Comparaison des termes avec partage

Quentin Ye

Comparaison des termes avec partage

Quentin YE

17 juillet 2018

Table des matières

1 Introduction

Le projet est de concevoir une procédure de comparaison des termes de Dedukti. Lorsque l'on analyse les termes sur lesquels on applique cette procédure, on se rend compte que souvent ils contiennent un sous-terme qui est répété plusieurs fois. Les algorithmes de Dedukti ne prend pas en compte ces répétitions et se retrouve à faire un grand nombre de comparaisons redondantes car il ne se souvient pas des comparaisons précédentes lorsqu'il compare les copies suivantes. On a à la fin une complexité de comparaison qui est de $O(2^n)$.

Notre objectif est d'optimiser l'algorithme des comparaisons des termes et d'avoir une algorithme de complexité polynomiale ou linéaire.

2 Congruence closure

2.1 Definition

L'algorithme de congruence closure fait une relation de congruence donnée par une suite de paire de termes sans les variables pour trouver la classe d'un élément et fusionner deux classes par congruence. La relation de congruence satisfait la symétrie, la réflexivité, la transitivité et la monotonicité c'est-à-dire pour tout f tel que $f(a1...an) = f(b1...bn)$ quel que soit $ai = bi$ pour $\forall i \in \{1...n\}$

Par exemple, on a une équation $a = b$ appartenant à une congruence généré par trois équations :$b = d \wedge f(b) = d \wedge f(d) = a$. On aura à la fin

$$(b = d \wedge f(b) = d \wedge f(d) = a) \equiv a = b$$

L'algorithme de congruence closure permet de comparer deux termes equivalents en *O(n log n)*

2.2 Constante

Nous allons introduire une nouvelle type : les constantes qui remplace tous les termes et sous-termes de Dedukti, ces constants permettront de comparer plus facilement les sous-termes sans faire des comparaisons redondants. Les constantes sont constitues de :

- Kind1 = Kind
- Type1 = Type
- DB1(i) ou i est l'entier et est l'équivalent de DB
- Const1(x) ou x est le nom de la constante et remplace Const
- E(i) qui est une constante qui représente une signature ou i est un entier.

2.3 Signature

Nous allons introduire un nouveau type qui sont les signatures qui sont similaires aux termes non constantes de Dedukti comme par exemple App, Lam et Pi mais constitué de constantes au lieu des termes. Ils sont constitués dans le source code de :

- App1(a,b,c) une application où a et b sont des constantes et c une liste de constantes équivalent a App.
- Lam1(a,b) une application de lambda où b est une constante et a une constante optionnelle qui est équivalent à Lam.
- Pi1(a,b) un produit pi qui contient deux constantes a et b qui est équivalent à Pi.
- Clos1(a,b) une application de clôture où a est une constante et b une liste de constante généré par l'application de lambda.

L'algorithme flatten permet de transformer le terme et ses sous-termes qui le constituent en constantes. Nous assurons qu'un terme n'apparaît pas plusieurs fois dans l'algorithme. Par exemple : $g(f(a,a), f(a,a)) \equiv g(E0, E0)$ *dont* $E0 = f(a,a)$ au lieu de $g(E0, E1)$ *dont* $E0 = f(a,a) \wedge E1 = f(a,a)$

2.4 Structures de donnees

Pour utiliser l'algorithme de congruence closure, nous allons besoin d'utiliser cinq structures de données.

1. Pending : une liste qui contient un couple de deux constants qui sont équivalents et qui vont être fusionnés.
2. Representative appelé représentants : un tableau de constant qui contient pour chaque constante une autre constante qui est son représentant canonique.

3. Classlist appelé classe : un tableau qui prend comme indice des constantes et que les valeurs sont un ensemble des constantes d'une classe équivalente.
4. Lookup : un tableau qui a comme indices des signatures de représentants de la constante et qui renvoie la constante équivalent. Il permet de trouver l'équivalence d'une signature à sa constante qui le définit.
5. Lookupinv : le tableau inverse du lookup, l'indice est une constante et son valeur un signature equivalent.
6. Uselist : un tableau d'indice en constante et qui ont comme valeur une liste de signature où pour une constante a, a est apparue dans l'un des arguments de la signature.

1. Algorithme constant_new(sign) :
2. $n := n + 1$
3. $mettrerepresentative(E(n))aE(n)$
4. $mettrelookup(sign)aE(n)$
5. $ajouterE(n)dansuselist(sign)$
6. $retournerE(n))$

La fonction constant_new permet de créer une nouvelle constante en mettant à jour lookup, representative et uselist.

Au début, tous ces structures de données seront initialisés comme vides et sont des variables globaux. Après avoir appelé l'algorithme flatten sur un terme t, Representative contient tous les sous-termes de t traduit en constantes. Representative et Classlist contient tous les constantes comme leurs propres représentants. Uselist contient pour chaque constante une liste de signature qui ont un des arguments égales à la constante. Lookup contient des signatures avec leurs constantes équivalentes.

2.5 L'algorithme

L'algorithme de congruence closure est constitué de deux opérations :

— merge(s,t) : l'équation s = t est ajoutée dans Pending. s et t sont des constantes.
— areCongruent(s,t) : cette opération vérifie si les deux termes sont équivalents par congruence.

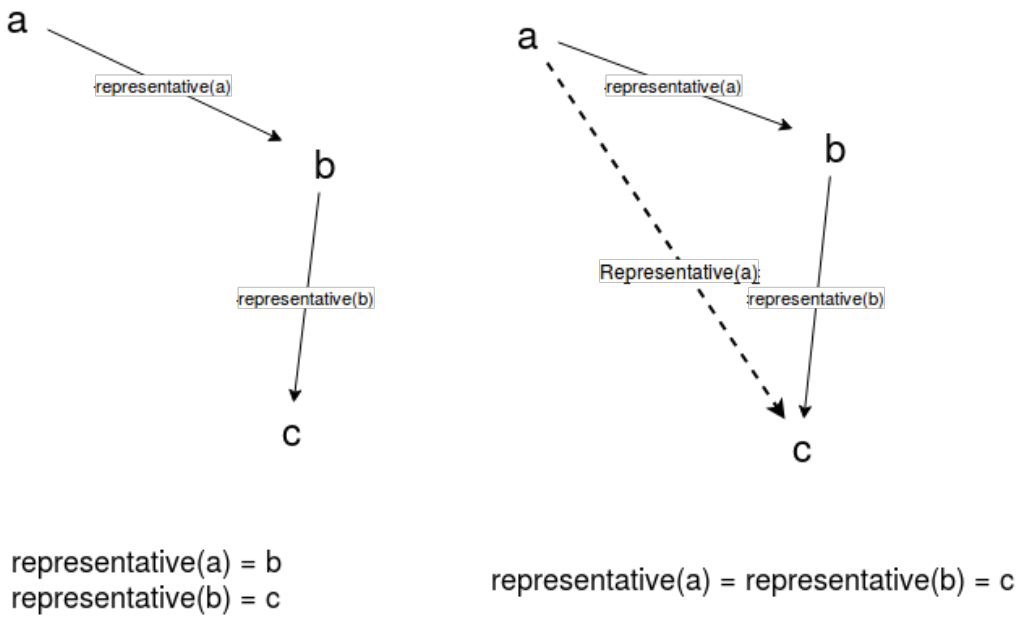
L'algorithme merge permet de réunir deux constantes équivalents.

1. Algoritme merge (s,t) :
2. Ajouter le couple (s,t) dans la liste Pending
3. Initialiser Classlist
4. Propagate()

1. Algorithme propagate () :
2. *Recuperer le couple* (a, b) *de Pending*
3. *si* $Representative(a) \neq Representative(b)$ *alors*
4. *si* $tailleclasslista) \leq taille(classlistb)$ *alors*
5. $old\_repr\_a = Representative(a)$
6. $for\_path\_comp(classlist(old_repr_a)), a, b)$
7. $for\_update\_cong(uselist(old_repr_a)), a, b)$
8. *sinon*
9. $old\_repr\_a = Representative(b)$
10. $for\_path\_comp(classlist(old_repr_a)), b, a)$
11. $for\_update\_cong(uselist(old_repr_a)), b, a)$
12. *finsi*
13. *propagate*()

1. Algorithme for_path_comp(cla,a,b) :
2. *siclaestvidealors*
3. *ajouterclasslist*(*representative*(*b*))*dansclasslist*(*representative*(*b*));
4. *viderclasslist*(*old_repr_a*)
5. *sinon*
6. $c = depiler(cla)$
7. $representative(c) = representative(b)$;
8. *for_path_compclaab*; ;
9. *finsi*

L'algorithme de for_path_comp permet de faire une compression de chemin c'est-a-dire que les représentants seront représentés sous un arbre. Un représentant est un chemin menant vers une constante qui est le racine, les représentants d'une même classe pointe vers un seule et même représentant qui est le racine d'un arbre. Par exemple :



Exemple de compression de chemin

$representative(b) = c$ et $representative(a) = b$ alors $representative(a) = representative(b) = c$

1. Algorithme for_update_cong (usel,a,b) :
2. *si usel est vide alors*
3. *vider classlist*(*old_repr_a*)
4. *sinon*
5. (sign,c) = depiler(usel)
6. $repr\_sign = representative(sign)$;
7. *si lookup*(*repr_sign*) *existe et* $c \neq lookup(repr_sign)$ *alors*
8. $(c, lookup(repr_sign))$ *est ajout dans pending*
9. *sinon*
10. *mettre dans lookup*(*repr_sign*)*ac*
11. *ajouter* $(sign, c)$ *dans uselist*(*representative*(*b*))
12. *for_update_cong*$(usel, a, b)$; ;

13. fin si

Ceci est l'algorithme de for_update_cong : cette algorithme a pour but de mettre a jour le uselist et le classlist. Il vide l'ancien représentant de a puis appelle récursivement pour mettre une nouvelle relation d'équivalence à réunir ou mettre a jour le lookup(repr_sign) à c et ajouter (sign,c) dans uselist du representant de b.

L'algorithme areCongruent permet de vérifier si deux termes sont équivalent. Il retourne vrai si c'est le cas ou faux sinon.

3 Comparaison avec partage

3.1 Constante de clos

Les signatures de clos sont constitués de 2 éléments : un qui est une constante et le deuxième une liste de constantes générée par lambda.

1. Algorithme cst_clos (ct,e) :
2. $si\ (Clos1(ct, e)) \in dans lookup alors$
3. $\quad retourner\ lookup(Clos1(ct, e))$
4. $sinon$
5. $\quad retourner\ (constant\_new(Clos1(ct, e)))$
6. $finsi$

La fonction cst_clos permet de créer une nouvelle constante qui représente le signature de clos au cas où il n'a pas une constante équivalente.

3.2 L'algorithme de comparaison avec partage

1. $compare\ c1\ c2 =$
2. $\quad numiter := !numiter + 1;$
3. $\quad if\ representative\_valuec1 = representative\_valuec2 then$
4. $\quad\quad true$
5. $\quad else$
6. $\quad\quad let\ Clos1(ct1, e1)\ =\ lookupinv\_valuec1 in$
7. $\quad\quad let\ Clos1(ct2, e2)\ =\ lookupinv\_valuec2 in$
8. $\quad\quad if\ (compare2\ (Clos1(ct1, e1))\ (Clos1(ct2, e2)))\ then$
9. $\quad\quad begin$
10. $\quad\quad\quad merge(c1, c2);$
11. $\quad\quad\quad true$
12. $\quad\quad end$
13. $\quad\quad else$
14. $\quad\quad\quad false;;$

L'algorithme compare vérifie si les constantes clos c1 et c2 sont équivalent, si ce n'est pas le cas, il va comparer les sous-termes clos de c1 et c2 en faisant appel à compare2, si les signatures clos sont équivalents alors on va réunir les deux termes en faisant indirectement l'algorithme de congruence closure.

compare2 vérifie que les constantes c1 et c2 sont de même types, si ce n'est pas le cas, il retournera faux. Si les deux constantes sont de type E, nous devrions comparer ses signatures et faire appel à cst_clos pour trouver des constantes clos équivalentes pour utiliser à nouveau compare récursivement les constantes à l'intérieur des signatures.

3.3 Complexité des algorithmes

Prenons par exemple :

$f(u) = (u, u)$ traduit en OCaml $App(Const(\text{"","","}), u, [u])$

$$f^n(u) = (f^{n-1}(u), f^{n-1}(u()$$

Nous pouvons transformer ce terme sous 2 formes :

— forme 1 (function_test1_init dans le fichier code test.ml) :

$$f(u) = App1(Const1(",", "), u, [u])$$

— forme 2 (function_test2_init dans le fichier code test.ml) :

$$f(n) = Clos(f(DB(0), DB(0)), [E(n-1)])$$

Ces formes permettent de réduire le nombre d'itération de comparaison lorsqu'on fait appel à la fonction compare. Supposons qu'on compare deux mêmes termes mais de constantes différentes générées par des procédures du code sources test.ml.

n	Comparaison de base	Comparaison du forme 1	Comparaison du forme 2
5	0.00	0.00	0.00
10	0.00	0.00	0.00
15	0.004	0.004	0.004
20	0.112	0.00	0.012
25	...	0.004	0.016
30	...	0.008	0.028
35	...	0.012	0.048
40	...	0.020	0.064

TABLE 1: Temps d'exécution des algorithmes en secondes.

n	Comparaison de base	Comparaison du forme 1	Comparaison du forme 2
5	2^5	16	21
10	2^{10}	31	46
15	2^{15}	46	71
20	2^{20}	61	96
25	2^{25}	76	121
30	2^{30}	91	146
35	...	106	171
40	...	121	196
50	...	151	246

TABLE 2: Nombres d'itération de chaque algorithmes.

Nous remarquons que la comparaison d'OCaml est de complexité O(2^n) alors que les deux autres sont de complexité O(n) en utilisant le comparaison avec partage. Lorsque n augmente de 1, le nombre d'itération de compare a augmente de 3 dans la forme 1 et 5 dans la forme 2 tandis que l'algorithme de comparaison de base double pour chaque incrémentation. Cela permet également de gagner en espace.

4 Conclusion

Pour conclure, L'algorithme de congruence closure résout en $O(nlog(n)$ et est un moyen de représenter des classes d'équivalence par le choix d'un représentant (Representative).

Nous avons développé un algorithme de comparaison avec partage qui résout le problème d'équivalence en $O(n)$ qui permettre d'utiliser les constantes de manière rapide en passant par l'algorithme de congruence closure qui permet de réunir des sous-termes.