Spike

Sujet proposé par Jean-Jacques Lévy

jean-jacques.levy@inria.fr

1 Préambule

Il s'agit de réaliser un petit démonstrateur automatique d'équations vérifiées par des programmes.

2 Détail du sujet

Dans la démonstration automatique et plus particulièrement dans les théories équationnelles, on a souvent besoin de mettre des termes sous forme canonique. Prenons, par exemple, le cas bien étudié et simple de la théorie des groupes libres définie par les trois équations:

$$\begin{aligned} (x+y)+z &= x+(y+z) \\ 0+x &= x \\ (-x)+x &= 0 \end{aligned}$$

Knuth et Bendix ont montré que cette théorie peut se réduire aux 10 *réécritures* suivantes. Deux termes sont égaux si et seulement si leur forme canonique (encore appelée *forme normale*) est la même dans le système suivant:

$$\begin{aligned} (x+y)+z &\to x+(y+z) && (1) \\ 0+x &\to x && (2) \\ (-x)+x &\to 0 && (3) \\ (-x)+(x+y) &\to y && (4) \\ -0 &\to 0 && (5) \\ x+0 &\to x && (6) \\ -(-x) &\to x && (7) \\ x+(-x) &\to 0 && (8) \\ x+((-x)+y) &\to y && (9) \\ -(x+y) &\to (-y)+(-x) && (10) \end{aligned}$$

Ainsi, les deux termes $-(x+y)+x$ et $x+-(y+x)$ sont égaux puisque leur formes normales sont les mêmes. En effet: $-(x+y)+x \to (-y+-x)+x \to -y+(-x+x) \to -y+0 \to -y$ par les règles 10, 1, 3, 6 et: $x+-(y+x) \to x+(-x+-y) \to -y$ par 10, 9. A tout moment, plusieurs peuvent s'appliquer sur un terme ou ses sous-termes. Mais dans cet exemple, la forme normale ne dépend pas de l'ordre dans lequel on applique les règles (propriété dite de *confluence*, ou Church-Rosser).

Une règle de réécriture revient à orienter une équation, c'est-à dire à l'utiliser toujours dans le même sens pour obtenir une forme normale. Mais certaines équations ne peuvent être orientées. Par exemple, dans un groupe commutatif, l'équation $x + y = y + x$ n'est pas orientable. On fait alors un traitement spécial avec des théories associatives et commutatives, que nous ne considérerons pas ici. Donc pour toute équation on doit se demander comment l'orienter pour obtenir un système de réécritures produisant des formes normales (propriété de *normalisation*, ou de *normalisation forte* si en outre *tous* les calculs sont finis). Ce problème est en général indécidable, mais il existe de bonnes approximations avec des ordres polynomiaux ou avec des "rpo" (*recursive path orderings* dus à Plaisted et Dershowitz). En première approximation, on ignorera le problème de l'orientation des règles et on le fera au hasard ou manuellement.

Comment donc trouver automatiquement les 10 règles de réécriture permettant de normaliser la théorie des groupes? On voit qu'il faut garder les 3 axiomes initiaux en les orientant, mais les compléter par 7 autres. La méthode de Knuth-Bendix consiste à considérer tous les recouvrements possibles entre les règles et à rajouter les règles nécessaires pour rendre le système confluent. Ainsi les règles 1 et 1 peuvent se recouvrir puisqu'on a:

$((x+y)+z)+t \to (x+y)+(z+t) \to x+(y+(z+t))$

$((x+y)+z)+t \to (x+(y+z))+t \to x+((y+z)+t) \to x+(y+(z+t))$

De même pour les règles 1 et 2

$(0+x)+y \to x+y$

$(0+x)+y \to 0+(x+y) \to x+y$

Mais le recouvrement entre 1 et 3 est plus problématique

$((-x)+x)+y \to (-x)+(x+y)$

$((-x)+x)+y \to 0+y \to y$

Pour confluer, on doit rajouter la règle 4. On doit à présent aussi considérer tous les recouvrements entre 1, 2, 3 et 4. Entre 1 et 4, on a:

$((-x)+(x+y))+z \to (-x)+((x+y)+z) \to (-x)+(x+(y+z)) \to y+z$

$((-x)+(x+y))+z \to y+z$

Entre 2 et 4, on a:

$(-0)+(0+x) \to x$

$(-0)+(0+x) \to (-0)+x$

Pour confluer, on doit rajouter $(-0)+x \to x$ (règle 5'). Etc. (Plus tard, cette règle deviendra inutile lorsque la règle 5 sera ajoutée).

La complétion de Knuth-Bendix s'arrête pour les groupes libres, mais peut diverger pour d'autres théories équationnelle. Il existe donc 2 causes de divergence: l'orientation des règles et la complétion elle-même. Toute une littérature existe sur cette méthode, on peut commencer par http://www-sal.cs.uiuc.edu/~nachum/papers/-README.html#general. Nous fournirons la littérature nécessaire.

Pour trouver tous les recouvrements, toutes les *paires critiques* entre règles se trouvent en cherchant les parties gauches de règles qui sont *unifiables* avec un sous-terme non trivail de tout membre gauche. Par exemple, $0 + x$ s'unifie avec un sous-terme de $(x' + y') + z'$, dans le terme $(0 + x) + y$. L'*unification* de deux termes est une opération classique dans le calcul symbolique: deux termes sont unifiables ssi il existe une substitution de leurs variables produisant le même terme. C'est par exemple l'opération de base de Prolog (cf http://pauillac.inria.fr/~levy/pi97/).

On peut aussi utiliser la complétion de Knuth-Bendix pour montrer des propriétés équationnelles sur les programmes. C'est un corrolaire immédiat, trouvé par Huet et Hullot, et développé par Bouhoula dans son système Spike (cf. *http://www.loria.fr/~bouhoula/spike.html*)

Prenons par exemple la définition de *append* sur les listes donnée par les équations:

$app(nil, x) = x$
$app(cons(h, x), y) = cons(h, app(x, y))$
Essayons de montrer l'associativité de *append*.
$app(app(x, y), z) = app(x, app(y, z))$
Par la méthode Knuth-Bendix, on voit que
$app(nil, x) \to x$
$app(cons(h, x), y) \to cons(h, app(x, y))$
$app(app(x, y), z) \to app(x, app(y, z))$
est confluent et complet (au sens de Knuth-Bendix). Donc la propriété est vraie!

Considérons maintenant toutes les définitions suivantes:
$x + 0 = x$
$x + s(y) = s(x + y)$

$x * 0 = 0$
$x * s(y) = (x * y) + x$

$dbl(0) = 0$
$dbl(s(n)) = s(s(dbl(n)))$

$half(0) = 0$
$half(s(0)) = 0$
$half(s(s(n))) = s(half(n))$

$even(0) = true$
$even(s(0)) = false$
$even(s(s(n))) = even(n)$

$evenm(0) = true$
$oddm(0) = false$
$evenm(s(n)) = oddm(n)$
$oddm(s(n)) = evenm(n)$

$len(nil) = 0$
$len(cons(h, t)) = s(len(t))$

$app(nil, a) = a$
$app(cons(h, a), t) = cons(h, app(a, t))$

$nth(0, l) = l$
$nth(x, nil) = nil$
$nth(s(x), cons(h, t)) = nth(x, t)$

$rot(0, l) = l$
$rot(n, nil) = nil$
$rot(s(n), cons(h, t)) = rot(n, app(t, cons(h, nil)))$

$rev(nil) = nil$
$rev(cons(h, t)) = app(rev(t), cons(h, nil))$

$qrev(nil, l) = l$
$qrev(cons(h, t), l) = qrev(t, cons(h, l))$
et prouvons les conjectures:

$x + y = y + x$
$x * y = y * x$
$s(x) + x = s(x + x)$
$dbl(x) = x + x$
$len(app(x, y)) = len(app(y, x))$
$len(app(x, y)) = len(x) + len(y)$
$len(app(x, x)) = dbl(len(x))$
$even(x + x) = true$
$evenm(x + x) = true$
$oddm(s(s(x))) = odd(x)$
$even(s(x + x)) = false$
$oddm(s(x + x)) = true$
$oddm(x + x) = false$
$len(rot(len(x), x)) = len(x)$
$len(rot(x, app(z, cons(nil, y)))) = s(len(rot(x, z)))$
$len(rev(x)) = len(x)$
$rev(rev(x)) = x$
$len(qrev(x, y)) = len(x) + len(y)$
$qrev(rev(x), nil) = x$
$nth(i, nth(j, x)) = nth(j, nth(i, x))$
$nth(i, nth(j, nth(k, x))) = nth(k, nth(j, nth(i, x)))$

Cette méthode marche sur les théories *suffisamment complètes*, c'est à dire dont les données sont construites avec des opérateurs libres (par exemple $cons$, nil, s, 0, $true$, $false$) et des fonctions définies pour toutes les données.

3 Travail demandé

On pourra faire ce projet à plusieurs niveaux. D'abord, il faudra représenter les termes sur les groupes libres ou plus généraux par une structure de données arborescente (syntaxe abstraite). Puis on écrira une procédure de normalisation des termes paramétrée par un système de réécriture. On prendra par exemple une stratégie *innermost* essayant de réduire d'abord les termes en post-ordre. Mais toute stratégie est correcte. Puis on écrira une procédure d'unification de deux termes, et la recherche de paires critiques paramétrée par deux règles de réécritures. On se lancera alors dans la complétion Knuth-Bendix.

On peut faire le projet uniquement sur les groupes et retrouver automatiquement les 10 règles qui donnent un système complet. On pourra aussi faire la complétion paramétrée par un système de réécriture quelconque. (Bien penser à avoir une intervention manuelle pour l'orientation des nouvelles règles dues au paires critiques, sinon il y a un fort risque de divergence).

Enfin, on pourra construire un mini-Spike pour montrer des équations sur des fonctions définies récursivement sur les listes ou les entiers.

Le projet peut se faire dans tout langage de programmation, toutefois il sera plus facile à réaliser en Caml à cause de la gestion automatique de la mémoire, des types somme et des instructions de filtrage.

Références

[1] A. Bouhoula "Automatic case analysis in proof by induction", Proc. 13th IJCAI (Chambery, France, 28 August-3 September 1993), Vol 1, ed. R. Bajscy, IJCAI, Inc., 1993, pp. 88-94.

[2] N. Dershowitz "A Taste of Rewriting", Functional Programming, Concurrency, Simulation and Automated Reasoning, Lecture Notes in Computer Science 693, 199-228 (1993)

[3] J. W. Klop. Chap. 1 in Handbook of Logic in Computer Science, vol. 2, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, eds., Oxford University Press, Oxford (1992) 1-117.

[4] G. Huet, J.-M. Hullot "Proofs by induction in equational theories with constructors", Rept. INRIA 28, Rocquencourt, 1980; also Proc. 21st IEEE Symp. on Foundations of Computer Science, Los Angeles, October 1980, pp. 96-107; also in J. of Computer and Science Systems 25(2):239-266, 1982.

[5] A. J. Robinson "On the mechanization of the theory of equations", Bull. Res. Council Israel, 9F, 1960, pp. 47-70.