Validation d'algorithmes
L3 MIAGE, UGA
2020/2021

Raphaël Charrondière
raphael.charrondiere@inria.fr
Semaine 3 - échéance S4

Correction TD 2: Preuves suite

Où l'on continue à manipuler les variants, les invariants, les ordres bien fondés.

Exercice 1 Tri bulle

```
   Entrées : l liste d'entiers
   Output : la liste l triée par ordre croissant.
 1 stop ← faux;
 2 n ← length(l);
 3 tant que ¬stop faire
 4 |   stop ← vrai;
 5 |   pour i de 1 à n − 1 faire
 6 |   |   si l[i − 1] > l[i] alors
 7 |   |   |   intervertir(l[i − 1], l[i]);
 8 |   |   |   stop ← faux;
 9 |   |   fin
10 |   fin
11 fin
12 retourner r
```

Algorithme 1 : Tri bulle

L'un des algorithmes de tri facile à mettre en place est le tri bulle (algorithme 1). En parcourant la liste, on intervertit les nombres consécutifs qui ne sont pas dans le bon ordre, puis on recommence jusqu'à ce que la liste ne soit plus modifiée.

La difficulté ici est dû au fait des boucles imbriquées. Il va d'abord falloir étudier la boucle interne (ligne 5 à 10) en premier et en déduire une propriété. On admet qu'elle termine (Il faudrait formaliser que $n - i$ diminue strictement et reste positif à chaque entrée de boucle ligne 5. Pour une boucle for simple comme celle-là, ça n'a pas grand intérêt.) Il faut trouver une propriété exploitable.

1 Quel ordre bien fondé pourrait être utilisé sur les listes pour montrer qu'une liste est mieux triée qu'une autre ?

`Correction :` L'ordre lexicographique suffit. On a vu en cours que ce n'est pas un ordre bien fondé, mais on peut montrer que si la longueur des mots est bornée (fixe dans ce cas) alors cet ordre est bien fondé. On notera cet ordre par l'opérateur classique $<$.

2 Formuler la propriété exploitable du bloc interne (lignes 5 à 10)

`Correction :` Il faut traduire qu'on "progresse un peu dans le tri". On propose la propriété $Q(\mathcal{M}_0, \mathcal{M}_f) \equiv$ Si l_0 n'est pas trié $l_f < l_0$ **et** les éléments de $l_0$sont le même que ceux de l_f.

3 Prouver que l'algorithme 1 termine et renvoie le bon résultat. Cette étape doit être intuitive si la question précédente à bien été répondue, sinon, essayez de modifier votre propriété à la question précédente.

`Correction :` Tout simplement d'après la question précédente le variant que la boucle principal est la liste $\phi(\mathcal{M}) = \mathcal{M}(l)$ pour la relation d'ordre $<$, l'invariant est $P(\mathcal{M}, \mathcal{M}_0) \equiv$«Les éléments de l et l_0 sont les mêmes». La démonstration est triviale.

Raphaël Charrondière
raphael.charrondiere@inria.fr


```
   Entrées : i ∈ ℕ
   Output : i
1  si i == 0 alors
2  |  retourner 0;
3  sinon
4  |  retourner 1+StupidAdd(i-1);
5  fin
```

Algorithme 2 : StupidAdd

```
   Entrées : t un arbre binaire de recherche, e
             un élément
   Output : e appartient-il à t ?
1  r ←root(t);
2  tant que r est défini faire
3  |  si r<e alors
4  |  |  r← filsDroite(t);
5  |  sinon si r>e alors
6  |  |  r← filsGauche(t);
7  |  sinon
8  |  |  retourner Vrai;
9  fin
10 retourner Faux;
```

Algorithme 3 : Recherche dans un ABR

Exercice 2 Et en mode récursion ?

Les exemples avec invariant ont pour l'instant fait intervenir une boucle while. Malheureusement dans certains cas, il est plus facile de coder une fonction récurrente, i.e. qui s'appelle elle-même, on va donc devoir s'adapter. Supposons qu'une fonction f ne contienne que des appels à elle même et des instructions qui terminent, et des conditions.

1 Considérer la fonction StupidAdd (Algorithme 2). Pourquoi termine-t-elle ?

`Correction :` La fonction StupidAdd s'appelle elle-même une seule fois si i n'est pas nul, c-à-d. $i > 0$. Or à chaque appel récursif, i est décrémenté de 1, donc i finira par atteindre 0 après i appels récursifs.

2 Proposer une méthode générique pour montrer que f termine, potentiellement avec une condition sur l'état mémoire en entrée. *Remarque : Inspirez vous fortement du cours. Aide : qu'est-ce que voudrait-dire que f ne termine pas*

`Correction :` On va s'inspirer de ce qu'on a vu en cours sur les variants : en cours, les blocs de codes s'exécutent en séquentiels les uns après les autres, ici en récursifs, ils sont imbriqués. Pour la terminaison, ça n'est pas trop un soucis. On définit un espace E muni d'un ordre bien fondé $<$, puis on défini un variant $\Phi : \mathbb{M} \mapsto E$. On note m_0 l'état mémoire en entrée de fonction. On doit prouver que pour chaque appel récursif, en notant m_1 l'état mémoire en entrée de cet appel, $\Phi(m_1) < \Phi(m_0)$.

Dans l'exemple de la question 1, on a $E = \mathbb{N}$, $<$ la relation classique, $\varphi : m \mapsto m(i)$. On montre rapidement $m_1(i) = m_0(i) - 1 < m_0(i)$.

3 Considérer la fonction StupidAdd (Algorithme 2). Pourquoi renvoie-t-elle ce qu'il faut (Pourquoi StupidAdd(i)=i) ?

`Correction :` Le résultat est incrémenté de 1 à chaque appel récursif, et il est de 0 s'il n'y a pas d'appel récursif, comme il y a i appels récursifs, le résultat est bien de i.

4 En supposant que f termine, proposer une méthode générique pour montrer que f «fait bien ce qu'il faut» c.-à-d. est correcte. *Remarque : Commencez avec le cas où f ne s'appelle qu'une fois maximum et pensez au cas où f ne s'appelle pas.*

`Correction :` Pour les boucles while, on montrait qu'un invariant de boucle était vrai en fin d'itération s'il était vrai en début d'itération, on concluait que s'il était vrai en entrée de boucle, alors le bloc était vrai. Ici, c'est un peu plus particulier car les appels s'imbriquent. On définit une propriété $P(m, r)$ où m est un état mémoire en entrée de fonction, et r un résultat. On doit prouver que si pour chaque appel récursif $P(m_r, r_c)$ est vrai où m_r est la mémoire en entrée d'appel récursif et r_c le résultat de cet appel, alors on a $P(m_0, r_0)$ où m_0 est la mémoire en entrée de fonction et r_0 son résultat.

Sur l'exemple on pose $P(m, r) \equiv r = m(i)$. On a deux cas : si $m(i) = 0$ il n'y a pas d'appel récursif donc $r = 0$ clairement $P(m, r)$, sinon on a un appel récursif, on appelle m_1 l'état mémoire en entrée

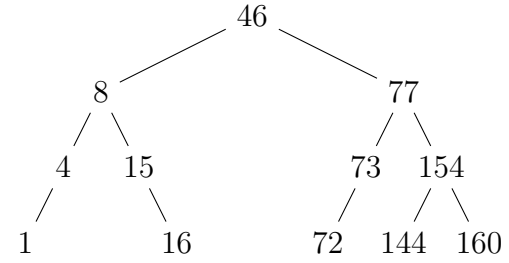
Raphaël Charrondière
raphael.charrondiere@inria.fr

de cette fonction et r_1 son résultat, $P(m_1, r_1) \equiv m_1(i) = r_1$, on peut alors calculer le résultat renvoyé $r = 1 + r_1 = 1 + m_1(i) = 1 + m_0(i) - 1 = m_0(i)$, ce qui prouve $P(m_0, r)$.

Exercice 3 Arbres binaires de recherche

Reprenons les arbres binaires du cours. L'intérêt d'un arbre binaire est de pouvoir recherche rapidement un élément. On propose une fonction dans l'algorithme 3

1 Créez un ABR équilibré (de profondeur minimale) contenant les éléments 1, 4, 8, 15, 16, 46, 72, 73, 77, 144, 154 et 160. Ensuite détaillez l'éxécution de l'algorithme 3 sur l'entrée $e = 5$ et $e = 78$.



Correction :

Attention cette réponse n'est pas unique, du coup, le reste de la question dépend de l'arbre proposé.

Exécution avec $e = 5$:

```
Entrée : M=e=5
Ligne 1 : M=e=5,r=46
Entrée Boucle
Condition ligne 5
Ligne 6 : M=e=5,r=8
Suite boucle
Condition ligne 5
Ligne 6 : M=e=5,r=4
Suite boucle
Condition ligne 3
Ligne 4 : M=e=5,r=Non défini
Sortie boucle
Ligne 10 : Renvoi Faux
```

Exécution avec $e = 78$:

```
Entrée : M=e=78
Ligne 1 : M=e=78,r=46
Entrée Boucle
Condition ligne 3
Ligne 6 : M=e=5,r=77
Suite boucle
Condition ligne 3
Ligne 6 : M=e=5,r=154
Suite boucle
Condition ligne 5
Ligne 4 : M=e=5,r=144
Suite boucle
Condition ligne 5
Ligne 4 : M=e=5,r=Non défini
Sortie boucle
Ligne 10 : Renvoi Faux
```

2 Donnez un invariant et un variant qui prouveraient la terminaison et la correction

Correction : Il faut utiliser la structure d'arbre. Commençons par le variant. À chaque étape on descend dans l'ABR, on propose donc le variant : $\varphi(m) \equiv$«Le nombre d'élément dans le sous arbre de racine $m(r)$». Pour l'invariant, on doit indiquer que l'on cherche dans la bonne partie de l'arbre. $P(m_0, m) \equiv$«Pour tout élément n de l'arbre $m_0(t)$ n'appartenant pas au sous arbre de racine $m(r)$, $n \neq e$». *Remarque : traduire le fait que l'on cherche dans la bonne partie de l'arbre n'est pas "facile", c'est pour cela que l'on formule l'invariant autrement.*

3 Réécrivez la fonction en récursif. *Utilisez une structure similaire à l'algorithme 2 si vous ne savez pas comment faire.*

Raphaël Charrondière
raphael.charrondiere@inria.fr


```
CHERCHER(t, e)
Entrées : t un arbre binaire de recherche, e un élément
Output : e appartient-il à t ?
1  r ←root(t);
2  si r est défini alors
3  |  si r<e alors
4  |  |  retourner CHERCHER (sousArbreDroit(t, e));
5  |  sinon si r>e alors
6  |  |  retourner CHERCHER (sousArbreGauche(t, e));
7  |  sinon
8  |  |  retourner Vrai;
9  fin
10 retourner Faux;
```

Algorithme 4 : Recherche dans un ABR

4 Reprenez les méthodes que vous avez proposé dans l'exercice 2 pour prouver la terminaison et correction de la nouvelle version de l'algorithme.

`Correction :` On va reprendre le variant de la question 2. $\varphi(m) =$«Le nombre d'élément de l'arbre $m(t)$». Clairement le nombre d'élément d'un sous-arbre est inférieur à celui de l'arbre.

Pour l'invariant par contre, contrairement à la question 2, on peut directement proposer $P(m, r) \equiv r =$«$m(t)$ contient $m(e)$». La preuve est en 4 cas :

- $m(r)$ n'est pas défini, c-à-d. $m(t)$ est l'arbre vide. Trivialement l'arbre vide ne contient pas $m(e)$ et on retourne bien $r =$Faux.
- $m(r) = m(e)$. C'est la condition ligne 7. $m(t)$ contient bien $m(e)$ et on renvoi bien $r =$Vrai.
- $m(r) < m(e)$. C'est la condition ligne 3. L'appel récursif vérifie l'invariant **car le sous-arbre droit est bien un ABR**. Donc la fonction renvoie le fait que $m(e)$ appartienne au sous-arbre droit de $m(r)$. Or, par la structure d'ABR on sait que $m(e)$ ne peut être que dans le sous-arbre droit, donc r est vrai si et seulement si $m(e)$ appartient à $m(t)$.
- Le cas $m(r) > m(e)$ est similaire.