

Conception Formelle

TD-TP : Un peu de théorie et de pratique.

Kristo DHIMA, Garion GOUBARD

2021-2022

Ce travail est à réaliser en binôme. La date de rendu est fixée au 1^{er} mai 2022.

Ce sujet (à part le premier exercice) est librement inspiré de l'algorithme Brute Force pour le tsp vu dans le cours de TAP de L3¹. Rassurez-vous, on va y aller pas à pas et ne démontrer que le plus simple.

Les exercices sont par difficulté croissante. Les deux premiers demandent uniquement d'appliquer le calcul de WP et de trouver des invariants de boucle. Les réaliser parfaitement devrait valoir autour de 9. Le troisième demande de produire un code correct et de le démontrer, et de comprendre un prédicat. Avec les trois premiers exercices parfaits, on devrait arriver vers 15. Le quatrième est plus technique, il demande de formaliser un contrat informel et déterminer les bons invariants de boucle est moins immédiat, mais devrait être suffisamment guidé pour rester faisable. Le cinquième est plus exploratoire, et devrait donc être plus vu comme un bonus (la première question devrait cependant rester accessible).

Structure du devoir et rendu Le devoir est constitué du présent document, ainsi que de fichiers de code à compléter en parallèle des questions posées. Le code fourni est découpé ainsi : un fichier `.c` et `.h` pour chaque fonction, et un fichier `formalism.h` pour les prédicats. Ces fichiers sont à remplir au fur et à mesure. `formalism.h` contient uniquement deux prédicat à remplir (à l'exercice 3), le reste vous est donné. Il contient des lemmes qu'on admettra.

Pour répondre aux questions, vous avez deux choix :

- Soit compléter le présent document `.tex` en plaçant vos réponses dans les blocs `solutionorbox` prévues à cet effet. Pour cela, référez-vous aux macros définies dans les sujets de TD.
- Soit produire un document pdf par d'autres moyens (autre logiciel, scan), tant que c'est lisible, ça me conviendra.

Vous devrez rendre (sur la page moodle du cours) :

- Le présent document de réponse (ou votre version).
- Le dossier `code` complété. Dans les fichiers qui le composent, vous pourrez si besoin rajouter des commentaires expliquant comment prouver certaines spécifications si ce n'est pas immédiat, ou expliquer ce qui vous bloque.

Rappels de logique : On rappelle que :

- $p \Rightarrow q \equiv \neg p \vee q$
- $(p \wedge q) \Rightarrow r \equiv p \Rightarrow q \Rightarrow r$.

Vous aurez à manipuler des expressions comprenant des \forall . Pour ces expressions-là, lorsque vous appliquez un pas de WLP, grossièrement il vous faut découper votre formule

1. <https://dept-info.labri.fr/~gavoille/UE-TAP/>

entre partie modifiée et partie non-modifiée, et appliquer le calcul. Plus concrètement ici, ce que vous devriez avoir, c'est quelque chose du genre :

$$\begin{aligned} & \text{WLP}(t[i] = x, \forall j; 0 \leq j \leq i \implies \varphi(t[j])) \\ & \equiv \text{WLP}(t[i] = x, (\forall j; 0 \leq j < i \implies \varphi(t[j])) \wedge \varphi(t[i])) \\ & \equiv (\forall j; 0 \leq j < i \implies \varphi(t[j])) \wedge \varphi(x) \end{aligned}$$

La règle la plus générale d'où vient cela est la suivante : pour φ et ψ des formules quelconques :

$$\forall j; \varphi(j) \equiv \forall j; \psi(j) \implies \varphi(j) \wedge \forall j; \neg \psi(j) \implies \varphi(j);$$

Un autre point, c'est «soyez paresseux» : si vous arrivez à un moment sur un calcul équivalent à $\perp \Rightarrow \text{WLP}(i - j, \Phi)$, il est inutile de calculer $\text{WLP}(i - j, \Phi)$ pour déterminer que l'implication est vraie.

Vous devrez aussi prouver des prédicats de la forme **Toto{Pre,Post}(...)**. Pour prouver ce genre de prédicats, rappelez-vous, que Pre correspond au début du programme : il ne sera donc pas modifié par le calcul de WP. Post, au contraire, correspond à la fin du programme et donc au label Here : lors du calcul de WP, les valeurs correspondant à la position courante sont modifiées (donc vous gardez le label Here, où vous mettez la ligne de la position courante). Ça correspond simplement au calcul normal de WP où on laisse la formule abstraite : cette formule contient un label implicite qui est bel et bien Here. Une bonne manière de voir cela, est que, lorsque que vous remplacez le nom du prédicat par sa définition, vous remplacerez les **at(x,Pre)** par $\text{old}(x)$, et les **at(x,Post)** par x , et effectuez le calcul normalement.

On attendra que vos calculs de WP soient suffisamment détaillés, mais vous pouvez sauter quelques étapes si elles sont faciles (comme par exemples, appliquer plusieurs substitutions d'un coup). Évidemment, tant que cela est correct.

Comme dit souvent, pour les justification de vérité de formule, on n'attendra pas de preuves formelles, mais des justifications convaincantes (i.e., qui n'oublient pas de cas, mais on restera tolérant sur la forme). En gros, quand vous aurez une implication, une possibilité sera de dire un truc du genre «l'implication est vraie car telle et telle partie de la partie gauche impliquent bien la partie droite». Un «ben oui» (ou plutôt un remplacement par \top) sera admissible pour des propriétés du genre $\perp \Rightarrow \varphi$, $\varphi \Rightarrow \top$ ou $a < b \Rightarrow a \leq b$. Cependant, pour faire cela de manière convaincante, vous auriez intérêt à simplifier vos formules avant.

Dans le présent sujet, on donne une version normalisée du code pour vous faciliter preuves (if then else développés, un seul return, que des while). Évidemment, vous pouvez coder autrement, mais il sera plus aisé de faire ces restrictions sur papier.

Attention : on manipule beaucoup de unsigned int. En pratique, dans vos formules, *quantifiez sur des integer, et non des int*.

Un mot sur les preuves : À partir de l'exercice 3, certaines des preuves commencent à être difficiles pour les solveurs, aussi faites bien les trois points suivants :

- Vérifiez que le solveur Z3 est activé (il parvient à démontré des propriétés où alt-ergo échoue).
- Si certaines propriétés ne sont pas prouvées, retentez la preuve : quand les solveurs tentent trop de preuves en même temps, il arrive que certaines timeout pour de mauvaises raisons. Les relancer peut régler le problème.
- Si cela ne marche toujours pas (et que vous avez confiance en la propriété), cliquez sur le nom du but, puis sur la tactique «filter». Cela peut parfois régler le problème.

Si rien de tout cela ne marche, vous avez probablement oublié de spécifier certaines hypothèses (ou votre propriété est fausse), donc reprenez votre stylo.

Exercice 1 : Swap par xor

On considère une variante du swap sans variable auxiliaire, en utilisant l'opération xor (notée \wedge). On va pour l'occasion, augmenter les opérations arithmétiques que l'on s'autorise dans notre langage jouet C1 avec l'opération xor, avec la même sémantique qu'en C. En particulier, on rappelle (et c'est la seule propriété dont on aura besoin) que pour tous entiers n, m , $(n \wedge m) \wedge n = m$, et que cette opération est commutative (et associative).

À noter que cette version du swap n'est en réalité pas vraiment plus efficace que celle avec mémoire auxiliaire. On va de plus voir qu'elle dispose d'un léger défaut supplémentaire.

```

1  /*@ ensures *a == \old(*b);
2      ensures *b == \old(*a);
3  */
4  void swap(int *a, int *b)
5  {
6      *a = (*a) ^ (*b);
7      *b = (*a) ^ (*b);
8      *a = (*a) ^ (*b);
9      return;
10 }
```

- Si ce n'est pas déjà fait et que vous rendrez le présent .tex, mettez vos noms dans la balise author situé en haut de ce document (celle où il y a écrit un message assez explicite). (0 points)
- Calculez $WP(\text{swap}, \psi)$ pour ψ la post-condition fournie, et déduisez-en un triplet de HOARE valide.

Réponse:

$$\begin{aligned}
 WP(\text{swap}, \psi) &\equiv WP(6, WP(7, WP(8, \psi))) \\
 &\equiv WP(6, WP(7, WP(8, (*a == \text{old}(*b)) \wedge (*b == \text{old}(*a))))) \\
 &\equiv WP(6, WP(7, ((*a \wedge *b) == \text{old}(*b)) \wedge (*b == \text{old}(*a)))) \\
 &\equiv WP(6, (*a \wedge (*a \wedge *b) == \text{old}(*b)) \wedge ((*a \wedge *b) == \text{old}(*a))) \\
 &\equiv WP(6, (*b == \text{old}(*b)) \wedge ((*a \wedge *b) == \text{old}(*a))) \\
 &\equiv (*b == \text{old}(*b)) \wedge ((*a \wedge *b) \wedge *b == \text{old}(*a)) \\
 &\equiv (*b == \text{old}(*b)) \wedge (*a == \text{old}(*a))[\text{old}(s) \leftarrow s | s \in C1] \\
 &\equiv *b == \text{old}(*b) \wedge *a == \text{old}(*a) \\
 &\equiv \top
 \end{aligned}$$

Donc $\{\top\} \text{swap} \{\psi\}$ est un triplet de HOARE valide.

- Évidemment, on a ici une correction partielle, dans le sens où on n'a pas tenu compte des comportements indéterminés. Quelle information manque-t'il pour assurer qu'il n'y aura pas d'erreur à l'exécution ?

Réponse:

Il faut s'assurer que les pointeurs a et b sont valides en lecture et en écriture.

- (d) Quel autre problème voyez-vous à notre preuve ? Quelle hypothèse cachée faut-il rajouter pour qu'elle soit correcte ?

Réponse:

Il faut s'assurer que les adresses des pointeurs a et b sont différentes. En fait, si les adresses sont les mêmes, nous ne pouvons pas échanger les valeurs, donc la post-condition ne peut pas être validée par Frama-C.

- (e) Compléter le fichier `swap-xor.h` de manière à ce que Frama-C puisse démontrer la post-condition fournie, en ajoutant les assertions RTE. Il est bien évidemment possible (et encouragé) de répondre aux questions précédentes grâce à celle-ci.

Exercice 2 : Reverse

On veut maintenant une fonction qui va inverser une portion consécutive d'un tableau. Le but de cet exercice est de formaliser le contrat et de le prouver. On considère le code de la fonction, ainsi que la précondition que l'on souhaite démontrer.

```

1  /*@
2  ensures Phi: \forallall integer k;
3             i <= k <= j ==> t[k] == \old(t[j-k+i]);
4  */
5  void reverse(int *t, unsigned int i, unsigned int j,
6             unsigned int n)
7  {
8      while (i < j)
9      {
10         int aux = t[i];
11         t[i] = t[j];
12         t[j] = aux;
13         i++;
14         j--;
15     }
16     return;
17 }
```

- (a) Quelle condition doit-être vraie à la sortie de la boucle (question sur 0 points, c'est quand même trivial) ?

Réponse:

C'est la condition Φ (la post-condition) qui doit être vrai à la sortie de la boucle.

- (b) Calculez $WLP(10 - 14, I)$ pour un invariant I quelconque.

Réponse:

$$\begin{aligned}
\text{WLP}(10 - 14, I) &\equiv \text{WLP}(10 - 13, I[j \leftarrow j - 1]) \\
&\equiv \text{WLP}(10 - 12, I[j \leftarrow j - 1][i \leftarrow i + 1]) \\
&\equiv \text{WLP}(10 - 11, I[j \leftarrow j - 1][i \leftarrow i + 1][t[j] \leftarrow aux]) \\
&\equiv \text{WLP}(10, I[j \leftarrow j - 1][i \leftarrow i + 1][t[j] \leftarrow aux][t[i] \leftarrow t[j]]) \\
&\equiv I[j \leftarrow j - 1][i \leftarrow i + 1][t[j] \leftarrow aux][t[i] \leftarrow t[j]][aux \leftarrow t[i]]
\end{aligned}$$

- (c) Proposez des invariants de boucle pour votre boucle. Vous devriez en avoir qui détermine un ordre sur les variables i, j et les comparent avec leurs valeurs initiales (un seul peut suffire, mais vous pouvez bien sûr le découper), ainsi qu'un invariant qui donne une relation entre i et j (i.e., il ne suffit pas de les encadrer, il faut décrire comment elles évoluent l'une par rapport à l'autre). Vous devriez également avoir des invariants qui parlent de la partie du tableau déjà modifiée, et d'autres qui parlent de la manière non encore modifier. Au total, 4 invariants peuvent suffire, mais ça dépend de comment vous les écrivez (en compressant on peut faire 3, et on peut les découper en plus). Vous leur donnerez des noms (par exemple I_1, I_2 , etc, ou des noms plus explicites). On appellera I leur conjonction.

Justifiez que $\neg i < j \wedge I \Rightarrow \text{WLP}(16, \varphi)$.

Pour chaque I_i , précisez $\text{WLP}(10 - 14, I_i)$ (utilisez le calcul de la question b), et justifiez que $i < j \wedge I \Rightarrow \text{WLP}(10 - 14, I_i)$.

Réponse:

Si $\neg i < j$ est vrai, alors pendant l'appel de la fonction le programme ne va pas entrer dans la boucle while. Par conséquent, la plus faible précondition est $\text{WLP}(16, \varphi)$. Nous avons donc bien $\neg i < j \wedge I \Rightarrow \text{WLP}(16, \varphi)$.

Les invariants sont les suivants :

- $I_1 : 0 \leq \backslash at(i, Pre) \leq i \leq j + 1 \leq \backslash at(j, Pre) + 1 \leq n$
- $I_2 : i == \backslash at(i, Pre) + \backslash at(j, Pre) - j$
- $I_3 : \forall \text{ integer } k, i \leq k \leq j \Rightarrow t[k] == \backslash at(t[k], Pre)$
- $I_4 : \forall \text{ integer } k, (\backslash at(i, Pre) \leq k < i) \Rightarrow t[k] == \backslash at(t[j - k + i], Pre)$
- $I_5 : \forall \text{ integer } k, j < k \leq \backslash at(j, Pre) \Rightarrow t[k] == \backslash at(t[j - k + i], Pre)$

Pour chaque invariant, nous avons :

1. $\text{WLP}(10 - 14, I_1) \equiv 0 \leq \backslash at(i, Pre) \leq i \leq j + 1 \leq \backslash at(j, Pre) + 1 \leq n[j \leftarrow j - 1][i \leftarrow i + 1][t[j] \leftarrow aux][t[i] \leftarrow t[j]][aux \leftarrow t[i]]$
 $\text{WLP}(10 - 14, I_1) \equiv 0 \leq \backslash at(i, Pre) \leq i + 1 \leq j \leq \backslash at(j, Pre) + 1 \leq n$

$i < j \wedge I \Rightarrow \text{WLP}(10 - 14, I_1)$ est vrai car : La valeur de $\backslash at(i, Pre)$ ne change pas, donc on a toujours $0 \leq \backslash at(i, Pre)$; $I_1 \in I$, donc nous avons $\backslash at(i, Pre) \leq i$ et donc $\backslash at(i, Pre) \leq i + 1$; Si $i < j$ alors $i + 1 \leq j$; De la même façon, on montre que $j \leq \backslash at(j, Pre) + 1 \leq n$ est vrai.

2. $\text{WLP}(10 - 14, I_2) \equiv i == \backslash at(i, Pre) + \backslash at(j, Pre) - j[j \leftarrow j - 1][i \leftarrow i + 1][t[j] \leftarrow aux][t[i] \leftarrow t[j]][aux \leftarrow t[i]]$

$$\equiv i + 1 == \backslash at(i, Pre) + \backslash at(j, Pre) - j + 1$$

$$\equiv i == \backslash at(i, Pre) + \backslash at(j, Pre) - j = I_2$$

$i < j \wedge I \Rightarrow \text{WLP}(10 - 14, I_2)$ est vrai, car $\text{WLP}(10 - 14, I_2) \equiv I_2$ et $I_2 \in I$.

3. $\text{WLP}(10 - 14, I_3) \equiv \forall \text{ integer } k, i \leq k \leq j \Rightarrow t[k] == \backslash at(t[k], Pre)[j \leftarrow j - 1][i \leftarrow i + 1][t[j] \leftarrow aux][t[i] \leftarrow t[j]][aux \leftarrow t[i]]$
 $\equiv \forall \text{ integer } k, i + 1 \leq k \leq j - 1 \Rightarrow t[k] == \backslash at(t[k], Pre)$

$i < j \wedge I \Rightarrow \text{WLP}(10 - 14, I_3)$ est vrai car $I_3 \in I$, donc si $\forall \text{ integer } k; i \leq k \leq j \Rightarrow t[k] == \backslash at(t[k], Pre)$ est vrai, nous avons $\forall \text{ integer } k, i + 1 \leq k \leq j - 1 \Rightarrow t[k] == \backslash at(t[k], Pre)$, puisque après un tour de boucle, k est compris dans un interval qui lui même est compris dans l'intervall initial.

4. $\text{WLP}(10 - 14, I_4) \equiv \forall \text{ integer } k, (\backslash at(i, Pre) \leq k < i) \Rightarrow t[k] == \backslash at(t[j - k + i], Pre)[j \leftarrow j - 1][i \leftarrow i + 1][t[j] \leftarrow aux][t[i] \leftarrow t[j]][aux \leftarrow t[i]]$
 $\equiv \forall \text{ integer } k, (\backslash at(i, Pre) \leq k < i + 1) \Rightarrow t[k] == \backslash at(t[j - 1 - k + i + 1], Pre)$
 $\equiv \forall \text{ integer } k, (\backslash at(i, Pre) \leq k < i + 1) \Rightarrow t[k] == \backslash at(t[j - k + i], Pre)$

$i < j \wedge I \Rightarrow \text{WLP}(10 - 14, I_4)$ est vrai, car : $\forall \text{ integer } k, (\backslash at(i, Pre) \leq k < i \Rightarrow t[k] == \backslash at(t[j - k + i], Pre))$ est valide parce que $I_4 \in I$. $t[i] == \backslash at(t[j - i + i], Pre) \equiv t[i] == \backslash at(t[j], Pre)$ est vrai, car avant la boucle nous avons $t[j] == \backslash at(t[j], Pre)$ (grace au fait que $I_3 \in I$) et après la boucle nous avons $t[i] == t[j]$.

5. $\text{WLP}(10 - 14, I_5) \equiv \forall \text{ integer } k, j < k \wedge at(j, Pre) \Rightarrow t[k] == \backslash at(t[j - k + i], Pre)[j \leftarrow j - 1][i \leftarrow i + 1][t[j] \leftarrow aux][t[i] \leftarrow t[j]][aux \leftarrow t[i]]$
 $\equiv \forall \text{ integer } k, j - 1 < k \leq \backslash at(j, Pre) \Rightarrow t[k] == \backslash at(t[j - k + i], Pre)$

$i < j \wedge I \Rightarrow \text{WLP}(10 - 14, I_5)$ est vrai, car : $\forall \text{ integer } k, j < k \leq (\backslash at(j, Pre) \Rightarrow t[k] == \backslash at(t[j - k + i], Pre))$ est valide parce que $I_5 \in I$. $t[i] == \backslash at(t[j - i + i], Pre) \equiv t[i] == \backslash at(t[j], Pre)$ est vrai, car avant la boucle nous avons $t[j] == \backslash at(t[j], Pre)$ (grace au fait que $I_3 \in I$) et après la boucle nous avons $t[i] == t[j]$.

(d) Dédisez-en un triplet de HOARE valide pour votre fonction.

Réponse:

Pour trouver un triplet de HOARE valide, nous devons appliquer $\text{WP}(7, I) = I$. Il faut trouver Φ à partir des conditions pour que ces invariants soient vrai au début de la fonction, donc avant la boucle. Ainsi :

1. Pour $I_1 : 0 \leq \backslash at(i, Pre) \leq i \leq j + 1 \leq \backslash at(j, Pre) + 1 \leq n = 0 \leq i \leq j + 1 \leq n$, il nous faut la précondition $0 \leq i \leq j < n$.

2. Pour $I_2 : i == \backslash at(i, Pre) + \backslash at(j, Pre) - j$ nous n'avons pas besoin de precondition car au debut de la fonction, c'est equivalent a :

$$i == j - j + i \equiv i == i$$

, qui est une totologie.

3. Pour $I_3 : \forall \text{ integer } k; i \leq k \leq j \Rightarrow t[k] == \backslash at(t[k], Pre)$, nous n'avons pas de precondition car c'est deja vrai au debut de la fonction ($t[k] == old(t[k])$).
4. Pour $I_4 : \forall \text{ integer } k; (\backslash at(i, Pre) \leq k < i) \Rightarrow t[k] == \backslash at(t[j - k + i], Pre)$, n'a pas besoin de precondition car il y a une contradiction a la partie gauche de l'implication, puisque nous avons : $\backslash at(i, Pre) \leq k < i \equiv i \leq k < i$ qui est faux, donc l'implication est toujours vraie.
5. Pour $I_4 : \forall \text{ integer } k; j < k \backslash at(j, Pre) \Rightarrow t[k] == \backslash at(t[j - k + i], Pre)$, n'a pas besoin de precondition car, comme dans I_4 , il y a une contradiction a la partie gauche de l'implication, puisque nous avons : $\backslash at(i, Pre) \leq k < i \equiv i \leq k < i$ qui est faux, donc l'implication est toujours vraie.

Donc, le triplet de HOARE $\{ 0 \leq i \leq j < n \}$ reverse $\{\Phi\}$ est un triplet valide.

- (e) Démontrez la terminaison de la fonction en donnant un variant de boucle et en démontrant qu'il décroît à chaque tour de boucle et qu'il est toujours positif (vous pouvez utiliser le calcul de la question b).

Réponse:

Afin de déterminer la terminaison de la fonction, nous utilisons le variant $j - i + 1$.

Puisque nous avons $i < j \wedge I \Rightarrow j - i + 1 \geq 0$, alors nous déduisons

$$i < j \equiv j - i > 0 \equiv j - i + 1 \geq 0$$

Donc, le variant est toujours positif.

En ce qui concerne sa décrémentation, nous appliquons notre variant dans la boucle :

$$\begin{aligned} & \text{WLP}(10 - 14, j - i + 1) - \backslash at(j - i + 1, Pre) \\ &= (j - 1) - (i + 1) + 1 - \backslash at(j - i + 1, Pre) \\ &= j - 1 - i - \backslash at(j - i + 1, Pre) \\ &= j - 1 - i - j + i - 1 = -2 \end{aligned}$$

Donc, notre variant est aussi strictement décroissant.

- (f) Quelles sont les valeurs mémoires modifiées par cette fonction (et la boucle) ? Vous répondrez en donnant les clauses assigns et loop assigns correspondantes.

Réponse:

La boucle modifie les valeurs $i, j, t[i..j]$. Nous avons ajouté ces deux clauses :

1 `assigns t[i..j];`

```
2      loop assigns i,j,t[\at(i,Pre).. \at(j,Pre)] ;
```

- (g) Que manque-t'il comme précondition pour que la fonction ne puisse pas faire d'erreur à l'exécution ?

Réponse:

La précondition qui manque est la vérification de validité en lecture et écriture des pointeurs $t[0..n-1]$ (ou plutôt $t[i..j]$).

- (h) Complétez les fichiers `reverse.c` et `reverse.h` de manière à ce que le contrat reflète tous les points vus ici et que frama-c accepte de prouver le contrat. Vous pouvez bien évidemment vous servir de cette question pour travailler les précédentes (c'est recommandé, d'ailleurs). On attendra bien sûr que toutes les assertions RTE soient démontrée, ainsi qu'une clause `terminates \true`.
- (i) (Bonus) Proposez une version récursive de la fonction et démontrez (avec Framac) qu'elle vérifie le même contrat. En particulier, vous fournirez une clauses `decreases` permettant à la clause `terminates \true` d'être satisfaite (frama-c ne la vérifiera pas, mais moi oui).

Exercice 3 : Permutations minimales et maximales

On s'intéresse maintenant à des fonctions générant des permutations sur l'ensemble $\{0, \dots, n-1\}$, vues comme des tableaux.

Définition 0.1. Une permutation sur $\{0, \dots, n-1\}$ est une fonction P qui à chaque entier $i \in \{0, \dots, n-1\}$ associe un entier $P[i]$ de $\{0, \dots, n-1\}$ et qui est une bijection.

En C, on va stocker une bijection dans un tableau. On définit donc un bijection avec le prédicat suivant.

```
1 predicate isPermutation(int *t, integer n) =
2   \forall integer i; 0 <= i < n ==> 0 <= v[i] < n &&
3   \forall integer i,j; 0 <= i < j < n ==> v[i] != v[j];
```

- (a) Justifiez (rapidement) que ce prédicat est bien équivalent à la définition d'une permutation.

Réponse:

Ce prédicat vérifie que toutes les valeurs de v sont dans l'intervalle $[0, n-1]$ et que toutes les valeurs de v sont différentes. Il s'agit bien d'une bijection, ce qui correspond à la définition d'une permutation.

On va également considérer un ordre entre les permutations. On le définit via les trois prédicats suivants :

```
1 predicate unchangedTab{L,M}(int *s, int *t, integer i,
2 integer n) = \forall integer j; i <= j < n ==>
3   \at(s[j],L) == \at(t[j],M);
4
5 predicate isStrictlyBiggerPerm{L,M}(int *t, int *s,
```



```

6 integer n) = (\exists integer m; 0 <= m <= n-1 &&
7 (\at(t[m],L) < \at(s[m],M) &&
8 unchangedTab{L,M}(t,s,0,m)));
9
10 predicate isBiggerPerm{L,M}(int* t,int* s, integer n) =
11 isStrictlyBiggerPerm{L,M}(t,s,n) ||
12 (unchangedTab{L,M}(t,s,0,n));

```

- (b) Quelles sont, selon cet ordre, les permutations minimales et maximales ? Donnez des prédicats `isMinPerm(int* t, integer n)` et `isMaxPerm(int* t, integer n)` qui ne sont vrais que si le tableau `t` est la permutation minimale (resp. maximale). Complétez ces prédicats dans `formalism.h`. Les deux premiers lemmes de la section exercice 4 devraient être démontrés, mais pas les autres (c'est normal).

Réponse:

```

1 predicate isMinPermL(int* t, integer n) =
2 \forall integer i ; 0 <= i < n - 1
3 ==> \at(t[i], L) < \at(t[i+1], L);;
4
5 predicate isMaxPermL(int* t, integer n) =
6 \forall integer i ; 0 <= i < n - 1
7 ==> \at(t[i], L) > \at(t[i+1], L);;

```

- (c) Implémentez maintenant une fonction qui place dans `P` la permutation minimale, avec le prototype et le contrat partiel suivants. Respectez les styles que l'on sait démontrer sur papier, à savoir un seul point de sortie de la fonction et uniquement des boucles while (pas de for).

```

1 /*@
2 terminates \true;
3 ensures isPermutation(P,n);
4 ensures isMinPerm(P,n);
5 */
6 void initPerm(int* P, unsigned int n);

```

- (d) Déterminez un contrat complet pour cette fonction. Vous aurez évidemment besoin de déterminer des invariants de boucles. On attend à cette question que vous fournissiez les calculs de WLP adéquats.

Réponse:

Les invariants I sont les suivants :

— $I_1 : 0 \leq i \leq n - 1$

— $I_2 : \forall \text{ integer } k; 0 \leq k < i \implies P[k] == k$

$$\begin{aligned} \text{WLP}(14 - 15, I) &\equiv \text{WLP}(14, \text{WLP}(15, I)) \\ &\equiv \text{WLP}(14, I[i \leftarrow i + 1]) \\ &\equiv I[i \leftarrow i + 1][P[i] \leftarrow i] \end{aligned}$$

- (e) Démontrez la terminaison de cette fonction en donnant un variant pour votre boucle, et en démontrant que c'en est un.

Réponse:

Afin de déterminer la terminaison de la fonction, nous utilisons le variant $n - i$.

Puisque nous avons $0 \leq i \leq n$, alors nous avons $n \geq i$, donc $n - i \geq 0$.

Donc, le variant est toujours positif.

En ce qui concerne sa décrémentation, nous appliquons notre variant dans la boucle :

$$\begin{aligned} \text{WLP}(14 - 15, n - i) &= \backslash at(n - i, Pre) \\ &= n - (i + 1) - \backslash at(n - i, Pre) \\ &= n - i - 1 - n + i = -1 \end{aligned}$$

Donc, notre variant est aussi strictement décroissant.

- (f) Votre implémentation de `initPerm` doit valider le contrat précédent dans Frama-C, ainsi que valider les assertions RTE, et comprendre la preuve de terminaison.
- (g) Donnez une implémentation de la fonction dont le contrat partiel et le prototype suivent. Attention, alt-ergo n'arrivera a priori pas à tout démontrer, mais Z3 y arrivera. On ne demande pas de démontrer sur papier cette fonction, vous êtes donc libre sur le style. On attend que les assertions RTE soient démontrées, et que vous démontriez que la fonction termine.

```
1 /*@
2 terminates \true;
3 ensures isMaxPerm(P,n) ==> \result == true;
4 ensures !isMaxPerm(P,n) ==> \result == false;
5 */
6 bool isMaxPerm(int *P, unsigned int n);
```

Cela dit, si vous ne parvenez pas à tout démontrer et souhaitez me montrer sur papier ce que vous essayez de faire et faire des preuves partielles, vous pouvez le faire en suivant :

Réponse:

Nous avons réussi à tout démontrer.

Exercice 4 : NextPermutation

On va maintenant s'intéresser à la fonction qui permet de considérer la prochaine permutation dans l'ordre défini précédemment.

```
1 bool NextPermutation(int *P, unsigned int n)
2 {
3     unsigned int i = n - 1, j;
4     j = n - 1;
5     while (j > 0 && P[j - 1] >= P[j])
6     {
7         j--;
8     }
```

```

9      if (j == 0)
10     {
11         return false;
12     }
13     i = j - 1;
14     while ((j < n) && (P[i] < P[j]))
15         j++;
16     j--;
17     swap(P + i, P + j);
18     reverse(P, i + 1, n - 1, n);
19     return true;
20 }

```

Le but de cet exercice est de déterminer un contrat de fonction pour cette fonction et de le prouver (avec Frama-C). Un mot d'avertissement : sur une telle fonction (et avec le genre de propriété qu'on souhaite montrer), les solveurs vont commencer à suer un peu : certaines propriétés ne seront démontrées qu'avec Z3 (et pas alt-ergo), et il y aura besoin de placer certaines assertions en cours de code pour les aider. En l'occurrence, je vous fournis dans le fichier `NextPermutation.c` les assertions dont j'ai besoin pour démontrer mon contrat (et qui sont sans doute liées au fait qu'on a fait des appels de fonction). Les preuves utilisent les lemmes de la section Exercice 4 de `formalism.h` (qu'on ne demande pas de démontrer).

- (a) Le premier but est donc de formaliser le contrat qui suit :
- Si la fonction reçoit une permutation maximale (i.e., qui satisfait `maxPerm(P,n)`), la fonction renvoie faux et ne modifie rien.
 - Si la fonction reçoit une permutation non-maximale, alors elle renvoie vrai et place dans P une permutation strictement plus grande que celle qui y était à l'entrée de la fonction.
 - Dans tous les cas, il y aura une permutation dans P à la fin de l'appel de la fonction.

Donnez une formalisation de ces post-conditions :

Réponse:

Nous avons séparé les pré-conditions en 3 parties :

- La partie générale, qui contient les pré-conditions qui se passeront dans tous les cas :

```

1  ensures isPermutation(\at(P, Post), n);
2  assigns P[0..n-1];

```

- La partie des pré-conditions qui ne se passeront que si la fonction reçoit une permutation maximale à l'entrée :

```

1  behavior MaxPerm:
2      assumes isMaxPerm{Pre}(P,n);
3      ensures \result == false;
4      assigns \nothing;

```

- La partie des pré-conditions qui ne se passeront que si la fonction reçoit une permutation non-maximale à l'entrée :

```

1  behavior NotMaxPerm:
2      assumes !isMaxPerm{Pre}(P,n);
3      ensures \result == true;
4      ensures isStrictlyBiggerPerm{Pre,Post}(P, P, n);

```

Bon, je ne vais pas vous faire l'insulte de vous demander de calculer les WLP des deux boucles, puisque leur corps n'est pas ultra compliqué.

- (b) Pour la première boucle, déterminez des invariants. Le but est surtout de déterminer ce que fait la boucle. Indication : la boucle s'arrête dès qu'on n'a trouvé un indice j pour lequel $P[j-1] > P[j]$. Que peut-on donc dire sur la fin du tableau si on est en j ? (ceci est une question rhétorique pour que vous trouviez l'invariant). Déterminez ces invariants et justifiez (rapidement) que c'en sont : il faut deux invariants. Déterminez une clause assigns et un variant.

Réponse:

Les invariants sont :

1. $0 \leq j \leq n-1$ Car j est initialisé à $n-1$. Elle est décrementée de 1 à chaque tour de boucle jusqu'à la sortie de boucle, et, dans le pire des cas, nous avons $j == 0$. Dans ce cas là, n'avons pas trouvé de j adéquat.
2. $\forall \text{ integer } k; j < k \leq n-1 \implies P[k-1] \geq P[k]$ Nous utilisons cet invariant car c'est simplement la deuxième condition d'arrêt de la boucle, ce qui doit donc être valide à chaque itération de la boucle. Nous vérifions ainsi que la partie parcourue par la boucle (dans ce cas $P[j..n-1]$) valide bien cette condition.

La clause assign ainsi que le variant choisi sont :

```

1  loop assigns j;
2  loop variant j;

```

- (c) Même question pour la seconde boucle. De la même manière, il faut être capable de décrire ce qui est vrai sur la portion du tableau exploré. Intuitivement, cette boucle place j sur l'élément suivant celui qui est le plus petit élément plus grand que $P[i]$ (c'est pour ça qu'on fait $j-$ après la boucle). Il y a besoin de deux invariants. Donnez la clause assigns et un variant.

Réponse:

Les invariants sont :

1. $i < j \leq n$ Car i est initialisé à $j-1$, donc $i < j$. j augmente de 1 à chaque tour de boucle jusqu'à la sortie de boucle, et, dans le pire des cas, nous avons $j == n$. Dans ce cas là, n'avons pas trouvé de j adéquat.
2. $\forall \text{ integer } k; i < k < j \implies P[i] < P[k]$ Nous utilisons cet invariant car c'est simplement la deuxième condition d'arrêt de la boucle, ce qui doit donc être valide dans chaque itération de la boucle. Nous vérifions

ainsi que la partie parcourue par la boucle(dans ce cas $P[i..j]$) valide bien cette condition.

La clause assign et le variant choisi sont :

```
1 loop assigns j;  
2 loop variant n - j;
```

Le choix du variant $n - j$ est adéquat car initialement nous avons $0 < j \leq n$, et j incrémente à chaque tour de boucle, ce qui fait que $n - j$ décrémente à chaque tour de boucle, car n reste inchangé.

- (d) Terminez les annotations dans le fichier pour démontrer le contrat de cette fonction. En particulier, précisez les requires pertinent, la bonne clause assigns pour le contrat, et la clause terminates qui assure que cette fonction termine. Si vous n'utilisez pas de comportements, alors il vous faut un ensures de plus (qui dit que la permutation n'est pas modifiée si la permutation en entrée est maximale). Si vous utilisez des comportements, il est possible d'avoir des clauses assigns différentes et de se passer de cet ensures.
- (e) Le contrat qu'on a donné ici n'est en réalité pas complet (ce qui manque est explicité dans l'exercice 5). Donnez une fonction qui respecte le même contrat (mais plus simple), mais qui ne répond pas à l'idée intuitive de ce que fait `nextPermutation` (on n'attend pas nécessairement que vous la prouviez avec Frama-C, vous pouvez vous contenter d'expliquer pourquoi elle satisfait le contrat). Indication : cette fonction pourrait systématiquement retourner la même permutation.

Réponse:

METTEZ VOTRE RÉPONSE ICI.

Exercice 5 : tsp

On va maintenant tenter de démontrer une fonction qui implémente un algo Brute Force pour le travelling salesman problem. Pour cela, on va éluder un détail : on admet l'existence d'une fonction `value` qu'on abstrait (techniquement, il faudrait plus d'arguments, mais ici on s'en fiche). On va également avoir besoin de quelques lemmes qu'on admettra (les solveurs n'arrivent pas à les démontrer). On va également ajouter la partie du contrat qui manque à la fonction précédente, qu'on admettra également (enfin, si vous arrivez à le démontrer, je serai impressionné). Tout cela est défini dans `tsp.h` et dans `formalism.h`.

À partir de ça, on considère le code suivant :

```
1 /*@  
2 ensures \forall int* t; \separated(t+(0..n-1), P+(0..n-1))  
3 ==> isPermutation(t,n) ==> value(t,n) >= \result;  
4 */  
5 int tsp(unsigned int n, int *P)  
6 {  
7     initPerm(P, n);  
8     int val = value(P, n);
```

```

9      while (!(isMaxPerm(P, n)))
10     {
11         NextPermutation(P, n);
12         if (val > value(P, n))
13             val = value(P, n);
14     }
15     return val;
16 }

```

Une note : dans le contrat que nous avons fourni, vous noterez la présence d'une clause `\separated`. Sans cela, Frama-C est incapable de démontrer le contrat. Cela est dû au fait qu'on formalise nos permutations comme des tableaux, et non réellement comme des objets abstraits, et que comme on modifie la permutation courante via `NextPermutation`, les solveurs ne sont pas capables de montrer qu'une formule quantifiant universellement sur les permutations n'est pas impactée par l'appel de `NextPermutation`. En précisant qu'on ne regarde que les permutations qui ne recoupent pas P , ils s'en sortent par contre. Ça devrait suffire pour être convaincant, mais ça rend la formalisation un peu plus pénible. Mais je ne vois pas d'autre moyen de quantifier universellement sur des permutations.

Vous aurez bien évidemment besoin de garder cette hypothèse de séparation dans vos invariants.

Frama-C n'arrivera pas à prouver le contrat sans assertions. Celles qui sont déjà placées dans `tsp.c` devraient être suffisantes.

- (a) Ajoutez les annotations de boucle et les préconditions nécessaires pour démontrer la correction partielle de cette fonction.
- (b) Démontrez sur papier que la fonction termine, en donnant un variant de boucle. Ce variant pourra être dans un autre ordre bien fondé que sur \mathbb{N} , $<$.

Je ne demande pas de le faire sur Frama-C (vous pouvez si vous le souhaitez, mais je pense que c'est un peu technique à écrire proprement).

Réponse:

METTEZ VOTRE RÉPONSE ICI.

- (c) (Bonus ++) Frama-C n'arrive pas à démontrer certains des lemmes qu'on a admis. Faites-le vous sur papier.

Réponse:

METTEZ VOTRE RÉPONSE ICI.

- (d) (Bonus ++++) Démontrez que la fonction `nextPermutation` vérifie le contrat supplémentaire qu'on a ajouté à cet exercice. Sur papier, c'est faisable (avec quelques arguments). Sur Frama-C, je n'ai pas réussi (mais pas tenté tant que ça), donc si vous y arrivez, je serai impressionné.

Réponse:

METTEZ VOTRE RÉPONSE ICI.