

TD Feuille 7 — Analyses de liveness, d'initialisation et élimination de code inutile

```

1 x=d+65;
2 y=d+66;
3 i=y;
4 z=d+67;
5 i=0;
6 while(i<=1000){
7     s=p;
8     if (i==0)
9         p=a;
10    else if (i==1)
11        p=s+b;
12    else
13        p=s+c;
14    if (y > 0)
15        z+=68;
16    i++;
17 }
18 a=b+s;
19 printf("%u\n",s);

```

FIGURE 1 – Code C

Exercice 1 (DCE de base). On considère le code de la figure 1. La seule variable observée est la variable s dans le `printf` ligne 19. Toutes les variables sont de type **unsigned int**. En voilà la liste complète : $a, b, c, d, i, p, s, x, y, z$.

► **Question 1.** Appliquer l'analyse $\forall \langle L_i \rangle S \langle L_o \rangle$ sur cet exemple.

1. En combien de tours de boucle, obtient-on la stabilité de l'invariant?
2. Quelles sont les variables qui ne sont pas *live-in*, c'est-à-dire dont l'analyse garantit que la valeur n'a aucune influence sur la valeur de s en sortie?
3. Trouver une formule donnant la valeur de s en fonction des entrées. En déduire les variables marquées *live-in* et dont la valeur d'entrée n'a aucune influence sur la valeur de sortie de s .

► **Question 2.** Éliminer de ce code les affectations détectées inutiles à l'analyse précédente.

Exercice 2 (Vérification de l'initialisation de variables à la Java). Le langage Java interdit l'utilisation de variables locales non-initialisées : cela participe par exemple à garantir que, dans ce langage, on ne peut pas "forger" de pointeurs.

```

1 x=1;
2 while(x<=7){
3     y=x;
4     x=x+1;
5 }
6 if(7<=y){
7     z=x;
8 } else {
9     z=z+1;
10 }

```

Le problème de savoir si un programme utilise ou pas des variables non-initialisées est indécidable. Le compilateur Java fait donc le choix d'interdire l'utilisation de toute variable locale dont il n'est pas *sûr* qu'elle soit initialisée. Par exemple, alors qu'à l'exécution le programme ci-contre initialise ses variables locales avec $y==7$ et $z==8$ (sans passer par la ligne 9), le compilateur Java rejette ce programme en déclarant que y et z sont utilisées aux lignes 6 et 9 alors qu'elles sont peut-être non-initialisées.

Le professeur Tournesol affirme que l'analyse de Java peut se réduire à un calcul de liveness dans lequel toute affectation est observable : il y a un $\text{obs}(T)$ implicite *avant* toute affectation $x := T$. D'après lui, les variables locales bien initialisées au sens de Java sont celles qui ne sont pas dans le *live-in* à l'entrée de fonction pour cette analyse.

► **Question 1.** Tester l'affirmation de Tournesol sur le programme Java ci-dessus. Qu'en pensez-vous?

► **Question 2.** Le professeur Tournesol affirme aussi qu'il est quand même préférable de faire l'analyse de Java en avant, car ça permet de faire des messages d'erreurs plus appropriés. Définir une analyse en avant $\forall \langle L_i \rangle S \langle L_o \rangle$ qui étant donné un ensemble de variables sûrement initialisées en entrée L_i calcule un ensemble sûrement initialisées L_o en sortie de la commande

gardée S . On considèrera que toute affectation est observable et on notera \mathbb{X} l'ensemble des variables du programme. Que pensez-vous de ce conseil de Tournesol? Testez votre système sur l'exemple de code Java.

Exercice 3 (DCE avec élimination des tests inutiles). On cherche à adapter l'analyse de liveness $\heartsuit \langle . \rangle S \langle . \rangle$ sur les commandes gardées encodant un CFG pour incorporer l'élimination des tests inutiles dans l'analyse. L'idée est qu'un "if C then ε else ε " inutile peut-être éliminé. Or, enlever des affectations inutiles peut créer des tests tels que vus à la question 2 de l'exo 1 : le test de la ligne 14 de la figure 1 devient inutile une fois que l'affectation ligne 15 a été éliminée.

Techniquement, on va faire apparaître les branches éliminées dans l'analyse flot de données, à l'aide d'une variable pc désignant la prochaine branche à exécuter. Il faut pour cela faire l'analyse sur le CFG du programme.

► **Question 1.** Dessiner le CFG du code C de la figure 1, sachant qu'il y a un "return;" implicite à la fin.

Formalisons maintenant la syntaxe des commandes gardées encodant un CFG. On utilise pc comme variable distinguée désignant une des n noms de bloc $(PC_k)_{1 \leq k \leq n}$ 2 à 2 distincts. Par convention, pc est initialisée à PC_1 au démarrage d'un corps de fonction F .

Ci-dessous, la lettre p représente une valeur possible de $(PC_k)_{1 \leq k \leq n}$.

corps de fonction $F ::= pc := PC_1; (\bigsqcup_{i=1}^n \text{assume}(pc = PC_k); K_k; B_k)^*$

suites de calculs $K ::= \varepsilon \mid x := T \mid \text{obs } T \mid K_1; K_2$

branchements $B ::= \text{return} \mid pc := p \mid pc := (C?p_1 : p_2)$

Ici $pc := (C?p_1 : p_2)$ est juste du sucre syntaxique pour "if C then $pc := p_1$ else $pc := p_2$ ".

Un **état abstrait** A est un map qui associe chaque PC_k à une *condition abstraite de bloc* qui est soit de la forme " L ", soit de la forme " $G(p')$ " avec la signification suivante

$A(p) = L$ donne l'ensemble L *live-in* du bloc p .

$A(p) = G(p')$ signifie que le bloc p est équivalent $\varepsilon; pc := p'$. On lit " $G(p')$ " comme *Goto* p' .

On définit la relation G_A par $(p, p') \in G_A \Leftrightarrow A(p) = G(p')$. On note G_A^* sa clôture réflexive et transitive. On va se limiter à construire des états abstraits A tels que pour tout p , il existe un unique p' vérifiant $(p, p') \in G_A^*$ et $\forall p'', (p', p'') \in G_A \Rightarrow p' = p''$. On note abusivement $G_A^*(p)$ un tel p' . On a ainsi $(p_1, p_2) \in G_A^* \Leftrightarrow G_A^*(p_1) = G_A^*(p_2)$. Autrement dit, G_A^* correspond à une structure de **union-find**.

Notons que, par construction, $A(G_A^*(p))$ retourne soit un ensemble L de *live-in*, soit un $G(p')$ avec $G_A^*(p) = p'$ et donc $A(p') = G(p')$. Ce dernier cas correspond à un bloc p qui boucle sur lui-même : son *live-in* est donc \emptyset . On définit ainsi le *live-in* associé à $G_A^*(p)$, noté $L_A(p)$, par

$$L_A(p) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{si } A(G_A^*(p)) = G(G_A^*(p)) \\ A(G_A^*(p)) & \text{sinon} \end{cases}$$

Ceci induit un ordre $A_1 \sqsubseteq A_2$ équivalent à $\forall k \in [1, n], L_{A_1}(PC_k) \subseteq L_{A_2}(PC_k)$ qui donne *in fine* la structure de **treillis borné** de nos états abstraits.

► **Question 2.** On considère une variante des conditions de blocs, notée a , de la forme ci-dessous, où le A représente un état abstrait global qui permet de calculer $G_A^*(p)$ et $L_A(p)$.

$$a ::= L|(A, G(p'))$$

Définir le système $\heartsuit\langle a_i \rangle K \langle a_o \rangle$ pour traiter les suites de calculs K en s'inspirant du système $\heartsuit\langle L_i \rangle S \langle L_o \rangle$ vu en cours. L'idée est de propager (en arrière) la forme $(A, G(p'))$ tant qu'on peut éliminer des affectations.

► **Question 3.** Définir le système $\heartsuit\langle a \rangle B \langle A \rangle$ pour traiter les branchements. **C'est là qu'on peut éliminer les tests inutiles!**

L'analyse d'un bloc "**assume** ($pc = PC_k$); K_k ; B_k " pour $k \in [1, n]$ et pour un état abstrait A en post, trouve un a_k puis un b_k tels que $\heartsuit\langle a_k \rangle B_k \langle A \rangle$ et $\heartsuit\langle b_k \rangle K_k \langle a_k \rangle$. On définit alors la nouvelle condition de bloc, notée \widehat{b}_k , comme valant $G(p')$ si b_k est de la forme $(A, G(p'))$ ou L si b_k est de la forme L . Le nouvel état abstrait est alors $A \oplus \{PC_k \mapsto \widehat{b}_k\}$. Partant d'un état abstrait initial $\{PC_k \mapsto \emptyset\}_{1 \leq k \leq n}$, l'analyse itère cette transformation sur tous les blocs (dans un ordre arbitraire) jusqu'à ce que chacun de leur état soit stable. Notons toutefois que l'itération converge plus ou moins vite suivant **l'ordre des blocs parcouru**. En pratique, il est bien de suivre l'ordre qu'on aurait suivi dans la représentation structurée.

► **Question 4.** Appliquer cette analyse au CFG de la question 1 : en s'inspirant du parcours sur la représentation structurée, commencer par donner l'ordre de parcours des blocs à l'aide d'une expression régulière dans le vocabulaire des $(PC_k)_k$ —comme c'est une analyse en arrière, on commence par la fin bien sûr.

Une fois l'analyse effectuée, éliminer les affections et tests inutiles d'après l'analyse : on donnera le résultat final sous la forme d'un programme structuré