Université de Liège

INFO0947

Compléments de Programmation

Un exercice dont vous êtes le Héros.

Fichiers Séquentiels

Simon Liénardy, Benoit Donnet

21 mars 2020

Préambule

Exercices

Dans ce « TP dont vous êtes le héros », nous vous proposons de suivre pas à pas la résolution de deux exercices. L'un sur un problème concernant seulement un unique fichier et l'autre sur les problèmes sur plusieurs fichiers.

Attention ! Dans ce chapitre, nous utilisons un *pseudo-langage* pour manipuler les fichiers. Tout le correctif utilise ce pseudo-langage. Inspectez donc votre inventaire avant de vous lancer dans la quête (3.2.2.1).

Il est dangereux d'y aller seul [1] !

Partagez vos commentaires, questions, solutions alternatives sur le forum eCampus. N'hésitez jamais !

3.2.1 Commençons par un Rappel

Si vous avez déjà lu ce rappel, vous pouvez directement atteindre le point de l'énoncé de l'exercice (Sec. 3.2.3).

3.2.1.1 Fichier Séquentiel ?

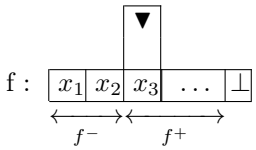


FIGURE 1 – Schéma d'un fichier séquentiel

On représente le fichier séquentiel en dessinant le ruban qui contient les données ainsi que la *tête de lecture/écriture*, symbolisée par le ▼.

En quoi cela diffère-t-il des tableaux ? On n'a accès qu'à un seul élément : celui sous la tête de lecture/écriture [2] (*élément courant*). Il faut de toute façon le lire pour avancer à l'élément suivant. On ne peut pas se déplacer dans le fichier (les ordinateurs en sont capables, évidemment, mais on n'envisage pas cette possibilité aujourd'hui). On peut tester si on est à la fin du fichier grâce à la valeur spéciale ⊥ qui représente le symbole `EOF` (*End Of File*). Au besoin, relisez le Chap. 3, Slides 5 → 8.

1. Référence vidéoludique bien connue des Héros.
2. Si vous avez connu les VHS (cassettes vidéos), vous avez une idée pratique de ce dont on parle. Personnellement, j'ai regardé *Toy Story* (le 2 était pas encore prévu) plusieurs fois grâce à ce média.

3.2.2 Notations sur les Fichiers

Faites en sorte de bien vous **familiariser** avec ces notations ! Si vous ne les comprenez pas, relisez les slides (Chap. 3, Slides 12 → 19)

Notion	Notation
Fichier f	$f = < x_1, x_2, \ldots, x_n >$
Ens. de valeur	$x_i \in V$
Fichier vide	$f = <>$
Restriction de f à l'intervalle $[i \ldots j]$	f_i^j, vide si $j > i$
Concaténation de fichier	$f_1 \,\|\|\, f_2$
Minorant $(\forall i, 1 \leq i \leq n, a < x_i)$	$a < f$
Contient valeur $(\exists i, 1 \leq i \leq n, a = x_i)$	$a \in f$
Ne contient pas valeur $(\forall i, 1 \leq i \leq n, a \neq x_i)$	$a \notin f$
Égalité des élém. à a $(\forall i, 1 \leq i \leq n, a = x_i)$	$a = f$
¬ Égalité des élém. à a $(\exists i, 1 \leq i \leq n, a \neq x_i)$	$a \neq f$
Fin de fichier	$\perp$
Fichier déjà parcouru	f^-
Fichier encore à parcourir (compris `head`[3])	f^+

3.2.2.1 Inventaire des Modules Disponibles

3.2.2.1.1 Ouvrir un fichier avec `fopen()`

```
//
// @pre: mode ∈ "r", "w" ∧ chemin valide
// @post: mode_0 = "r" ⇒ (¬eof(f) ∧ f^- = < > ∧ f^+ = f)
//      ∨ mode_0 = "w" ⇒ (eof(f) ∧ f = < >)
//
FILE fopen(char *chemin, char *mode);
```

Traduction en français Si le mode est `"r"` pour **R**ecture[4], le fichier spécifié par le chemin est ouvert en lecture et la tête de lecture est positionnée sur le premier élément. Si le mode est `"w"` pour **W**écriture[5], le fichier est ouvert en écriture et il est vide. S'il n'existe pas, il est créé avec le nom `chemin`.

Remarque Importante Ici, `FILE` est une variable de type fichier. En temps normal, ce serait un type opaque (c'est d'ailleurs le cas dans `stdio.h` et il faudrait passer un pointeur, `FILE *`). **On ne s'encombre pas de ces considérations dans ce TP**. On manipule `FILE` *comme si* c'était un type primitif. Consultez les slides (Chapitre 3, Slides 24, 29, 31, 36, 39) pour connaitre la traduction du pseudo-langage étudié ici et le C.

3.2.2.1.2 Tester la fin de fichier avec `eof()`

```
// @pre: /
// @post: eof(f) ⇔ f^+ = < >
bool eof(FILE f);
```

Traduction en français Le prédicat est vrai si et seulement si on a atteint la fin du fichier.

3. `head` fait référence à l'élément courant (i.e., sous la tête de lecture/écriture).
4. Prononcé par un non francophone
5. Ceci constitue une blague

3.2.2.1.3 Lire un élément dans un fichier

```
// @pre: f⁻ = f_1^{i-1} ∧ f⁺ = fin ∧ ¬ eof(f)
// @post: val = x_i ∧ f⁻ = f_1^i ∧ f⁺ = f_{i+1}^n ∧
void read(FILE f, type val);
```

Traduction en français[6] L'élément qui est sous la tête de lecture (i.e., l'élément courant) est lu et placé dans val. La tête de lecture progresse ensuite vers l'élément suivant.

3.2.2.1.4 Écrire un élément dans un fichier

```
// @pre: f = <x_1, x_2 ..., x_{i-1}> ∧ val = x_i ∧ eof(f)
// @post: f = <x_1, x_2 ..., x_{i-1} x_i> ∧ val = val_0 ∧ eof(f)
void write(FILE f, type val)
```

Traduction en français La valeur `val` est écrite en fin de fichier. La tête d'écriture est toujours positionnée en fin de fichier suite à l'appel

3.2.2.1.5 Fermer un fichier

```
void fclose(FILE f)
```

Traduction en français Ferme le fichier `f`.

Questions

- Quels fichiers doit-on fermer ?
  R : Tous ceux qui ont été ouverts avec `fopen()`.
- Même ceux ouverts en lecture ?
  R : Oui, c'est une habitude à prendre !
- Même si ça ne risque rien ?
  R : Oui, c'est une bonne habitude à prendre !
- Quel est le risque à ne pas fermer un fichier ?
  R : Les fonctions d'interactions avec le système de fichier, style `fprintf` n'écrivent pas tout de suite dans le fichier lorsqu'elles sont invoquées mais écrivent d'abord dans une mémoire tampon (*buffer* en anglais). La fermeture du fichier force l'écriture du tampon dans le fichier. Un oubli de fermeture de fichier pourrait ne pas écrire le tampon et il pourrait s'ensuivre une perte de données.

6. Par rapport aux slides, j'ai retiré la référence à `eof` dans la Postcondition parce que c'est vrai dans tous les cas

3.2.3 Algorithme sur plusieurs fichiers

3.2.3.1 Énoncé

Écrivez une fonction qui retourne la concaténation de deux fichiers.

3.2.4 Méthode de résolution

Nous allons suivre l'approche constructive vue au cours. Pour ce faire nous allons :

1. Définir de nouvelle·s notation·s (Sec. 3.2.5) ;
2. Spécifier le problème complètement (Sec. 3.2.6) ;
3. Proposer une découpe en Sous-Problèmes (Sec. 3.2.7) ;
4. Chercher un Invariant pour le Sous-Problème (Sec. 3.2.8) ;
5. Construire les Sous-Problèmes (Sec. 3.2.9) ;
6. Rédiger le programme final (Sec. 3.2.10).

3.2.5 Notation

Avant de spécifier le problème, il faut définir de nouvelles notations.
— Si vous ne voyez pas de quoi on parle, rendez-vous à la Sec. 3.2.5.1
— Si vous ne savez plus comment on défini une notation, voyez la Sec. 3.2.5.2.
— Si vous voyez de quelles notations nous avons besoin, rendez-vous à la Sec. 3.2.5.3

3.2.5.1 Quelle·s notation·s ?

En relisant l'énoncé, on a :

> Écrivez une fonction qui retourne la concaténation de deux fichiers.

Avons-nous besoin d'une nouvelle notation ? Non, il suffit relire le tableau 3.2.2 pour s'en convaincre.

Suite du travail

Rendez-vous à la Sec. 3.2.5.3 pour découvrir les notations que nous allons utiliser dans la suite de l'exercice.

3.2.5.2 Comment définir une notation ?

Voici la forme que doit prendre une notation :

NomPredicat(Liste de parametres) ≡ Détail du predicat

NomPredicat est le nom à donner du prédicat. Il faut souvent trouver un nom en rapport avec ce que l'on fait. Mathématiquement parlant, cela ne changera rien donc on pourrait appeler le prédicat « Simon » mais il vaut mieux trouver un nom en rapport avec la définition.

Liste de paramètres C'est une liste de symboles qui pourront être utilisés dans la définition du prédicat. Comment cela fonctionne-t-il ? Le prédicat peut être utilisé en écrivant son nom ainsi qu'une valeur particulière de paramètre. On remplace cette valeur dans la définition du prédicat et on regarde si c'est vrai ou faux (Voir TP1).

Détail du prédicat C'est une combinaison de symboles logiques qui fait intervenir, le plus souvent les paramètres.

Exemple :

$$\text{Pair}(X) \equiv X \ \% \ 2 = 0$$

Le nom du prédicat est « Pair » et prend un argument. $Pair(4)$ remplace $4 \ \% \ 2 = 0$ qui est vrai et $Pair(5)$ signifie $5 \ \% \ 2 = 0$ qui est faux. Le choix du nom du prédicat est en référence à sa signification : il est vrai si X est pair.

▷ **Exercice** Si vous avez compris, vous pouvez définir facilement le prédicat Impair(X).

3.2.5.2.1 Voir aussi

Comment évaluer un prédicat	TP 1
Signification des opérateurs logiques	Chapitre 1, Slides 6 → 9
Signification des quantificateurs logiques	Chapitre 1, Slides 11 → 24
Signification des quantificateurs numériques	Chapitre 1, Slides 25 → 33

Suite de l'exercice

Créer des notations pour cet exercice et rendez-vous à la Sec. 3.2.5.3.

3.2.5.3 Notations utiles

Concaténation de fichier Cette notation est déjà définie :

$$f_{\text{concat}} = f \,||\, g$$

f_{concat} est la concaténation de f et g

3.2.5.4 Si vous aviez d'autres notations

Vous êtes atteint de studentite aigüe !

Studentite aigüe Vous cherchez midi à quatorze heure et vous vous complaisez dans une solution trop complexe. Reposez-vous et passez votre tour ! Éloignez-vous de votre feuille d'exercice pendant 15 minutes [7]. Reprenez ensuite le travail.

Suite de l'exercice

Passons maintenant à la spécification (Sec. 3.2.6).

7. Sérieux, faites-le.

3.2.6 Spécification

Nous avons donc bien une notation de concaténation ||

Suite de l'exercice

Si ce n'est pas le cas pour vous, demi-tour ! Repartez du point 3.2.5.1
Si vous ne savez plus comment spécifier, lisez la Sec. 3.2.6.1
Lorsque vous avez ces deux notations, en avant, spécifiez ! 3.2.6.2

3.2.6.1 Rappel sur les spécifications

La Précondition C'est une condition sur les données. Pour l'établir, il faut se demander quelles doivent être les données du problème. Ici, dans notre exemple, cela parle de tableau. Quelles conditions devons-nous imposer sur ce tableau ? N'hésitez pas à vous référer au cours INFO0946, Chapitre 6, Slides 54 → 64).

La Postcondition C'est une condition sur le résultat du problème, si tant est que la Précondition est remplie (si tel n'est pas le cas, on pratique la *programmation défensive* (voir INFO0946, Chapitre 6, Slides 65 → 70) et on arrête l'exécution du programme.

La signature est souvent nécessaire lors de l'établissement de la Postcondition des fonctions (voir INFO0946, Chapitre 6, Slide 11). En effet, le résultat retourné par l'invocation de la fonction est représenté par le nom de la fonction dans la Postcondition. Le nom de la fonction doit donc être défini avant d'en parler.

À vous !

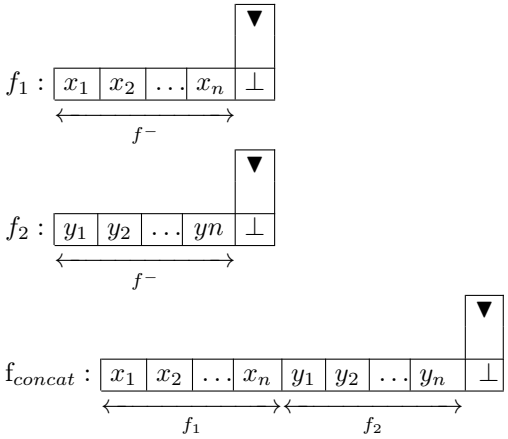
Spécifiez le problème, rendez-vous en Sec. 3.2.6.2.

3.2.6.2 Correction de la spécification

Précondition On demande que les noms des fichiers soient initialisés. Il y a trois fichiers : le premier, le deuxième et leur concaténation.

$$Pre : \texttt{nom\_fichier1}\ init \wedge \texttt{nom\_fichier2}\ init \wedge \texttt{nom\_concat}inits$$

Postcondtion On doit atteindre la situation représentée par le schéma suivant [8]



Dans cette situation, f_{concat} contient la concaténation des deux fichiers f_1 et f_2. Ces fichiers ne sont pas modifiés

$$Post : f_{concat} = f_1 \,||\, f_2 \wedge f_1 = f_{1_0} \wedge f_2 = F_{2_0}$$

Signature

```
1 void fconcat(char *nom_fichier1, char *nom_fichier2, char *nom_concat)
```

8. Commencer par un dessin est toujours une bonne idée. Cela facilite l'écriture formelle (il suffit d'une simple traduction).

3.2.7 Découpe en Sous-Problèmes

Pourquoi devrions-nous envisager une découpe en Sous-Problèmes dans cet exercice ? C'est la question qu'il faut se poser. Y'a-t-il quelque chose qu'il faudra exécuter plusieurs fois et qu'il faudrait mettre en évidence ?

Suite de l'exercice

Trouvez ce qu'il convient de calculer à l'aide d'un Sous-Problème (SP) et spécifiez-le ! Rendez-vous à la Sec. 3.2.7.1.

3.2.7.1 Spécification du Sous-Problème

Concaténer deux fichiers dans un fichier résultat consiste :

1. À recopier f_1 dans f_{concat}
2. À recopier f_2 dans f_{concat}, à la suite de f_1

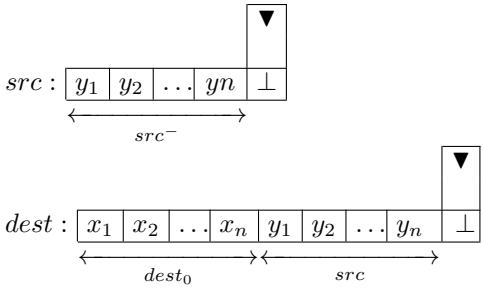
Il nous faudrait donc un SP qui ajoute en fin de fichier le contenu d'un autre fichier. Appelons ce SP `fappend`. Spécifions cette fonction maintenant (voir Sec. 3.2.7.2).

3.2.7.2 Spécification de `fappend`

Précondition Il faut avoir ouvert *src* en lecture et *dest* en écriture :

$$dest^{+} = <> \wedge \text{mode}(src) = "r" \wedge \text{mode}(dest) = "w"$$

Postcondition On doit atteindre la situation représentée par le schéma suivant [9]



Donc $dest_0$ ne doit pas être écrasé.

$$Post : dest = dest_0 \wedge dest = dest_0 \,||\, src$$

Signature

```
1 void fappend(FILE src, FILE dest);
```

9. Commencer par un dessin est toujours une bonne idée. Cela facilite l'écriture formelle (il suffit d'une simple traduction).

3.2.8 Invariant du Sous-Problème

3.2.8.1 Rappels sur l'Invariant

Si vous voyez directement ce qu'il faut faire, continuez en lisant les conseils de la Sec. 3.2.8.2.

Petit rappel de la structure du code (cfr. Chapitre 2, Slide 28) :

```
// {Pre}
INIT
// {Inv}
while(B){
   // {Inv ∧ B}
   ITER
   // {Inv}
}
// {Inv ∧ ¬B}
END
// {Post}
```

La première chose à faire, pour chaque sous-problème, est de déterminer un Invariant. La meilleure façon de procéder est de d'abord produire un Invariant Graphique et de le traduire ensuite formellement. Le code sera ensuite construit sur base de l'Invariant.

Règles pour un Invariant Satisfaisant

Un bon Invariant Graphique respecte ces règles :

1. La structure manipulée est correctement représentée et nommée ;
2. Les bornes du problèmes sont représentées ;
3. Il y a une (ou plusieurs) ligne(s) de démarcation ;
4. Chaque ligne de démarcation est annotée par une variable ;
5. Ce qu'on a déjà fait est indiqué par le texte et utilise des variables pertinentes ;
6. Ce qu'on doit encore faire est mentionné.

Ce sera un encore meilleur Invariant dès que le code construit sur sa base sera en lien avec lui.

3.2.8.2 Quelques conseils préalables

Un Invariant décrit particulièrement ce que l'on a déjà fait au fil des itérations précédentes. Dans le cadre d'un fichier séquentiel, on possède une notation des éléments déjà vus : f^-. L'Invariant parlera donc **quasi toujours** de f^-. Ce qu'il reste à calculer correspond au traitement à appliquer à f^+.

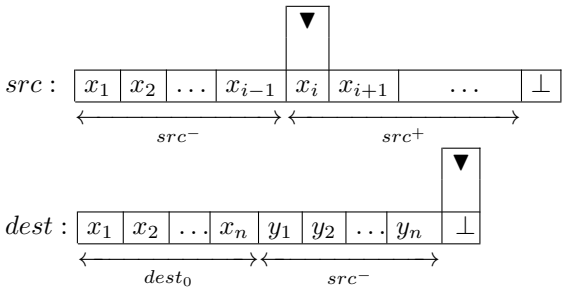
Une bonne technique consiste à retravailler la Postcondition (cfr. Chapitre 2, Slide 29) et d'y faire apparaitre f^-. Il faut ensuite se poser la question : « à l'itération i, de quelle·s information·s ai-je besoin pour traiter le i^e élément que je m'apprête à lire dans le fichier ? » (dans ce cas, on a évidemment $long(f^-) = i - 1$ [10]).

Suite de l'exercice

Pour l'Invariant Graphique 3.2.8.3
Pour l'Invariant Formelle 3.2.8.4

10. La définition de la notation $long(\ldots)$ est donnée dans l'Exercice dont vous êtes le héros portant sur un seul fichier

3.2.8.3 Invariant Graphique



On dessine deux fichiers. On indique que ce qu'on a déjà parcouru (lu) dans le fichier source *src* a été écrit dans le fichier destination *dest*

La traduction de cet Invariant Graphique en Invariant Formel est disponible à la Sec. 3.2.8.4.

3.2.8.4 Invariant Formel

On traduit l'Invariant Graphique (Sec. 3.2.8.3) en Invariant Formel :

$$Inv : dest = dest_0 \,||\, src^- \wedge eof(dest)$$

On est constamment en train d'écrire dans *dest*, donc on est toujours à la fin du fichier. On décrit la concaténation.

3.2.9 Construction du Sous-Problème

Voici ce qu'il faut spécifier dans l'Approche Constructive (Chapitre 2, Slides 25 $\rightarrow$ 37)

INIT De la Précondition, il faut arriver dans une situation où l'Invariant est vrai

B Il faut trouver d'abord la condition d'arrêt $\neg B$ et en déduire le gardien

ITER Il faut faire progresser le programme puis restaurer l'Invariant.

END Vérifier si la Postcondition est atteinte si $\{Inv \wedge \neg B\}$, sinon, amener la Postcondition.

Fonction t Il faut donner une fonction de terminaison pour montrer que la boucle se termine (cfr. INFO0946, Chapitre 3, Slides 97 $\rightarrow$ 107).

Tout ceci doit être justifié sur base de l'Invariant, à l'aide d'assertions intermédiaire. Notez que l'ordre dans lequel vous faites ces étapes n'a pas d'importance !

Suite de l'exercice

La suite de l'exercice est disponible à la Sec. 3.2.9.1.

3.2.9.1 Approche constructive pour `fappend`

3.2.9.1.1 INIT

Dans `fappend`, les fichiers sont déjà ouverts. Il ne faut donc pas les réouvrir.
Il faut par contre déclarer la variable `val` qui servira à la lecture de *src* et à l'écriture de *dest*

```
type val;
```

3.2.9.1.2 B

On commence par définir la condition d'arrêt en comparant la postcondition et l'invariant (voir le schéma de la section 3.2.6. Il faut s'arrêter après avoir lu l'entièreté de *src*

$$\neg B \equiv eof(src)$$

$$B \equiv \neg eof(src)$$

3.2.9.1.3 ITER

```
while(!eof(src)) {
      // {Inv ∧ B : dest = dest0 || src⁻ ∧ src⁺ ≠ < > ∧eof(dest) ⇒ Pre_read ∧ eof(dest)}
      read(src, val);
      // {Post_read : src⁻ = src⁻⁻ || < val > ∧ dest = dest0 || src⁻⁻ ∧ eof(dest) ⇒ Pre_write}
      write(dest, val);
      // {src⁻ = src⁻⁻ || < val > ∧ dest = dest0 || src⁻⁻ || < val > ⇒ dest = dest0 || src⁻ ∧ eof(dest) ⇒ Inv}
   }
```

On lit l'élément sous la tête de lecture de *src* (i.e., l'élément courant) et on l'écrit dans *dest*. Ce faisant, l'Invariant est déjà rétabli. On passe donc à l'itération suivante. Avant d'appeler `read`, on se met dans une situation où sa Précondition est vrai (c'est-à-dire $\neg eof(src)$, garantie par le gardien). Après l'appel, par l'approche constructive, la Postcondition est vérifiée. On est toujours dans une situation où `write` peut être appelée car $eof(dest)$ est garanti par l'Invariant. Après son appel, par l'approche constructive, sa Postcondition est respectée et `val` est rajoutée en dernière position du fichier de destination.

3.2.9.1.4 END

Vu $Inv \wedge \neg B$, il n'y a rien à faire :

$$Inv \wedge \neg B \Rightarrow src^+ = \ < > \ \wedge src^- = src \Rightarrow dest = dest_0 \,||\, src$$

Il faut juste quitter le module `fappend`.

3.2.9.1.5 Fonction de terminaison

Le fichier *src* n'est pas de taille infinie. Trouver ce qui décroit à chaque itération est simple : $long(src^+)$ [11]

$$t = long(src^+)$$

11. La définition de la notation $long(\ldots)$ est donnée dans l'Exercice dont vous êtes le héros portant sur un seul fichier

3.2.9.1.6 Code final du Sous-Problème `fappend`

```
1  void fappend(FILE src, FILE dest){
   int val;
3     while(!eof(src)) {
         // {Inv ∧ B : dest = dest_0 || src^- ∧ src^+ ≠ < > ∧eof(dest) ⇒ Pre_read ∧ eof(dest)}
5        read(src, val);
         // {Post_read : src^- = src^-- || < val > ∧ dest = dest_0 || src^-- ∧ eof(dest) ⇒ Pre_write}
7        write(dest, val);
         // {src^- = src^-- || < val > ∧ dest = dest_0 || src^-- || < val > ⇒ dest = dest_0 || src^- ∧ eof(dest) ⇒ Inv}
9     }
      // {¬B ⇒ src^- = src}
11    // {Inv ∧ ¬B ⇒ dest = dest_0 || src}
      return;
13    // {Post}
   } // fin fappend
```

Un dernier pour la route

Grâce à `fappend`, on peut facilement construire le programme principal (`fconcat`). (Sec. 3.2.10).

3.2.10 Programme final

```
void fconcat(char *nom_fichier1, char *nom_fichier2, char *nom_concat) {
   FILE f1 = fopen(nom_fichier1, "r");
   FILE f2 = fopen(nom_fichier2, "r");
   FILE f_concat = fopen(nom_concat1, "w");
   // {Post_{fopen,f1} ∧ Post_{fopen,f2} ∧ Post_{fopen,fconcat}}
   // { ⇒ f_concat = < > ∧ eof(f_concat) ∧ f1 = f1+ ∧ f2 = f2+ }

   // {Pre_fappend}
   fappend(f_1, f_concat)
   // {Post_fappend : f_concat = f_concat0 || f1 = < > || f1 = f1 ∧ eof(f_concat)}
   //{Pre_fappend}
   fappend(f_2, f_concat)
   // {Post_fappend : f_concat = f_concat0 || f2 = f1 || f2 ⇒ Post}

   fclose(f1);
   fclose(f2);
   fclose(f_concat);
}
```

Avant d'appeler `fappend`, il faut vérifier que sa Précondition est vérifiée. Ensuite, après l'appel, sa Postcondition est respectée par **l'approche constructive**.

Par les appels multiples à `fappend`, on obtient le fichier

$$< > || f_1 || f_2 = f_1 || f_2$$

Qui est bien le fichier final attendu.