Projet Logique
Algorithme Congruence-Closure

Thomas Filasto

14 mai 2023

1 L'algorithme Congruence-Closure

Étant donné une conjonction de prédicats d'égalités ou d'inégalités sur des variables et des fonctions d'arité au plus 1, on souhaite implémenter un algorithme décidant si la formule est ou non satisfiable. On s'intéresse ici à l'algorithme *Congruence-Closure* qui consiste à partitionner l'ensemble des termes apparaissant dans la formule en les classes d'équivalence de la relation d'égalité. En effet, une formule de ce type est non-satisfiable ssi elle contient un prédicat de la forme $t_i \neq t_j$ tel que t_i et t_j appartiennent à la même classe d'équivalence, c'est à dire sont égaux.
On autorisera également les symboles $\neg$ que l'on simplifiera afin de se ramener uniquement à des prédicats d'égalité ou inégalité.

Présentation de l'algorithme

Entrée : Une conjonction de prédicats d'égalités sur des variables éventuellement précédés de négation
Sortie : Répond *"Satisfiable"* si la formule est satisfiable et *"Non satisfiable"* sinon
Algorithme :

Étape 0 : Éliminer des négations
Étape 1.a : Placer dans une même classe deux variables apparaissant dans un même prédicat d'égalité
Étape 1.b : Fusionner les classes possédant un terme commun
Étape 1.c : Fusionner les classes possédant chacune un terme de la forme $F(t_i)$ où F est le même symbole de fonctions et les t_i sont dans la même classe
Étape 2 : Pour chaque prédicat de la forme $t_i \neq t_j$ vérifier que t_i et t_j n'appartiennent pas à la même classe

L'étape 1 correspond au calcul des classes d'aquivalences, les sous-étapes correspondent à l'utilisation des axiomes de la théorie de l'égalité permettant d'établir l'égalité entre deux termes :

- l'étape 1.a correspond à l'application de la règle hypothèse
- l'étape 1.b correspond à l'application de l'axiome de transitivité
- l'étape 1.c correspond à l'application du schéma d'axiomes de congruence de fonctions

Après cette étape, deux éléments sont dans la même classe d'équivalence ssi ils sont égaux. Donc deux éléments ne peuvent être différents que s'ils n'appartiennent pas à la même classe d'équivalence. Réciproquement, en affectant un symbole différents aux variables apparaissant dans chaque classe d'équivalence tels que ces symboles soient deux à deux non égaux, on construit un einterprétation qui satisfait la formule.
Par conséquent, la formule est satisfiable ssi toute paire termes ne devant pas être égaux ne se trouvent pas dans la même classe d'équivalence, c'est ce que vérifie l'étape 2.

Exemple détaillé

On va dérouler l'algorithme Congruence-Closure sur l'exemple suivant :

$$x_1 = x_2 \wedge x_2 = x_3 \wedge x_4 = x_5 \wedge \neg(f(x_1)) = f(x_3))$$

La première étape est d'éliminer les $\neg$ en transformant les prédicats de la forme $\neg(t_i = t_j)$ en $t_i \neq t_j$, on obtient alors :

$$x_1 = x_2 \wedge x_2 = x_3 \wedge x_4 = x_5 \wedge f(x_1)) \neq f(x_3))$$

On va maintenant construire les classes de congruence pour la relation d'égalité. Pour commencer, on regroupe dans une même classe les termes qui apparaissent dans un même prédicat d'égalité. La classe d'un terme apparaissant dans la formule mais pas dans un prédicat d'égalité est un singleton qui ne contient que lui-même. On obtient alors la partition suivante :

$$\{x_1, x_2\}, \{x_2, x_3\}, \{x_4, x_5\}, \{f(x_1)\}, \{f(x_3)\}$$

La sous-étape suivante consiste à fusionner les classes ayant un terme commun et à itérer ce procédé afin de calculer la clôture de l'ensemble des classes d'équivalence pour l'application de l'axiome de transitivité. Ici, il n'est nécessaire d'appliquer le procédé qu'une seule fois pour fusionner $\{x_1, x_2$ et $x_2, x_3\}$. On obtient :

$$\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{f(x_1)\}, \{f(x_3)\}$$

La dernière sous-étape consiste à faire la même chose pour la congruence des fonctions, c'est à dire qu'on fusionne deux classes possédant un terme de la

forme $f(t)$ avec le même symbole de fonction et où les termes entre parenthèse sont dans la même classe d'équivalence puis on réitère le procédé jusqu'à ce que plus rien ne change. Notons que l'algorithme ne prend en compte que des symboles de fonctions d'arité 1. Ici encore, il suffit d'appliquer le procédé une fois pour fusionner $\{f(x_1)\}$ et $\{f(x_3)\}$ car x_1 et x_3 appartiennent à la même classe d'équivalence. On obtient alors :

$$\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{f(x_1), f(x_3)\}$$

$f(x_1) \neq f(x_3)$ apparaît dans la formule, or $f(x_1)$ et $f(x_3)$ apparaissent dans la même classe d'équivalence, donc la formule n'est pas satisfiable : l'algorithme renvoie **Insatisfiable**.

Implémentation

Représentation des termes et prédicats

La première étape de l'implémentation a été de représenter les formules, les termes et les prédicats. Comme la formule à traiter est une conjonction de prédicats et que l'algorithme nécessite de traiter les prédicats un à un, j'ai décidé de les représenter par des listes de prédicats qui sont simples à parcourir et les prédicats et les variables avec des constructeurs.

Étape 0

Cette étape consiste à éliminer les $\neg$ qui précèdent les prédicats. Pour cela, il suffit de parcourir la liste des prédicats et pour chaque prédicat d'éliminer les $\neg$ de manière inductive de la manière suivante :

- Simpl$(t_1 = t_2) = t_1 = t_2$
- Simpl$(t_1 \neq t_2) = t_1 \neq t_2$
- Simpl$(\neg t_1 = t_2) = t_1 \neq t_2$
- Simpl$(\neg t_1 \neq t_2) = t_1 = t_2$

Ce qui peut être implémenté à l'aide d'une fonction récursive.

Étape 1.a

Pour réaliser cette étape, il faut d'une part créer des paires $\leftarrow t_i, t_j \rightarrow$ pour tout prédicat $t_i = t_j$ et d'autre part créer des singletons pour chaque terme qui n'apparaît pas dans un prédicat d'égalité.

Pour cela, on réalise deux passes : une première au cours de laquelle on ajoute les paires d'éléments pour chaque prédicat d'égalité rencontré à la liste de classes d'équivalence (qui est vide au départ) en ignorant les prédicats d'inégalités. En

même temps, on stocke dans une liste tous les éléments rencontrés. Au cours de la seconde passe, on vérifie pour chaque terme apparaissant dans un prédicat d'inégalité s'il a déjà été rencontré. Si ce n'est pas le cas, on ajoute un singleton le contenant à la liste des classes et on l'ajoute à la liste des termes rencontrés.

Étape 1.b

Pour réaliser cette étape, j'ai décidé de parcourir la liste des classes puis pour chaque classe c rencontrée de reparcourir la liste en ajoutant à une liste leq les classes partageant au moins un terme avec c à la fin du parcours, on fusionne les classes de leq avec c. Le point délicat est de savoir quand on doit s'arrêter de répéter le processus. Pour cela, j'utilise une une référence vers un booléen qui indique si une fusion a eu lieu. Tant que des fusions ont toujours lieu, on continue de répéter le procédé (voir le test 3).

Étape 1.c

Cette étape a été la plus difficile à implémenter. J'ai voulu l'implémenter de manière analogue à la précédente en créant d'abord une fonction qui étant donné une classe c parcourt la liste des classes et établit une liste de celles qui doivent être fusionnées avec c. Pour cela, on utilise une fonction qui étant donnée une classe d'équivalence c liste tous les termes de la forme $f(t_i)$ avec t_i dans c apparaissant dans l'ensemble des classes d'équivalence ainsi que leur classe d'équivalence. Une fois cela fait, toutes les classes contenant un terme de la liste doivent être fusionnées.
La principale difficulté à laquelle j'ai été confronté a été de savoir quand s'arrêter d'itérer le procédé. En effet, contrairement à l'étape précédente, les classes à fusionnée ne doivent pas forcément être fusionnées avec c. Finalement, j'ai décidé de stocker les classes obtenues avant et après application du procédé en les actualisant à chaque fois qu'on applique le procédé jusqu'à ce qu'elles soient égales (en effet le résultat du procédé ne dépend que de la liste des classes).

Étape 2

Pour réaliser cette étape, je fais appel aux fonctions de l'étape 1 pour obtenir la liste des classes d'équivalence. Par ailleurs, j'extrait de la formule toutes les paires de termes devant être différents. Il suffit ensuite de parcourir la liste de ces paires et de vérifier que leurs éléments sont bien différents. Pour cela, on parcourt la liste des classes jusqu'à trouver celle du premier terme. Il suffit ensuite de vérifier que le deuxième terme n'apparaît pas dans la classe, si c'est la cas on renvoie "Insatisfiable". Sinon, on continue à parcourir la liste des paires. Si à la fin du parcours on n'a jamais dû renvoyer insatisfiable, cela signifie que la formule est satisfiable.

Extensions

Pretty printer

J'ai réalisé un pretty printer permettant d'afficher les formules ainsi que les listes de classes. Pour cela, j'ai d'abrd écrit une fonction affichant les termes puis une fonction affichant les prédicats puis pour afficher les conjonctions de prédicats, j'affiche tous les prédicats en les séparant par "$\wedge$". Pour les clauses j'ai fait de même : on affiche les termes en les séparant par des "," puis si la liste est non vide on l'entoure d'accolades. Puis on énumère toutes les classes en les séparant par des virgules.

Prise en charge des fonctions d'arité supérieure à 2

L'algorithme fourni requiert que tous les symboles de fonction apparaissant dans la formule soient d'arité au plus 1. Or, d'après l'axiome de congruence des fonctions de la théorie de l'égalité, si deux termes sont de la forme $f(t_1, ..., t_n)$ avec le même symbole de fonction et que chaque t_i apparaissant entre parenthèse sont égaux alors ils sont égaux. On peut donc généraliser l'algorithme Congruence-Closure en prenant en compte les fonctions d'arité quelconque.

La différence avec l'algorithme actuel apparaît à l'étape 1.3. Il s'agit cette fois de fusionner les classes contenant deux termes $f(t_1, ..., t_n)$ avec le même symbole de fonction et où les t_i de chaque terme sont égaux. Il suffit donc de modifier la fonction *fusionne_congruence*. Toutefois, comme les différents termes apparaissant entre parenthèse peuvent appartenir à des classes différentes, on ne peut pas se contenter de généraliser la fonction existante (ou alors il faudrait considérer les produits cartésiens des des classes d'équivalence et chercher toutes les fonctions dont les arguments sont la même classe que $(t_1, ..., t_n)$ mais ce serait beaucoup pus difficile à implémenter).
J'ai voulu procéder de la manière suivante : on parcourt la liste des classes d'équivalence en parcourant chaque classe. Lorsqu'on tombe sur un terme de la forme $f(t_1, ..., t_n)$, on parcourt les classes suivantes en cherchant un terme de la forme $f(t'_1, ..., t'_n)$. On vérifie alors si $\forall i \in [\![1, n]\!]$, t_i et t'_i sont dans la même classe. Si c'est le cas, on ajoute cette classe à la liste des classes équivalentes à la classe courante (celle qui contient $f(t_1, ..., t_n)$ puis une fois cette dernière parcourue, on la fusionne avec les autres classes de la liste. Puis on réitère le procédé jusqu'à ce que plus aucune classe ne soit modifiée.
Malheureusement mon code ne compile pas et je n'ai pas réussi à résoudre les erreurs.