LOG3210 – Éléments de langages et compilateurs

TP Final

Département de génie informatique et de génie logiciel

École Polytechnique de Montréal

Soumis par

Roman Zhornytskiy (1899786)

Jeannorasinh Tong (1745133)

Groupe : 01

Soumis à Doriane Olewicki

Hiver 2020

1. Description de l'implémentation

1.1 Description de la génération du graphe d'interférence

Notre graphe d'interférence est un HashMap <String, ArrayList<String>> où le String représente le nœud et le ArrayList<String> représente les nœuds qui sont connectés à ce nœud (ce sont les arrêts du noeud).

On construit ce graphe en parcourant, chaque ligne du CODE et chaque nœud du Next_OUT. Si notre graphe d'interférence ne contient pas le nœud du Next_OUT, alors on l'ajoute à notre graphe. Ensuite, on parcourt une autre fois chaque nœud du Next_OUT pour ajouter tous les arrêts du nœud.

En même temps qu'on construit le graphe, on construit notre List<String> nodes qui contient tous les nœuds en ordre alphabétique que nous allons utiliser plus tard.

1.2 Description de la coloration du graphe d'interférence

1. On crée une copie identique de notre graphe d'interférence qui sera utilisé plus tard (nommé : grapheInterferance2) et on crée le Stack pour empiler les nœuds.

Géneration de la Stack

2. Tant que notre graphe d'interférence n'est pas vide :
    a. On sélectionne le nœud ayant le nombre de voisin le plus proche et inférieur à k en parcourant les nœuds en ordre alphabétique. Si rien n'a été trouvé, alors c'est le premier nœud de la List « nodes » qui va être sélectionné.
    b. On enlève le nœud sélectionné (« nodeToStack ») du graphe et on parcourt tous les nœuds du graphe pour enlever les instances de ce nœud dans la liste des autres nœuds (on enlève ses arrêts). On push le nœud sur la Stack.

3. On crée le colorMap<String, Integer> (le string représente le nœud et Integer représente le numéro ou la couleur du nœud)

4. Tant que la Stack n'est pas vide :
    a. color = 0. On pop un nœud et on ajoute ce nœud dans notre graphe d'interférence.
    b. On parcourt tous les nœuds du graphe et on ajoute ses arrêts avec l'aide de la copie identique du graphe.
    c. Coloration se fait en 2 parties :
        i. On crée une liste <Integer> « voisins » ordonnée en ordre croissant. Cette liste contient les valeurs de couleurs de ses voisins.
        ii. On parcourt en ordre croissant cette liste et si la valeur du voisin est égale à color, alors on fait color++.
    d. On ajoute le nœud qu'on a pop et sa valeur de color dans colorMap.

Changement du CODE

5. On parcourt chaque ligne du CODE
    a. Pour chaque ligne on remplace les pointeurs qui commencent par « @ », par « R + color » dans colorMap.

2. Description des tests

En parcourant les tests fournis dans les sources du TP nous avons constaté que ce ne sont pas toutes les opérations qui sont testées. Nous observons que les seules opérations utilisées sont l'addition, la multiplication et la division. Alors, c'est à partir de cette observation que nous avons décidé d'ajouter un test qui contient les trois formes de soustractions.

Forme 1 : t = minus a (la négation)

Forme 2 : t = a – b (soustraction de deux variables)

Forme 3 : t = a – 3 (soustraction d'une constante)

Alors le bloc 3 prouve que notre code est valide, car il alloue correctement des registres à toutes nos variables. De plus, notre code traite la forme 1 comme un SUB RX, #0, RY, la forme 2 comme un MIN RX, RY, RZ et la forme 3 comme un MIN RX, RY, #3.

Nous avons aussi décidé de tester nos 3 cas de réduction de code dans différents blocs. Pour tester le premier cas de réduction de code nous avons ajouté l'expression « a = a » dans le bloc 4. Dans ce cas, le code devrait sauter cette ligne, car elle ne sert à rien et c'est exactement ce que fait notre programme.

Le deuxième cas de réduction que nous avons ajouté est testé dans le bloc 5 qui est la division par 1 (ex : b = b / 1). Dans ce cas, on vérifie que notre code saute aussi cette ligne correctement et on peut affirmer que c'est le cas avec ce test.

Le troisième cas de réduction que nous avons ajouté est testé dans le bloc 6 qui est la multiplication par 1 (ex : c = c * 1 ou d = 1 * d). Nous vérifions encore que notre code saute ces lignes et on confirme que c'est le cas avec les résultats que nous obtenons.

3. Diagramme UML