x_n]$ est un terme, on note $t[\sigma]$ ou bien $t[x_1 := \theta_1, ..., x_n := \theta_n]$ où $\theta_i = \sigma(x_i)$ la substitution obtenue à partir de t en remplaçant chaque occurence de x_i par le terme θ_i. Si E est un ensemble de termes, on note $E[\sigma]$ l'ensemble $\{t[\sigma], t \in E\}$.

Définition 4. Un terme s *filtre* le terme t si il existe une substitution σ telle que $s[\sigma] = t$.

Exemple 5. On cherche à reconnaître un motif dans un langage fonctionnel comme Caml, considérons l'exemple d'une fonction qui a pour motif $h :: t$. On lui passe en paramètre une liste, par exemple $1 :: (2 :: 3)$. Alors la subsitution σ donnée par $\sigma(h) = 1$,$\sigma(t) = 2 :: 3$ montre que $h :: t$ filtre $1 :: (2 :: 3)$.

Remarque 6. L'exemple précédent est en fait un cas plus simple que le cas général qu'on a présenté, car on interdit en Caml aux variables d'apparaître plusieurs fois dans un motif.

Définition 7. Un *unificateur* de deux termes s et t est une substitution σ telle que $s[\sigma] = t[\sigma]$. Deux termes sont dits *unifiables* s'il existe un unificateur de ces deux termes.

Exemple 8. $s = f(x, x, y)$ et $t = f(f(y, y, z), f(y, y, z), a)$ sont unifiables par $\sigma(x) = f(a, a, z)$ et $\sigma(y) = a$

Théorème 9. *Soient s et t deux termes unifiables. Il existe alors un unificateur σ de s et t vérifiant que pour tout unificateur σ' de s et t, il existe une substitution σ'' telle que $\sigma' = \sigma'' \circ \sigma$. Cet unificateur est appelé* unificateur le plus général de s et t *et est noté $mgu(s, t)$.*

Proposition 10. *Deux unificateurs les plus généraux sont égaux à renommage des variables près, ce qui justifie la notation ci-dessus.*

1.2. **L'algorithme naïf d'unification.** On va décrire ici quelques règles qui, appliquées tant que possible vont permettre :

- De déterminer si deux termes sont unifiables ;
- Le cas échéant, calculer leur unificateur le plus général.

Les règles portent sur des ensembles finis d'équations

$$\{s_1 \sim t_1, ..., s_n \sim t_n\},$$

l'idée étant de partir de l'ensemble $\{s \sim t\}$ où s et t sont les termes à unifier et d'appliquer successivement ces règles. Les règles sont les suivantes :

- $E \sqcup \{f(s_1, ..., s_n) \sim f(t_1, ..., t_n)\} \to E \sqcup \{s_i \sim t_i, i = 1, ..., n\}$
- $E \sqcup \{f(s_1, ..., s_n) \sim g(t_1, ..., t_m)\} \to \mathsf{clash}$ pour $f \neq g$
- $E \sqcup \{x \sim x\} \to E$ pour $x \in \mathcal{V}$
- $E \sqcup \{x \sim t\} \to \mathsf{occurs-check}$ si x apparaît dans t
- $E \sqcup \{x \sim t\} \to E[\sigma := t]$ si x n'apparaît pas dans t

Théorème 11. *Si l'application successive de ces règles à l'ensemble $\{s \sim t\}$ termine*

- *par clash ou occurs-check, alors s et t ne sont pas unifiables*
- *par $\emptyset$, alors s et t sont unifiables.*

Théorème 12. *L'application successive de ces règles depuis $\{s \sim t\}$ dans un ordre quelconque termine toujours. De plus, la subsitution obtenue en composant successivement les substitutions obtenues lors de l'application de la dernière règle donne l'unificateur le plus général de s et t.*

Exemple 13. Dans l'exemple ci-dessus, l'application des règles nous montre que l'unificateur trouvé est bien le plus général.

Ce théorème nous donne donc un algorithme de calcul de l'unificateur le plus général de deux termes. Cet algorithme est cependant exponentiel, à la fois en temps et en espace.

Remarque 14. L'algorithme présenté est non déterministe : il se peut que plusieurs règles soient applicables. La terminaison ne dépend pas de l'ordre choisi. En général, on choisit de traiter les cas d'échec en premier.

2. Applications

2.1. **La méthode de résolution.** La méthode de résolution est un système de déduction, particulièrement adapté à la démonstration automatique. Il est défini à l'aide de l'unification.

On considère $\mathcal{L}$ un langage du premier ordre.

Définition 15. Un *littéral* est une formule atomique sur $\mathcal{L}$ ou bien sa négation. Si L est un littéral, si A est une formule atomique, on note $\bar{A} = \neg A$ et $\overline{\neg A} = A$.

Définition 16. Une *clause* est un ensemble de littéraux. Une clause représente la disjonction de ses éléments. Un ensemble de clauses représente la conjonction de la clôture universelle de chacune de ses clauses. On note $\perp$ la clause vide.

La méthode de résolution est donnée par deux règles :
La résolution :

$$\frac{C_1, L_1\ C_2, L_2\ \sigma = mgu(L_1, \overline{L_2})}{C_1[\sigma], C_2[\sigma]} res$$

La contraction :

$$\frac{C_1, L_1, L_2\ \sigma = mgu(L_1, L_2)}{C_1[\sigma], L_1[\sigma]} contr$$

Remarque 17. On peut interpréter $\neg$ et les prédicats R comme des symboles de fonctions, ce qui permet de généraliser l'unification à des littéraux et non seulement à des termes.

Théorème 18. *On peut déduire $\perp$ à partir d'un ensemble de clauses E et en utilisant les deux règles res et $contr$ si et seulement si E n'admet pas de modèle.*

Remarque 19. L'utilisation de ce système de déduction pour des formules clauses suppose la mise sous forme clausale d'une formule, ce qui peut être fait à l'aide de la Skolémisation par exemple.

Dans le cas où on se restreint à des clauses bien spécifiques où n'apparaît qu'au plus un littéral positif, on utilise une seule règle :

Définition 20. Une *clause de Horn* est une clause C qui contient au plus un littéral positif
- Si $C = \neg A_1, \neg A_2, ..., \neg A_m, A$, C est une *clause définie* notée $A \leftarrow A_1, ..., A_m$
- Si $C = \neg A_1, ..., \neg A_m$, C est une *clause négative.*

Un *programme* est un ensemble de clauses définies.
La règle utilisée est la suivante, dite règle de résolution SLD :

$$\frac{A \leftarrow A_1, ..., A_m\ \neg B_1, ..., \neg B_n\ \sigma = mgu(A, B_i)}{(\neg B_1, ..., \neg B_{i-1}, \neg A_1, ..., \neg A_m, \neg B_{i+1}, ..., \neg B_n)[\sigma]} SLD$$

Théorème 21. *Ce système de déduction est correct et complet pour la réfutation.*

On utilise ce système de déduction en Prolog : étant donné un but, et un ensemble de clauses définies, Prolog cherche à construire une réfutation SLD.

DEV : exemple de programme Prolog.

2.2. **Réécriture : paires critiques.** L'unification permet de montrer que le problème de la confluence d'un système de réécriture fini et terminant est décidable. On considère dans cette partie un système de réécriture fini et terminant $\rightarrow$.

On rappelle le lemme de Newman, qui va être à la base de notre démonstration :

Théorème 22. *$\rightarrow$ est confluent si et seulement si il est localement confluent.*

Définition 23. Soient $l_1 \rightarrow r_1$ et $l_2 \rightarrow r_2$ deux règles. On suppose qu'on a renommé les variables de telle sorte que les variables de l_1, r_1 n'apparaissent pas dans l_2, r_2. Soit $[l_1]_p$ un sous-terme de l_1 apparaissant en position p qui ne soit pas une variable et soit σ un mgu de $[l_1]_p$ et l_2. Alors on appelle *paire critique* la paire $(r_1[\sigma], l_1[\sigma]')$, où $l_1[\sigma]'$ est obtenu à partir de $l_1[\sigma]$ en remplaçant le sous-arbre en position p par $r_2[\sigma]$.

Définition 24. Une paire (a, b) est dite joignable si il existe x tel que $a \to^* x$ et $b \to^* x$.

Théorème 25. *R est localement confluent si et seulement si toutes les paires critiques sont joignables.* ***DEV2***

Corollaire 26. *R est confluent si et seulement si toutes les paires critiques sont joignables.*

Corollaire 27. *Le problème de savoir si un système de réécriture fini et terminant est confluent est décidable.*

Remarque 28. Dans le cas général où on ne suppose pas R fini et terminant, ce problème est indécidable.

2.3. **Inférence de types.** On considère un langage fonctionnel très simple appelé PCF, ayant une syntaxe proche de celle de Caml. Ce langage est défini à l'aide de la grammaire

$$\begin{aligned} T \quad = \quad & x | \mathsf{fun}\, x \to T | T\, T \\ & | 1, 2, ... | T + T | T * T \\ & | T - T | T / T \\ & | \mathsf{ifz}\, T\, \mathsf{then}\, T\, \mathsf{else}\, T \\ & | \mathsf{fix}\, x\, T | \mathsf{let}\, x = T\, \mathsf{in}\, T \end{aligned}$$

La plupart de ces constructions sont naturelles lorsqu'on est familier avec un langage comme Caml. ifz permet de tester l'égalité à 0 et $\mathsf{fix}\, x\, T$ est le plus petit point fixe de la fonction $\mathsf{fun}\, x \to T$: cela permet des définitions récursives.

Exemple 29. La fonction factorielle peut être définie en PCF par

$$\mathsf{fix}\, f\, \mathsf{fun}\, x \to \mathsf{ifz}\, n\, \mathsf{then}\, 1\, \mathsf{else}\, n * f\, (n-1).$$

Pour s'assurer que les fonctions produites ont bien un sens, il faut attribuer des types aux variables et aux fonctions.

Définition 30. Les *types* sont des termes sur le langage $\{\to, \mathsf{int}\}$ où $\to$ est d'arité 2 en notation infixe et int est un symbole de constante.

Exemple 31. $(\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$ est un type.

On aimerait maintenant déterminer si une expression est correctement typée, et le cas échéant trouver son type. Pour chaque construction du langage PCF, on a alors des contraintes sur les types.

Par exemple, pour construire $V\, U$ il faut que U soit de type A et V de type $A \to B$. Le type de $V\, U$ est alors $A \to B$.

On veut de plus que le type de notre expression soit le plus général possible : par exemple, $\mathsf{fun}\, x \to x$ peut être de type $\mathsf{int} \to \mathsf{int}$ mais est en réalité de type $A \to A$ où A est une variable de type.

Nous allons maintenant montrer qu'on peut se ramener à un problème d'unification.

On parcourt l'arbre de dérivation de notre expression, et :

- Lorsqu'on type un noeud $\mathsf{fun}\, x \to t$ on ajoute une variable X associée à x puis on type t ;
- Lorsqu'on type une expression $t\, u$, on calcule les types de t et u et on ajoute une équation $T = U \to X$ où X est une variable associée à $t\, u$
- Lorsqu'on type $u + t$ (ou autres opérateurs) on calcule les types de t et u puis on pose les équations $U = \mathsf{int}$ et $V = \mathsf{int}$
- Une constante est de type int
- etc.

On obtient alors un ensemble d'équation dont on calcule un unificateur. Cet unificateur nous donne le type de l'expression.

Exemple 32. Montrons le fonctionnement de l'inférence de type sur un exemple :

Pour type une expression comme $\mathsf{fun}\, f \to 2 + f\, 1$, on pose $f : F$ et on doit typer l'expression $2 + f\, 1$ pour cela on doit d'abord typer 2 qui est de type int puis $f\, 1$. Pour typer $f\, 1$ on calcule le type de f qui est F, le type de 1 qui est int et on ajoute l'équation $X = \mathsf{int} \to Y$ où Y est associée à $f\, 1$. On ajoute ensuite les équations $\mathsf{int} = \mathsf{int}$ et $Y = \mathsf{int}$.

Ces équations ont pour unificateur le plus général la substitution $X := \mathsf{int}$ et $Y := \mathsf{int}$, donc notre expression est de type $(\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$.

Références

[1] Baader. *Term rewriting and all that.*

[2] G. Dowek. *Introduction à la théorie des langages de programmation.*

[3] K. Nour. *Introduction à la logique.*

Exemples de programmes PROLOG

Clause définie : $A \leftarrow A_1 \wedge \dots A_n$ (règle) ou $A \leftarrow$ (fait)

Clause négative : $B_1, \dots B_n$ $(\equiv \neg B_1 \vee \dots \vee \neg B_n)$ (but)

Effacement des buts

$$\frac{C = A \leftarrow A_1, \dots A_n \in P \text{ (ou une variante)} \qquad \sigma \in UP(A, B_i)}{B_1, \dots, B_k \xrightarrow{i,\sigma,C} (B_1, \dots B_{i-1}, A_1, \dots A_n, B_{i+1}, \dots B_n)}$$

Objectif : construire une preuve par réfutation de $B_1, \dots B_k$
càd effacer tous les buts : $B_1, \dots B_k \rightsquigarrow^{*,\sigma} \square$.

On peut donc se restreindre à $i = 1$ (toujours effacer le but le plus à gauche).

Arbre de résolution SLD : arbre de toutes les possibilités d'effacement.

Ex 1

$C_1 : p(0).$

$C_2 : p(s(s(x))) \leftarrow p(0).$

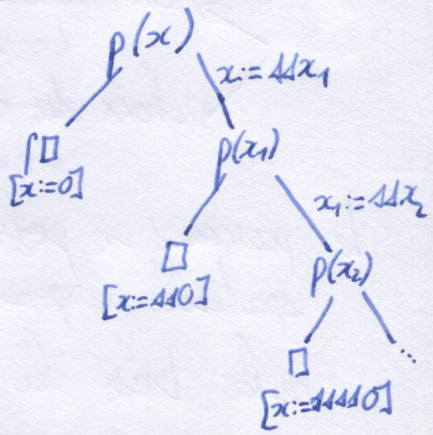
- **but réfutable**

$p(s(s(0)))$
| C_2
$p(0)$
| C_1
$\square$
renvoie [] (id)

- **but non réfutable**

$p(s(s(s(0))))$
| C_2
$p(s(0))$
X
échec car $p(s(0))$ n'est unifiable avec aucun des membres gauche.

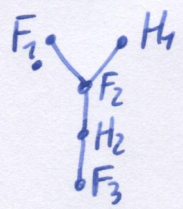
- **énumérer les solutions**



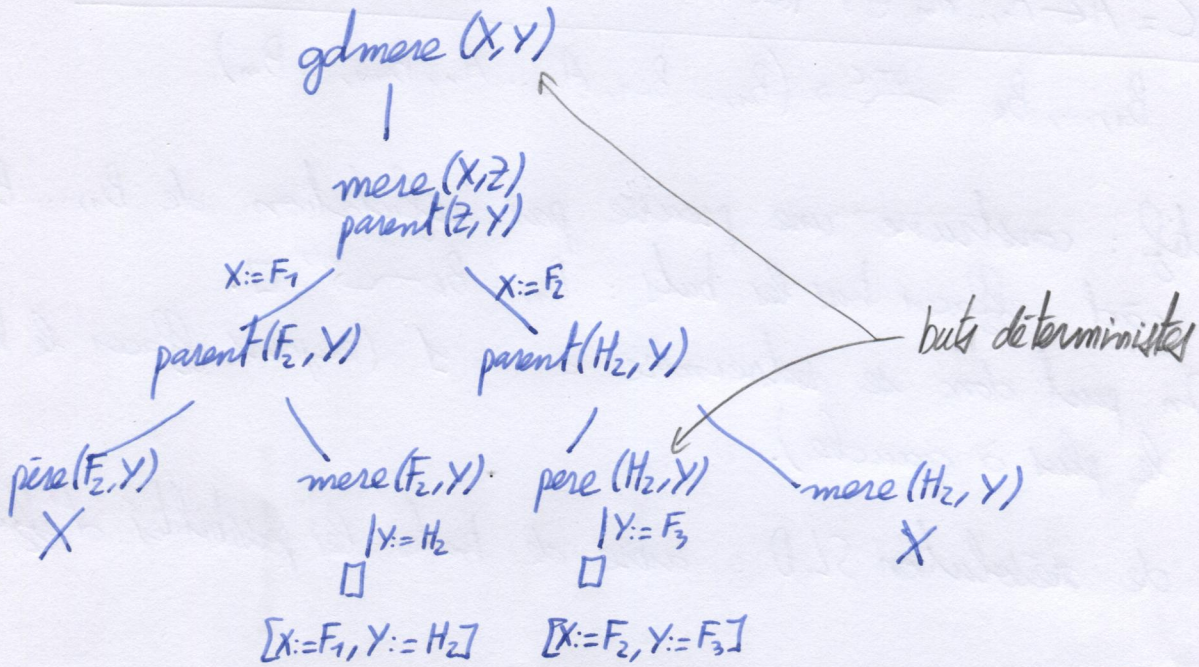
↳ c'est en cela que Prolog calcule

3 type de branches : succès, échec, infinie

Ex 2 arbre généalogique



mere (F_1, F_2)
pere (H_1, F_2)
mere (F_2, H_2)
pere (H_2, F_3)
parent $(X,Y) \leftarrow$ pere (X,Y)
parent $(X,Y) \leftarrow$ mere (X,Y)
gdmere $(X,Y) \leftarrow$ mere (X,Z), parent (Z,Y)



→ on a bien toutes les solutions.

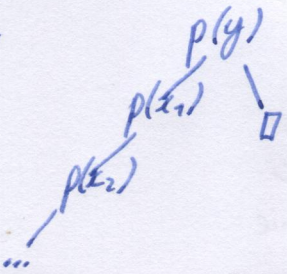
Non déterminisme

- choix du B_i (règle de sélection : forme de l'arbre)
  ↳ i = 1
- choix du noeud et de C (règle de recherche : parcours de l'arbre)
  ↳ en profondeur d'abord, et de G à D (ie. C dans l'ordre du pgm)

le parcours en profondeur d'abord permet d'éviter la complexité spatiale exponentielle d'un parcours en profondeur. Mais cela brise la complétude !

Ex 3

$p(s(x)) \leftarrow p(x)$.
$p(0)$.



l'interpréteur boucle avant de trouver □ ...