TD Feuille 4 — Analyse statique des déréférencements de pointeurs nuls

Le but de cet exercice est de définir une analyse statique visant à vérifier que les pointeurs NULL ne sont jamais déréférencés. On considère initialement un fragment du C ne contenant que des variables de type `int *`, et pour simplifier des hypothèses similaires au cas de Java : il y a un `malloc`, mais pas de `free`, pas d'arithmétique de pointeurs et tous les pointeurs sont implicitement initialisés à NULL. De plus, on suppose qu'il n'y a pas de pointeurs de pointeurs (ce qui n'est pas vrai en Java). On considère par contre que `malloc` peut retourner un pointeur NULL (ou pas). On voudrait définir une analyse statique qui détecte :

— des accès mémoire qui, s'ils sont exécutés, provoqueront systématiquement un déréférencement d'un pointeur `NULL` (signalés en rouge)

— des accès mémoire qui ne provoqueront jamais un déréférencement d'un pointeur `NULL` (signalés en vert)

Comme il n'y a ni `free` et ni pointeur non initialisé, on admet que les pointeurs non nuls sont valides. On utilisera aussi les couleurs suivantes :

— la couleur orange pour signaler qu'on ne sait pas décider

— la couleur grise pour signaler du code mort.

En particulier le code mort peut apparaitre si des vérifications ont déjà eu lieu. Par exemple :

```
 o = NULL;
*o = 3 ;    // en rouge déréférencement certain de NULL
*o = 3 ;    // en gris: inaccessible étant donné bug précédent
```

On coloriera tout le code inaccessible en gris (pas seulement les assertions).

Exercice 1 (Analyse manuelle). On considère chacun des deux fragments de code ci-dessous (après une initialisation implicite à `NULL`). Insérer les assertions RTE pour prévenir l'absence de déréférencement des pointeurs nuls. Puis colorier ces assertions, en vous basant sur votre compréhension intuitive du C.

```
1   p = malloc(sizeof(int)) ;
2   *p = 4;
3   *o = 5 ;
4   *o = *p ;
```

```
p = malloc(sizeof(int)) ;      1
if (p != o) {                  2
  *p = 4;                      3
  o = p;                       4
}                              5
if (o != NULL) {               6
  *o = 5 ;                     7
}                              8
*o = *p ;                      9
```

```
// initialement o=p=NULL
p = malloc(sizeof(int));
assert(p != NULL);  // orange
*p = 4;
assert(o != NULL);  // rouge
*o = 5 ;            // gris
assert(o != NULL);  // gris
assert(p != NULL);  // gris
*o = *p ;           // gris
```

```
// initialement o=p=NULL
p = malloc(sizeof(int)) ;
if (p != o) {
  assert(p != NULL);  // vert
  *p = 4;
  o = p;
}
if (o != NULL) {
  assert(o != NULL);  // vert
  *o = 5 ;
}
assert(o != NULL);  // orange
assert(p != NULL);  // orange
*o = *p ;
```

Exercice 2 (Mise en place du treillis borné). On veut utiliser ici un domaine non-relationnel $\mathbb{D}$ où les états abstraits sont des maps associant une valeur abstraite d'un domaine $\mathbb{V} \stackrel{def}{=} \{T, F\}^2$ à chaque variable x : si $A \in \mathbb{D}$ et x une variable du programme, on note $A(x)$ la valeur abstraite associée à x. De plus, un élément $v \in \mathbb{V}$ associé à une variable x est alors un couple de booléens $(v.Z, v.N)$ tel que :

- si $v.Z = F$ alors $x \neq$ `NULL`;
- et si $v.N = F$ alors $x =$ `NULL`.

▹ **Question 1.** Soit X l'ensemble des variables du programme et un état abstrait $A \in \mathbb{D}$. Définir $\gamma(A)$, la concrétisation de A, c'est-à-dire, dans le formalisme du cours, la proposition logique qui caractérise l'ensemble d'états représenté par A.

$$\gamma(A) \equiv \bigwedge_{x \in X} (\neg A(x).Z \Rightarrow x \neq \texttt{NULL}) \wedge (\neg A(x).N \Rightarrow x = \texttt{NULL})$$

Note : le cours, qui définit le γ à partir de la sémantique des domaines abstraits en tant qu'ensembles, fait un abus de notation. Il est plus correct – comme on le fait ici – de définir le γ sur la représentation informatique des domaines abstraits. Ainsi, le γ donne justement une sémantique des domaines abstraits en tant que propositions logiques.

▹ **Question 2.** On considère l'assertion "`assert(p != NULL);`" dans le cas où on a une unique variable p. Donner la couleur dans laquelle il faut colorier l'assertion en fonction de la valeur abstraite v de p.

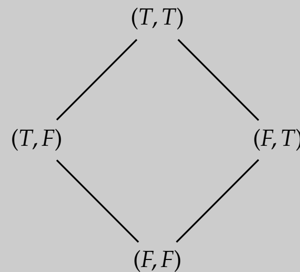
- orange pour $v.Z = T$ et $v.N = T$
- vert pour $v.Z = F$ et $v.N = T$

— rouge pour $v.Z = T$ et $v.N = F$
— gris pour $v.Z = F$ et $v.N = F$

▹ **Question 3.** On considère la façon dont il faut définir le treillis $\mathbb{V}$ pour que la coloriage de l'interpréteur générique du cours fonctionne comme attendu.

1. Quel est son $\top_\mathbb{V}$? Quel est son $\bot_\mathbb{V}$? Exprimer son $\sqcup_\mathbb{V}$ et son $\sqcap_\mathbb{V}$ à l'aide d'opérateurs booléens (logiques) usuels. De même exprimer $\sqsubseteq_\mathbb{V}$ à partir d'opérateurs logiques. Dessiner le treillis borné de $\mathbb{V}$. Quelle est sa hauteur ?
2. Retrouver comment sont définis le $\top$, $\sqcup$, $\sqcap$ et $\sqsubseteq$ du domaine non-relationnel $\mathbb{D}$ sur l'ensemble de variables X à partir de ceux de son domaine $\mathbb{V}$ de valeurs abstraites.
3. Soit $A \in \mathbb{D}$, avec $A(x) = \bot_\mathbb{V}$. Que peut-on dire sur $\gamma(A)$? Comment définir $\bot$?

1. Le treillis est de hauteur 2, avec $\top_\mathbb{V} \stackrel{def}{=} (T,T)$, $\bot_\mathbb{V} \stackrel{def}{=} (F,F)$,
   $(Z_1,N_1) \sqcup_\mathbb{V} (Z_2,N_2) \stackrel{def}{=} (Z_1 \vee Z_2, N_1 \vee N_2)$ et $(Z_1,N_1) \sqcap_\mathbb{V} (Z_2,N_2) \stackrel{def}{=} (Z_1 \wedge Z_2, N_1 \wedge N_2)$ et $(Z_1,N_1) \sqsubseteq_\mathbb{V} (Z_2,N_2) \stackrel{def}{=} (Z_1 \Rightarrow Z_2) \wedge (N_1 \Rightarrow N_2)$.



2. Pour $\mathbb{D}$, on a $\top = \{x \mapsto \top\}$; $A_1 \sqcup A_2 \stackrel{def}{=} \{x \mapsto A_1(x) \sqcup_\mathbb{V} A_2(x)\}$; idem pour $\sqcap$.
   Enfin, $A_1 \sqsubseteq A_2 \stackrel{def}{=} \bigwedge_{x \in X} A_1(x) \sqsubseteq_\mathbb{V} A_2(x)$.
3. Si $A(x) = \bot_\mathbb{V}$, alors on doit avoir $\gamma(A) = \texttt{false}$. En toute rigueur, pour avoir l'unicité de $\bot$, il faut "normaliser" les états abstrait à $\bot$ dès qu'il y a un x tel que $A(x) = \bot$. C'est une approche systématique dans les domaines non-relationnels.

Exercice 3 (Interprétation abstraite des affectations). Dans un code comme celui de l'exo 1, l'interprétation abstraite des affectations sur le contenu des pointeurs "`*p = ...`" correspond juste à un `assert` : l'effet de bord est ignoré (le contenu des pointeurs est ignoré par le domaine abstrait). Définir l'interprétation de "$A[p_1 := T]$" uniquement pour les 3 cas suivants : $T \in \{\texttt{NULL}, \texttt{malloc(sizeof(int))}, p_2\}$ où p_1 et p_2 sont des variables.
On notera $A \oplus \{p_1 \mapsto v\}$ la mise à jour de A en p_1 avec la valeur abstraite v.

1. $A[p_1 := \texttt{NULL}] \stackrel{def}{=} A \oplus \{p_1 \mapsto (T,F)\}$ – ce qu'on aurait aussi pu noter $\texttt{store}(A, p_1, (T,F))$.
2. $A[p_1 := \texttt{malloc(sizeof(int))}] \stackrel{def}{=} A \oplus \{p_1 \mapsto (T,T)\}$
3. $A[p_1 := p_2] \stackrel{def}{=} A \oplus \{p_1 \mapsto A(p_2)\}$

Rappeler qu'il faudrait en tout rigueur vérifier que ces définitions satisfont la propriété de la diapo 9 du cours sur $\gamma(A[p := T])$, même si ça peut sembler assez évident... Une remarque similaire s'applique aussi à l'exo qui précède et l'exo qui suit.

Exercice 4 (Interprétation abstraite des gardes). Définir les gardes suivantes :

1. $A \sqcap (p \texttt{ == NULL})$
2. $A \sqcap (p \texttt{ != NULL})$
3. $A \sqcap (p_1 \texttt{ == } p_2)$
4. $A \sqcap (p_1 \texttt{ != } p_2)$

1. $A \sqcap (p \texttt{ == NULL}) \stackrel{def}{=} A \sqcap \{p \mapsto (T,F)\}$. Attention, ici dans la notation $\{p \mapsto (T,F)\}$, les autres variables que p sont implicitement associées à $\top_{\mathbb{V}}$.
2. $A \sqcap (p \texttt{ != NULL}) \stackrel{def}{=} A \sqcap \{p \mapsto (F,T)\}$
3. $A \sqcap (p_1 \texttt{ == } p_2) \stackrel{def}{=} A \sqcap \{p_1 \mapsto A(p_2)\} \sqcap \{p_2 \mapsto A(p_1)\}$
4. Attention, pour $A \sqcap (p_1 \texttt{ != } p_2)$, ce serait incorrect de retourner $\perp$ quand p_1 et p_2 sont issus de 2 mallocs non nuls distincts. Donc, si $A(p_1) = (T,F)$ ou $A(p_2) = (T,F)$, on se ramène au cas 2 ci-dessus. Et sinon, on retourne A.

Exercice 5. Exécutions de l'interpréteur

▹ **Question 1.** Dériver une règle de l'interpréteur abstrait pour la commande gardée **if** C **then** S_1 **else** S_2. Puis calculer les états abstraits pour chaque ligne des exemples de l'exo 1 (après insertion des asserts RTE et en simplifiant les négations dans les conditions). Le coloriage de l'intepréteur correspond-il à celui donné à l'exo 1 ?

Règle du if :

$$\frac{{}^{\#}\{A \sqcap C\}\, S_1\, \{A_1\} \qquad {}^{\#}\{A \sqcap \neg C\}\, S_2\, \{A_2\}}{{}^{\#}\{A\}\ \textbf{if}\ C\ \textbf{then}\ S_1\ \textbf{else}\ S_2\, \{A_1 \sqcup A_2\}}$$

Pour faire les calculs, il est aussi pratique de dériver que sous l'hypothèse $A \not\sqsubseteq \perp$, on a :

$$\frac{}{{}^{\#}\{A\}\ \textbf{assert}(p \neq \texttt{NULL})\, \{A \sqcap \{p \mapsto (F,T)\}\}}\ \neg A(p).Z$$

Ça donne

```
// initialement o=p=(T,F)
p=malloc(sizeof(int));// p=(T,T)
assert(p != NULL);   // p=(F,T)
*p = 4;
assert(o != NULL);   // o=(F,F)
*o = 5 ;             // idem
assert(o != NULL);   // idem
assert(p != NULL);   // idem
*o = *p ;            // idem
```

```
// initialement o=p=(T,F)
p=malloc(sizeof(int));// p=(T,T)
if (p != o) {          // p=(F,T)
  assert(p != NULL);   // p=(F,T)
  *p = 4;
  o = p;               // o=(F,T)
} else { // o=p=(T,F)
} // o = p = (T,T)
if (o != NULL) { // o = (F,T)
  assert(o != NULL);   // o = (F,T)
  *o = 5 ;
} else { // o=(T,F)
} // o = p = (T,T)
assert(o != NULL);   // o = (F, T)
assert(p != NULL);   // p = (F, T)
*o = *p ;
```

On retrouve bien les couleurs attendues.

▹ **Question 2.** Dans l'exemple de droite de l'exo 1, on remplace la ligne 6 par `p != NULL`. Calculer les états abstraits pour chaque ligne de code. Commenter le coloriage des assertions sur cet exemple. Qu'en déduisez-vous sur la complétude de cette analyse statique ?

En commençant directement à la ligne qui change :

```
1  } // o = p = (T,T)
2  if (p != NULL) { // p = (F,T)
3    assert(o != NULL);   // o = (F,T)
4    *o = 5 ;
5  } else { // p=(T,F)
6  } // o = p = (T,T)
7  assert(o != NULL);   // o = (F, T)
8  assert(p != NULL);   // p = (F, T)
9  *o = *p ;
```

Le coloriage des assert ne change qu'au premier assert du code ci-dessus qui passe à l'orange. C'est un faux positif, car dans la sémantique concrète, on a `o==p` à cet endroit. Mais le domaine abstrait ne permet pas d'inférer cette égalité. Cette analyse statique est donc incomplète.

▹ **Question 3.** Comme on est sur un treillis de hauteur finie, on prend $\nabla \stackrel{def}{=} \sqcup$. Quel est l'invariant de boucle trouvé par l'intepréteur dans le code ci-dessous ? En combien d'itérations de la boucle ? De quelle couleur colorie-t-il les assert ? On commence toujours dans un état initial où toutes les variables sont à NULL.

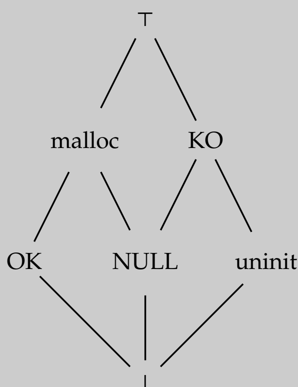
```
1  p=malloc(sizeof(int));
2  if (p!=NULL) {
```

```
3      while (o1 == o4)  {
4        o1=o2;
5        o2=o3;
6        o3=p;
7      }
8      assert (o1 != NULL);
9      assert (p != NULL);
10     assert (o2 != NULL);
11   }
```

A l'entrée de la boucle toutes les variables sont à (T, F) sauf p à (F, T). A l'issue du premier tour (ligne 6), seul o_3 à changé en (F, T). Donc à l'entrée du 2° tour, o_3 prend la valeur (T, T). A l'issue du 2° tour, on a maintenant $o_2 = o_3 = (T, T)$ (les autres étant dans l'état d'avant la boucle). Donc à l'entrée du 3° tour, idem. Et en sortie $o_1 = o_2 = o_3 = (T, T)$. On refait un 4° tour de boucle et là, on a trouvé l'invariant $o_1 = o_2 = o_3 = (T, T)$ avec $p = (F, T)$ et $o_4 = (T, F)$. Donc les 2 premiers assert sont verts et le dernier orange (fausse alarme).
NB c'est le résultat de `frama-c-gui -eva` sur ce programme (cf. fichier `pointfixe.c` dans le dépôt enseignant).

Exercice 6 (Extension avec pointeurs non-initialisés). On relâche l'hypothèse que les pointeurs sont bien initialisés. En conséquence, au niveau des assert RTE, il faut maintenant garantir que les déréférencements ont bien lieu sur des pointeurs non nuls ayant été initialisés avec un `malloc`. Quel $\mathbb{V}$ faudrait-il prendre pour conserver un analyseur correct ? Quelle serait la valeur abstraite retournée par un `malloc` ?

Il faut partir sur un sous-treillis de $\mathcal{P}(\{\texttt{uninit}, \texttt{NULL}, \texttt{OK}\})$ où `OK` représente un malloc non nul et `uninit` une valeur représentant la non-initialisation (qui est traditionnellement une valeur poison vis-à-vis des opérations du langage). Le malloc retourne donc $\texttt{NULL} \sqcup \texttt{OK}$. La valeur $\texttt{KO} = \texttt{NULL} \sqcup \texttt{uninit}$ est utile pour colorier en rouge certains déréférencements.



Exercice 7 (Extension avec `free`). À la suite de l'exercice précédent, on envisage de gérer "`free(p)`" comme une désinitialisation du pointeur `p` sous la précondition que `p` est initialement initialisé et non-nul (c'est-à-dire que le pointeur `p` sera alors considéré dans la suite

de l'analyse comme non-initialisé). Est-ce que cette analyse statique reste correcte ? Justifier la réponse.

Non, c'est pas correct. Voilà un contre-exemple avec un UAF (Use-After-Free).

```
p = malloc(sizeof(int));
if (p != NULL) {
  q = p;
  free(p);
  *q = 5;
}
```

Pour corriger il faudrait avoir une abstraction plus fine des pointeurs, qui modélise le contenu des adresses mémoires, comme fait le plugin EVA de frama-c (ce qui lui permet de détecter un tel UAF). En effet, après un "`free(p)`", ce n'est pas la variable `p` qui devient non-initialisée, mais le contenu associé à cette adresse mémoire.