Travaux dirigés - AST - 21 Février 2020

Question 1.

Complétez la méthode `onePass` suivante, issue de l'implémentation de l'analyse de variables vivantes, afin qu'elle mette à jour les champs `liveIn` et `liveOut` après une itération de point fixe. Il est inutile de se soucier des tables permettant de faire le test de point fixe ici. Vous trouverez en dernière page toutes les méthodes nécessaires pour répondre à cette question.

```
Map<Node,Set<Ident>> liveIn;
Map<Node,Set<Ident>> liveOut;

public void onePass(FlowGraph g) {
  for (Node n : g.nodes()) {
      ...
      this.liveIn.put(n, ...);
      ...
      this.liveOut.put(n, ...);
    }
  }
```

Question 2. Détecter les variables non initialisées

Pour les programmes du langage RTL, on souhaiterait pouvoir détecter statiquement s'il existe une exécution du programme qui peut aboutir à une lecture d'une variable qui n'aurait jamais été affectée au préalable, et qui ne soit pas un paramètre de la fonction courante.

Question 5.1 Expliquez quelle analyse de flot de donnée vue en cours permettrait de faire cela et donnez les grandes lignes de l'algorithme correspondant (sans détails de programmation Java).

Question 5.2 Pensez-vous que cet algorithme est *complet*, c'est-à-dire qu'il pourra trouver toutes les situations de lecture de variables non-initialisées de n'importe quel programme ? Dans le cas contraire, donnez un contre-exemple.

Question 3. Analyse des usages possibles

On s'intéresse à une analyse *des usages possibles* qui calcule en chaque point de programme l, des ensembles $U_{in}(l)$, $U_{out}(l) \in \mathcal{P}(Var \times Lab)$ tels que :

- $U_{in}(l)$ contient l'ensemble des couples (x, n) tels qu'il existe un *usage* de x au point de programme n, accessible depuis **l'entrée** du point l, sans passer par un point de définition de x;

- $U_{out}(l)$ contient l'ensemble des couples (x, n) tels qu'il existe un *usage* de x au point de programme n, accessible depuis **la sortie** du point l, sans passer par un point de définition de x

Question 3.1 Représentez schématiquement (boîtes + flèches) le graphe de flot de contrôle du programme suivant, puis donner le résultat attendu pour son analyse des usages possibles.

```
func Main()
  entry:
    a = 2
    b = 3
    t = Lt(a,0)
    if t goto loop else exit
  loop:
    b = Add(b,2)
    a = Sub(a,1)
    t = Lt(a,0)
    if t goto loop else exit
  exit:
    ret b
```

Question 3.2 Proposez une équation d'analyse de flot de la forme

$$\begin{aligned} U_{in}(l) &= \dots \\ U_{out}(l) &= \dots \end{aligned}$$

pour décrire formellement l'analyse des usages possibles, sans utiliser d'analyses auxiliaires autres que use, def. On pourra s'inspirer des présentations vues en cours pour les analyses de définitions possibles, durée de vie et expressions disponibles.

Question 3.3 Précisez avec quelles valeurs initiales il faudrait initialiser le calcul itératif de point fixe et justifiez ce choix.

Question 3.4 Écrivez les équations associées au programme ci-dessus (pour chaque variable $U_{in}(l)$ $U_{out}(l)$ de chaque point de programme l), puis simplifiez le système pour en obtenir un équivalent, mais sur au plus 6 variables bien choisies.

Question 3.5 Résolvez le système réduit obtenu par une approche itérative en expliquant bien les équations utilisées à chaque itération et les résultats intermédiaires. Expliquez la cohérence des résultats obtenus avec ceux de la question 3.1.

Question 4. Calcul des chaînes *use-def* et *def-use*

La classe ci-dessous implémente l'analyse de la question précédente. Proposez une implémentation pour les deux méthodes `UseDef()` et `DefUse()` qui calculent respectivement les chaînes *use-def* et *def-use* (voir CM4). Le corps de la méthode `build()` n'est pas demandé, l'analyse des usages possibles étant supposée calculée par l'appel au constructeur de la classe, et placée dans les champs `uIn` et `uOut`.

```
public class PossibleUse {

  private class Pair {
    final Ident ident;
    final Node node;
    Pair(Ident ident, Node node) { this.ident = ident; this.node = node; }
    public boolean equals(Object o) { ... }
    public int hashCode() { ... }
  }

  private Map<Node,Set<Pair>> uIn = new Hashtable<>();
  private Map<Node,Set<Pair>> uOut = new Hashtable<>();
  private FlowGraph g;
  private Map<Node,Map<Ident,Set<Node>>> ud;
  private Map<Node,Map<Ident,Set<Node>>> du;

  public PossibleUse(FlowGraph g) {
    this(g);
    this.build()
  }

  private void build() { ... }

  public Map<Node,Map<Ident,Set<Node>>> UseDef() {
    // TODO
  }

  public Map<Node,Map<Ident,Set<Node>>> DefUse() {
    // TODO
  }
}
```

Annexe

```
interface Set<E> implements Collection<E>, Iterable<E> {
  boolean add(E e)
  //Adds the specified element to this set if it is not already present.

  boolean addAll(Collection<? extends E> c)
  //Adds all of the elements in the specified collection to this set if
  //they're not already present.

  void clear()
  //Removes all of the elements from this set.

  boolean remove(Object o)
  //Removes the specified element from this set if it is present.

  boolean removeAll(Collection<?> c)
  //Removes from this set all of its elements that are contained in
  //the specified collection.
}

class HashSet<E> implements Set<E> {
  HashSet()
  //Constructs a new, empty set.

  HashSet(Collection<? extends E> c)
  //Constructs a new set containing the elements in the specified collection.
}

interface Map<K,V> {
  V get(Object key)
  //Returns the value to which the specified key is mapped, or null if this map
  //contains no mapping for the key.

  V put(K key, V value)
  //Associates the specified value with the specified key in this map.
}

class FlowGraph {
  Set<Node> nodes()
  //Returns the set of nodes of this graph

  Set<Ident> def(Node node)
  //Returns the set of identifiers that are defined by the instruction at this node.

  Set<Ident> use(Node node)
  //Returns the set of identifiers that are used by the instruction at this node.
}

class Node {
  Set<Node> succ()
  //Returns the set of successors of this node.

  Set<Node> pred()
  //Returns the set of predecessors of this node.
}
```