TD3 : Unification et analyse de vivacité

Exercice 1

- Oui, avec X :=int et Y :=int.
- Oui, avec $X := tab[Y]$ et $Z := tab[Y]$.
- Non, il y a un clash quand on unifie Y et $tab[Y]$.
- Oui, avec $X := (Z \to Z) \to Z$, $Y := V$ et $W := Z \to Z$.

Exercice 2

Le type de f est initialement $(X, Y, Z) \to R$, et on doit unifier :

- X et $tab[Y]$ à cause de `x==`{`y`}
- $tab[Z]$ et X à cause de {`z`}`+x`
- X et R à cause du premier return
- Z et Y à cause de {`z,y`}
- $tab[Z]$ et R à cause du second return

Ces 5 contraintes sont unifiables avec $X := tab[Z]$, $Y := Z$ et $R := tab[Z]$, ce qui donne le type $(tab[Z], Z, Z) \to tab[Z]$.

Une analyse similaire pour $g : (A, B) \to S$ donne les contraintes:

- $A \sim tab[A']$
- $B \sim tab[Z]$, $A \sim Z$, $S \sim Z$
- $tab[Z] \sim B$
- $B \sim tab[S]$

Ce qui donne le type $(tab[A'], tab[tab[A']]) \to tab[A']$.

Remarque : après la compilation de g, le type de f est toujours $(tab[Z], Z, Z) \to tab[Z]$!

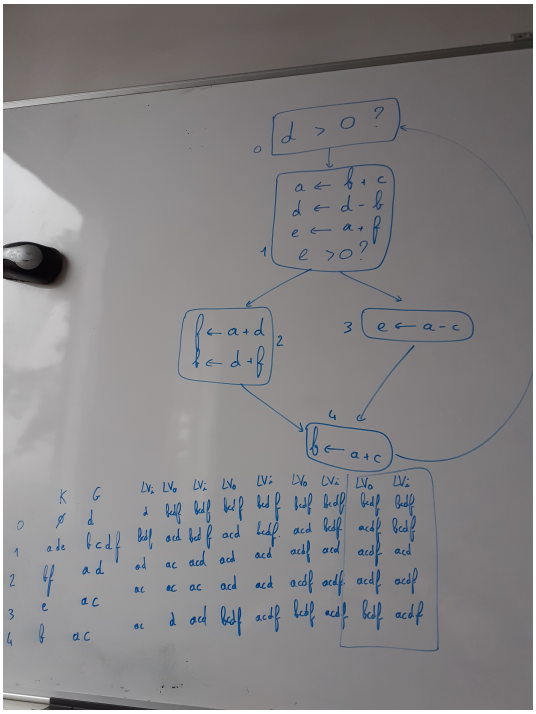
Exercice 3

On obtient le tableau suivant:

Bloc	kill	gen	live
L0	a, i	$\varnothing$	a, i, k
L1	$\varnothing$	k	a, i, k
L2	$\varnothing$	a	k
L3	b, j	i	a, b, i, j, k
L4	a	a, b, j, k	a, i, k
L5	j, k	a, k	a, i, k
L6	$\varnothing$	k	$\varnothing$

Exercice 4

On obtient le graphe suivant :



et on a :

Bloc	kill	gen	LV_{entry}	LV_{exit}
B0	$\varnothing$	d	b, c, d, f	b, c, d, f
B1	a, d, e	b, c, d, f	b, c, d, f	a, c, d, f
B2	b, f	a, d	a, c, d	a, c, d, f
B3	e	a, c	a, c, d, f	a, c, d, f
B4	b	a, c	a, c, d, f	b, c, d, f

Exercice 5

1. On obtient le code suivant:

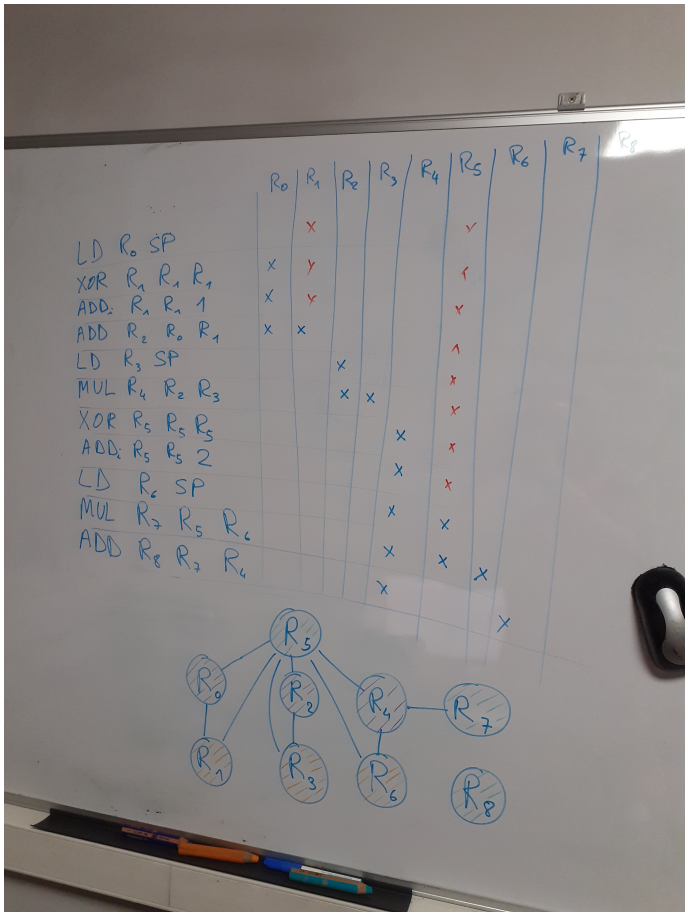
```
1 LD R0 SP
2 XOR R1 R1 R1
3 ADDi R1 R1 1
4 ADD R2 R0 R1
5 LD R3 SP
6 MUL R4 R2 R3
7 XOR R5 R5 R5
8 ADDi R5 R5 2
```

```
 9 LD R6 SP
10 MUL R7 R5 R6
11 ADD R8 R4 R7
```

2.

Registre	Intervalle de vivacité
R0	2,3,4
R1	0-4
R2	5,6
R3	6
R4	7-11
R5	0-10
R6	10
R7	11
R8	∅

3. Le graphe de conflit est le suivant :



4. Une coloration optimale est donnée sur la figure.

5. Méthode sans pagination: on utilise R_0 pour stocker les registres verts, R_1 pour les oranges. Pour le registre violet, on utilise la mémoire, à l'adresse 0. Le code devient

```
 0’ XOR R3 R3 R3
  1 LD R2 SP
 1’ ST R2 R3
  2 XOR R1 R1 R1
  3 ADDi R1 R1 1
 4’ LD R2 R3
  4 ADD R2 R2 R1
4’’ ST R2 R3
  5 LD R1 SP
 6’ LD R2 R3
  6 MUL R2 R2 R1
6’’ ST R2 R3
  7 XOR R0 R0 R0
  8 ADDi R0 R0 2
  9 LD R1 SP
 10 MUL R0 R0 R1
11’ LD R2 R3
 11 ADD R0 R2 R0
```

Méthode avec pagination :

```
  0 XOR R4 R4 R4
  0 ADDi R3 R4 1
  1 LD R0 R5
  2 LD R1 R3
  2 XOR R1 R1 R1
  3 ADDi R1 R1 1
  4 ADD R0 R0 R1
  5 LD R1 R5
  6 MUL R0 R0 R1
  7 LD R2 R4
  7 ST R0 R4
  7 ADDi R0 R2 0
  7 XOR R0 R0 R0
  8 ADDi R0 R0 2
  9 LD R1 R5
 10 MUL R0 R0 R1
 11 LD R2 R4
 11 ST R1 R4
 11 ADDi R1 R2 0
 11 ADD R0 R1 R0
```

Remarques :

- On est partis d'une table de pagination vide (tout en défaut, aux adresses 0 1 et 2).
- on a une optimisation pour l'instruction 2 (pas besoin de ST puisqu'on écrase la valeur).
- attention au buffer tournant quand on a besoin de charger les deux arguments d'une instruction en mémoire