Devoir en temps libre n° 3 — corrigé

I Quelques exemples de calculs de complexité

I.1 Retour sur le DM n° 2

EXERCICE 2 (*Minimum et maximum d'une liste*)

1. Soit $d \in \mathcal{D}_n$ une liste de longueur $n \geq 1$. On entre dans le troisième cas du filtrage, pour lequel il y a deux appels récursifs à la queue de longueur $n-1$, notée q, et des opérations en temps constant. On a donc $C(d) = 2C(q) + \Theta(1)$ et donc $C(n) \leq 2C(n-1) + \Theta(1)$. On ne sait pas *a priori* s'il y a bien égalité, mais il suffit de remarquer que $\sup_{d=t::q\in\mathcal{D}_n} C(q) = \sup_{q\in\mathcal{D}_{n-1}} C(q)$ pour conclure.
2. $\displaystyle\sum_{k=1}^{n} \frac{1}{2^k} = 1 - \frac{1}{2^n} \leq 1.$
3. Pour $k \geq 1$ on a $C(k) - 2C(k-1) = \Theta(1)$ et on divise par 2^k.
4. On a donc $u_n - u_0 = \displaystyle\sum_{k=1}^{n} u_k - u_{k-1} = \Theta(1)$ et donc $C(n) = \Theta(2^n)$.
5. On obtient un arbre d'appels récursifs, complet, de profondeur 4. Chaque nœud interne possède deux fils identiques. Le nombre de nœuds est $2^5 - 1 = 31$. On obtient une complexité exponentielle puisqu'à chaque étape le nombre d'appels récursif double.
6. Il est dommage de calculer deux fois exactement la même chose. Il suffit de mémoriser le résultat à l'aide d'une liaison locale :

```
OCAML
| tete :: queue ->
      let mini, maxi = min_et_max queue
      (min tete mini, max tete maxi)
```

   L'équation de récurrence devient alors $C(n) = C(n-1) + \Theta(1)$ dont la solution est bien une complexité linéaire.

☞ **Remarque 1**

On peut aussi commencer par calculer le min *et le* max *d'une liste, indépendamment, en $\Theta(n)$, et faire ensuite un appel à ces deux fonctions. On obtient aussi une complexité linéaire en $\Theta(2n) = \Theta(n)$. Mais un seul parcours de liste est souvent exigé à CCP par exemple.*

7. Ma solution dans le corrigé du DM n° 2 était bien de complexité linéaire (ouf !).

PROBLÈME 1 *Représentation des polynômes*

Dans la question 7, la fonction `nettoyage` revenait à supprimer les 0 en fin de liste. La plupart des solutions proposées étaient :

```
OCAML
let rec nettoyage polynome =
  match List.rev polynome with
  | [] -> []
  | 0. :: queue -> nettoyage (List.rev queue)
  | _ -> polynome
```

1. La complexité de cette fonction vérifie une équation de récurrence de la forme $C(n) = C(n-1) + \Theta(n)$, dont la solution est en $\Theta(n^2)$ (le pire cas est atteint si chaque élément de la liste est un 0). À chaque appel récursif il y a deux appels à la fonction `rev` ce qui justifie le coût en $\Theta(n)$ dans l'équation de récurrence.
2. Il ne faut pas renverser la liste à chaque appel, mais une fois au tout début et à la toute fin. Cf. corrigé DM n° 2.

I.2 Complexité du tri fusion

1. Cf. corrigé TP n° 5.
2. Pour une liste de longueur n, il y a $\lceil n/2 \rceil$ appels récursifs à la fonction `division`, de coûts constants, pour une complexité finale linéaire $C_{\text{division}}(n) = \Theta(n)$. Pour la fonction `fusion`, on choisit comme taille n la somme des longueur des deux listes. En dehors des deux cas de base, cette taille diminue de 1 à chaque appel récursif et il y a donc au plus n appels récursifs pour une complexité linéaire en $O(n)$. Un pire cas possible est celui où la première liste est formée d'un élément plus grand que les $n-1$ éléments de la deuxième liste, ce qui montre que la complexité est en $C_{\text{fusion}}(n) = \Theta(n)$.

3. Pour $n \geq 0$, notons $c_n = C_{\text{division}}(n) + C_{\text{fusion}}(n)$ le coût dans le pire cas cumulé de la division et de la recombinaison d'une entrée de taille n. On a bien $c_n \geq 0$, on a vu que $c_n = \Theta(n)$ et on montre sans difficulté que $(C_{\text{division}}(n))_{n\geq 1}$ et $(C_{\text{fusion}}(n))_{n\geq 1}$ sont croissantes et donc $(c_n)_{n\geq 1}$ également. Pour une liste de taille $n \geq 2$, on entre dans le troisième cas du filtrage de la fonction `tri_fusion`, dont le coût au pire est celui de la division, de la fusion et [1] de deux appels récursifs sur deux listes de tailles $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$. On a alors bien une équation de récurrence de la forme :
$$C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + c_n \tag{1}$$
avec $a = b = 1$.
4. Pas de question.
5. On a pour $j \in \mathbb{N}^*$, $C(2^j) = 2C(2^{j-1}) + \Theta(2^j)$ et donc, en divisant par 2^j, on obtient $v_j - v_{j-1} = \Theta(1)$, puis, en sommant de 1 à k, $v_k = \Theta(k)$.
6. On a donc $C(n) = 2^k v_k = \Theta(2^k k) = \Theta(n \log n)$.
7. Montrons par récurrence sur $n \in \mathbb{N}^*$ que la suite $(C(k))_{k\in[\![1,n]\!]}$ est croissante. Pour initialiser, avec $n = 1$, la suite comporte un seul terme et il n'y a rien à faire. Pour $n = 2$, $C(2) = 2C(1) + c_2 \geq C(1)$. Supposons donc le résultat vrai jusqu'à un rang $n - 1 \geq 2$ et montrons que la suite $(C(k))_{k\in[\![1,n]\!]}$ est croissante. Comme la suite $(C(k))_{k\in[\![1,n-1]\!]}$ est croissante, il suffit de montrer que $C(n) \geq C(n-1)$. Comme $n - 1 \geq 2$, on a $n > \lceil n/2 \rceil \geq \lceil (n-1)/2 \rceil \geq 1$ et $n > \lfloor n/2 \rfloor \geq \lfloor (n-1)/2 \rfloor \geq 1$ et donc, par hypothèse de récurrence, $C(\lceil n/2 \rceil) \geq C(\lceil (n-1)/2 \rceil)$ et $C(\lfloor n/2 \rfloor) \geq C(\lfloor (n-1)/2 \rfloor)$. Comme de plus $c_n \geq c_{n-1}$, on a donc $C(n) \geq C(\lceil (n-1)/2 \rceil) + C(\lfloor (n-1)/2 \rfloor + c_{n-1} = C(n-1)$ ce qui enclenche la récurrence.
8. Soit $n \in \mathbb{N}^*$, il existe $k \in \mathbb{N}$ tel que $2^k \leq n < 2^{k+1}$ (on prend $k = \lfloor \log_2 n \rfloor$). Par croissance de $(C(n))_{n\in\mathbb{N}^*}$, on a $C(2^k) \leq C(n) \leq C(2^{k+1})$, donc $C(n) = \Omega(C(2^k)) = \Omega(n \log n)$ et $C(n) = O(C(2^{k+1})) = O((n+1)\log(n+1)) = O(n \log n)$. Finalement, $C(n) = \Theta(n \log n)$.
9. Cette complexité est bien, bien meilleure. Par exemple, pour une liste de taille 10^6 sur un ordinateur à un milliard d'opérations par seconde on passe de 1/4 h (pour le tri par sélection ou insertion) à 10ms !

1. Plus un éventuel coût constant que l'on peut intégrer dans c_n.

II Autour du tri rapide

1. La fonction `partition` est de type `'a list -> 'a -> 'a list * 'a * 'a list` et la fonction `tri` de type `'a list -> 'a list`.
2. Les étapes du calcul avec les appels récursifs sont détaillées sur la figure 1.

☞ Remarque 2

On ne connaît pas l'ordre dans lequel `tri petits` et `tri grands` vont être effectués. Cela peut dépendre de la machine, de la version de CAML*, etc.*

3. L'appel `partition liste pivot` partitionne la liste `liste` en deux sous-listes : les éléments plus petits ou égaux à `pivot`, qui sont renvoyés dans la liste `petits` ; et ceux strictement plus grands qui sont renvoyés dans la liste `grands`. Il est immédiat que `milieu` est toujours égal à `pivot` ce que l'on prouvera formellement dans la question suivante. La fonction `tri`, comme son nom l'indique, va trier la liste par ordre croissant. L'appel `tri liste` commence par partitionner la liste, puis trie récursivement les deux sous-listes. Comme les deux sous-listes sont triées, que les éléments de `tri petits` sont inférieurs à `milieu` et que tous ceux-ci sont inférieurs à `tri milieu`, la concaténation renvoyée est bien triée.
4. On retrouve bien les trois étapes de l'approche *diviser pour régner* :
   - On partitionne le problème initial en sous-problèmes de tailles plus petites, ici deux, de tailles divisées par environ 2. C'est le rôle de la fonction `partition` (diviser) ;
   - On résout récursivement les deux sous-problèmes (régner) ;
   - On combine le résultat des sous-appels récursifs. Ici, c'est très simple puisque l'on peut directement concaténer dans l'ordre.
5. Si $n = 0$, `p` est une séquence vide. On note g de taille r et d de taille s, les deux séquences telles que `(g, v, d) = (partition e p)`. Il s'agit du cas de base de la fonction récursive `partition`, et donc g et d sont vides et $r = s = n = 0$, ce qui montre (c) et (d). On a également, dans le cas de base $v = e$ ce qui donne (a). Les propriétés (b), (e) et (f) sont vérifiées puisque $\forall i \in \emptyset$, $\mathcal{P}(i)$ est toujours vrai quelle que soit la proposition $\mathcal{P}$.
6. On a choisi les notations pour que l'hypothèse de récurrence, appliquée ici sur p de taille n et e, s'écrive exactement comme la propriété de l'énoncé :

OCaml

```
tri <-- [3; 1; 4; 2]
    (* partition queue tete) *)
    partition <-- [1; 4; 2] 3
        partition <-- [4; 2] 3
            partition <-- [2] 3
                partition <-- [] 3
                partition --> ([], 3, [])
            partition --> ([2], 3, [])
        partition --> ([2], 3, [4])
    partition --> ([1; 2], 3, [4])
    (* tri grands *)
    tri <-- [4]
        partition <-- [] 4  (* partition queue tete) *)
        partition --> ([], 4, [])
        tri <-- []  (* tri grands *)
        tri --> []
        tri <-- []  (* tri petits *)
        tri --> []
    tri --> [4]
    (* tri petits *)
    tri <-- [1; 2]
        partition <-- [2] 1  (* partition queue tete) *)
            partition <-- [] 1
            partition --> ([], 1, [])
        partition --> ([], 1, [2])
        tri <-- [2]  (* tri grands *)
            partition <-- [] 2  (* partition queue tete) *)
            partition --> ([], 2, [])
            tri <-- []  (* tri grands *)
            tri --> []
            tri <-- []  (* tri petits *)
            tri --> []
        tri --> [2]
        tri <-- []  (* tri petits *)
        tri --> []
    tri --> [1; 2]
tri --> [1; 2; 3; 4]
```

FIGURE 1 – Étapes du calcul de `tri [3; 1; 4; 2]` pour la question 2.

(a) $e = v$
(b) $\forall i \in [\![1, n]\!],\ |p|_{p_i} = |g|_{p_i} + |d|_{p_i}$
(c) $0 \leq s \leq n$ et $0 \leq r \leq n$
(d) $n = s + r$
(e) $\forall i \in [\![1, r]\!],\ g_i \leq v$
(f) $\forall i \in [\![1, s]\!],\ v < d_i$

7. On suppose que $p'_{n+1} \leq v$ c'est donc le premier cas de l'expression conditionnelle qui est évalué et donc $g' = (p'_{n+1}, g_r, \ldots, g_1)$ est de taille $r' = r + 1$, $d' = d = (d_s, \ldots, d_1)$ de taille $s' = s$ et $v = v'$.
   - Par hypothèse de récurrence $e = v$ donc $e = v'$. On a ainsi (a).
   - Soit $i \in [\![1, n+1]\!]$, on veut montrer $|p'|_{p'_i} = |g'|_{p'_i} + |d'|_{p'_i}$. Si $p'_i = p'_{n+1}$, alors $|d'|_{p_{n+1}} = |d|_{p_{n+1}} = 0$ car $p'_{n+1} \leq v$ et le point (f) de l'hypothèse de récurrence impose $\forall j \in [\![1, s]\!],\ v < d_j$. On a donc $|p'|_{p'_{n+1}} = 1 + |p|_{p'_{n+1}} \overset{HR}{=} 1 + |g|_{p'_{n+1}} + |d|_{p'_{n+1}} = 1 + |g|_{p'_{n+1}} = |g'|_{p'_{n+1}} = |g'|_{p'_{n+1}} + |d'|_{p'_{n+1}}$. Si $p'_i \neq p'_{n+1}$ alors $|p'|_{p'_i} = 1 + |p|_{p'_i} \overset{HR}{=} |g|_{p'_i} + |d|_{p'_i} = |g'|_{p'_i} + |d'|_{p'_i}$. Ainsi, on a bien (b).
   - On a $0 \leq s' = s < s + 1 \leq n + 1$ et $0 \leq r' = r + 1 \leq n + 1$ et $n + 1 = s + r + 1 = s' + r'$ et donc (c) et (d).
   - Soit $i \in [\![1, r']\!]$. Si $i \leq r$, alors $g_i \leq v$ par hypothèse de récurrence. Si $i = r + 1$, alors $g_i = p'_{n+1} \leq v$ ce qui montre (e).
   - Le cas (f) de l'hypothèse de récurrence se transmet à l'identique.

   On a donc (laborieusement) montré que H_{n+1} était vérifiée, si $p'_{n+1} \leq v$.
8. On remarque simplement que le cas $p'_{n+1} > v$ se traite de manière analogue, sans le détailler, et donc H_{n+1} est vérifiée dans tous les cas, ce qui assure l'hérédité. Le raisonnement par récurrence permet donc bien de conclure que la propriété H_n est vraie pour tout $n \in \mathbb{N}$ et que la fonction `partition` se comporte comme nous l'avons expliqué de manière informelle précédemment.
9. On note P_m cette propriété et on effectue une récurrence sur l'entier $m \in \mathbb{N}$.
   Initialisation : On suppose $m = 0$. On se donne une séquence p de taille m. Soit la séquence r de taille n telle que `r = tri p`. Dans ce cas p est la séquence vide et il en est donc de même pour r et on a directement (a), (b) et (c).
   Hérédité : Soit $m \in \mathbb{N}$. On suppose la propriété P_k vérifiée pour tout $0 \leq k \leq m$. Soit une séquence $p' = (p'_{m+1}, p'_m, \ldots, p'_1)$ de taille $m + 1$ et la séquence $q = (q_n, \ldots, q_1)$ telle que `q = tri p'`. L'appel `tri p'` engendre l'appel `partition v p` où $e = p'_{m+1}$ et $p = (p_m, \ldots, p_1) = (p'_m, \ldots, p'_1)$. Le retour de `partition v e` est $g = (g_r, \ldots, g_1)$ de taille r, $d = (d_s, \ldots, d_1)$ de taille s et v. D'après la question précédente, g, d et e vérifient :

(a) $e = v$
(b) $\forall i \in [\![1, n]\!],\ |p|_{p_i} = |g|_{p_i} + |d|_{p_i}$
(c) $0 \leq s \leq n$ et $0 \leq r \leq n$
(d) $n = s + r$
(e) $\forall i \in [\![1, r]\!],\ g_i \leq v$
(f) $\forall i \in [\![1, s]\!],\ v < d_i$

On peut donc appliquer l'hypothèse de récurrence à `tri g` et `tri d` qui renvoient respectivement les séquences g' et d' de taille s' et r' telles que

(a) $r' = r$ et $s' = s$
(b) $\forall i \in [\![1, r]\!], |g|_{g_i} = |g'|_{g_i}$ et $\forall i \in [\![1, s]\!], |d|_{d_i} = |d'|_{d_i}$
(c) $\forall i \in [\![1, r-1]\!],\ g'_i \leq g'_{i+1}$ et $\forall i \in]\!]1, s-1[\![,\ d'_i \leq d'_{i+1}$.

La séquence `q` est alors $(d'_1, \ldots, d'_s, v, g'_1, \ldots, g'_r)$ de taille $n = r + s + 1$
et on vérifie directement que

(a) $m + 1 = n$
(b) $\forall i \in [\![1, m+1]\!], |q|_{p_i} = |p|_{p_i}$
(c) $\forall i \in [\![1, m]\!],\ q_i \leq q_{i+1}$.

Conclusion : On a montré par récurrence la correction de la fonction `tri` en fonction de la taille de la séquence argument.

10. Notons n la taille de la liste reçue en argument pour `partition`. Si $n = 0$, la fonction termine clairement. Si $n \in \mathbb{N} \setminus \{0\}$, il y a un unique appel récursif sur une liste de taille strictement inférieure à n, les autres opérations ne posant pas de problème de terminaison. Ceci montre que `partition` termine sur toute entrée. Le cas de la fonction `tri` est similaire. Si $n = 0$ la terminaison est claire. Si $n \in \mathbb{N} \setminus \{0\}$, il y a un nombre fini d'appels récursifs (deux) sur des listes de tailles strictement inférieures à n et les autres opérations effectuées, en particulier l'appel à `partition`, terminent bien.
11. Pour une liste de taille $n \in \mathbb{N}$, il y a exactement n appels récursifs et donc n comparaisons. Ainsi, pour tout $n \in \mathbb{N}$, $P(n) = n$.
12. On a immédiatement $T(0) = 0$. Pour une liste de taille 1 le nombre de comparaisons est $P(1-1) = 0$ et donc $T(1) = 0$.
13. Si une liste est triée par ordre croissant et que tous les éléments sont distincts, le plus petit élément est choisi comme pivot puisque l'on prend le premier. L'appel `partition queue pivot` découpe la liste en `([], pivot, queue)` puisque tous les éléments sont strictement plus grands que le pivot. Ainsi, les listes restent triées par ordre strictement croissant pour tous les appels récursifs suivants et le pivot est toujours le plus petit élément de la liste. On peut aussi prendre la liste triée par ordre décroissant (non nécessairement strictement).
14. Comme l'une des deux listes est vide, on a
$$T(n) = T(n-1) + T(1) + (n-1) = T(n-1) + n - 1$$
15. On écrit $T(n) - T(n-1) = n - 1$ et par télescopage et avec $T(0) = 0$ on obtient le résultat demandé.
16. Notons $C(s)$ le nombre de comparaisons effectuées par `tri` sur une séquence $s = (s_n, \ldots, s_1)$, avec $n \geq 1$. Notons $g = (g_p, \ldots, g_1)$ et $d = (d_{n-p-1}, \ldots, d_1)$ les listes renvoyées par un appel à `partition` sur $(s_{n-1}, \ldots, s_1)$ et s_n comme pivot. On a $C(s) = C(g) + C(d) + p + (n-p-1) = C(g) + C(d) + n - 1$. Par définition de la complexité dans le pire des cas $C(g) \leq T(p)$ et $C(d) \leq T(n-p-1)$. Ainsi
$$C(s) \leq T(p) + T(n-p-1) + n - 1 \leq \max_{0 \leq p \leq n-1} (T(p) + T(n-p-1) + n - 1)$$
Ce dernier terme ne dépendant que de la taille de s, on a donc
$$T(n) \leq \max_{0 \leq p \leq n-1} (T(p) + T(n-p-1) + n - 1)$$
Ce maximum étant atteint pour une liste triée décroissante, il y a en fait égalité.
17. Pour $n = 1$ on a $T(1) = 0 = \frac{n(n-1)}{2}$. Pour $n \geq 2$, supposons le résultat établi pour $0 \leq k \leq n-1$ et montrons-le au rang n. D'après la question précédente et par hypothèse de récurrence,
$$\begin{aligned} T(n) &= \max_{0 \leq p \leq n-1} (T(p) + T(n-p-1) + n - 1) \\ &= \max_{0 \leq p \leq n-1} \left(\frac{p(p-1)}{2} + \frac{(n-p-1)(n-p-2)}{2} + n - 1 \right) \end{aligned}$$
La formule reste vraie pour $p = 0$. La fonction quadratique $x \mapsto x(x-1) + (n-x-1)(n-x-2)$ sur $[0, n-1]$ atteint son minimum en $\frac{n-1}{2}$ et son maximum à ses deux bornes, qui vaut $(n-1)(n-2)$, ce que l'on montre avec une étude de fonction. On a donc $T(n) = \frac{(n-1)(n-2)}{2} + n - 1 = \frac{n(n-1)}{2}$ et l'hérédité est prouvée. Le raisonnement par récurrence permet de conclure que pour tout $n \in \mathbb{N}^*$ (et même $n \in \mathbb{N}$) on a $T(n) = \frac{n(n-1)}{2}$.
18. Soit $n \geq 1$. On a
$$\sum_{k=0}^{n-1} \left(T_{\text{moy}}(k) + T_{\text{moy}}(n-k-1) \right) = 2 \sum_{k=0}^{n-1} T_{\text{moy}}(k)$$

et donc

$$nT_{\text{moy}}(n) = 2\sum_{k=0}^{n-1} T_{\text{moy}}(k) + n - 1$$

On a ainsi

$$\begin{aligned}
nT_{\text{moy}}(n) + (n-1)T_{\text{moy}}(n-1) &= 2T_{\text{moy}}(n-1) + n(n-1) - (n-1)(n-2)\\
nT_{\text{moy}}(n) + (n+1)T_{\text{moy}}(n-1) &= 2(n-1)\\
\frac{T_{\text{moy}}(n)}{n+1} + \frac{T_{\text{moy}}(n-1)}{n} &= \frac{2(n-1)}{n(n+1)}
\end{aligned}$$

19. On part de la relation précédente, on somme pour obtenir une série télescopique, et on décompose la fraction rationnelle de droite en éléments simples.

$$\begin{aligned}
\sum_{k=1}^{n}\left(\frac{T_{\text{moy}}(k)}{k+1} + \frac{T_{\text{moy}}(k-1)}{k}\right) &= \sum_{k=1}^{n}\frac{2(k-1)}{k(k+1)}\\
\frac{T_{\text{moy}}(k)}{k+1} + \frac{T_{\text{moy}}(0)}{1} &= \sum_{k=1}^{n}\left(\frac{-2}{k} + \frac{4}{k+1}\right)\\
\frac{T_{\text{moy}}(k)}{k+1} &= -2\sum_{k=1}^{n}\frac{1}{k} + 4\sum_{k=1}^{n}\frac{1}{k+1}\\
\frac{T_{\text{moy}}(n)}{n+1} &= -2\sum_{k=1}^{n}\frac{1}{k} + 4\left(\sum_{k=0}^{n-1}\frac{1}{k+1} + \frac{1}{n+1} - 1\right)\\
\frac{T_{\text{moy}}(n)}{n+1} &= 2\sum_{k=1}^{n}\frac{1}{k} + \frac{4}{n+1} - 4\\
T_{\text{moy}}(n) &= 2(n+1)\Theta(\log n) + 4 - 4(n+1)\\
T_{\text{moy}}(n) &= \Theta(n\log n)
\end{aligned}$$

III Problème : représentation d'images par des arbres quaternaires

Question III.6 Ce qui est demandé ici est de formaliser dans le langage mathématique la définition **III.5**. Je propose de formaliser les quatre axiomes :

$$\begin{aligned}
&\text{AX}_1(a) \equiv \forall n \in \mathcal{N}(a),\ (\mathcal{X}(n), \mathcal{Y}(n), \mathcal{T}(n)) \in (\mathbb{N}^*)^3\\
&\text{AX}_2(a) \equiv \forall d \in \mathcal{D}(a),\ \forall o \in \{\text{SO}, \text{SE}, \text{NO}, \text{NE}\},\ \mathcal{T}(f_o(d)) = \mathcal{T}(d)/2\\
&\text{AX}_3(a) \equiv \forall d \in \mathcal{D}(a),\ \begin{cases}
\mathcal{X}(f_{\text{SO}}(d)) = \mathcal{X}(d) \wedge \mathcal{Y}(f_{\text{SO}}(d)) = \mathcal{Y}(d)\\
\mathcal{X}(f_{\text{SE}}(d)) + \mathcal{T}(f_{\text{SE}}(d)) = \mathcal{X}(d) \wedge \mathcal{Y}(f_{\text{SE}}(d)) = \mathcal{Y}(d)\\
\mathcal{X}(f_{\text{NO}}(d)) = \mathcal{X}(d) \wedge \mathcal{Y}(f_{\text{NO}}(d)) + \mathcal{T}(f_{\text{NO}}(d)) = \mathcal{Y}(d)\\
\mathcal{X}(f_{\text{NE}}(d)) + \mathcal{T}(f_{\text{NE}}(d)) = \mathcal{X}(d) \wedge \mathcal{Y}(f_{\text{NE}}(d)) + \mathcal{T}(f_{\text{NE}}(d)) = \mathcal{Y}(d)
\end{cases}\\
&\text{AX}_4(a) \equiv \forall d \in \mathcal{D}(a),\ \exists o_1, o_2 \in \{\text{SO}, \text{SE}, \text{NO}, \text{NE}\},\ \exists b_1 \in \mathcal{B}(f_{o_1}), \exists b_2 \in \mathcal{B}(f_{o_2}),\\
&\qquad o_1 \neq o_2 \wedge \mathcal{C}(b_1) \neq \mathcal{C}(b_2)
\end{aligned}$$

On a alors

$$\text{VAQ}(a) \equiv \text{AX}_1(a) \wedge \text{AX}_2(a) \wedge \text{AX}_3(a) \wedge \text{AX}_4(a)$$