U.F.R SCIENCES ET TECHNIQUES
Département d'Informatique
B.P. 1155
64013 PAU CEDEX
Téléphone secrétariat : 05.59.40.79.64
Télécopie : 05.59.40.76.54

IV-Correction des algorithmes

Un des problèmes majeurs auxquels sont confrontés tous les **concepteurs d'algorithmes** c'est la **correction** de leurs algorithmes.

Autrement dit est-ce qu'un algorithme donné:

-**réalise bien** le traitement pour lequel il a été conçu?

- et **rien que** ce traitement?

Disons d’ores et déjà que ce problème **ne relève pas du domaine de calcul: il est** **indécidable**

C’est d’ailleurs l’**inexistence** d’**algorithme** :

- garantissant la **correction** des algorithmes
- qui explique la **présence d’erreurs** dans les logiciels d’application.

Faute de **confiance totale**, il existe néanmoins:

-des méthodes qui contribuent de manière significative

- à augmenter la **confiance** que l'on peut avoir en la correction d'un algorithme.

C'est l'objet de l'étude de la **correction** des algorithmes.

I- Test et Vérification

Les méthodes utilisées pour augmenter la **confiance** dans la correction d'un algorithme sont regroupées en deux catégories:

- le **test**
- et la **vérification**.

Qu’est-ce qu’un test ?

Le test consiste à exécuter un algorithme :

-avec des **données de test** (**DT**)

-pour lesquelles on connaît les résultats attendus: **Oracle**

Seulement, il n'est souvent pas possible de traiter tous les jeux de DT possibles: **test exhaustif**

C'est pour cela qu'on se limite aux de jeux de DT dites **représentatives**

Ce sont celles qui sont générées selon certains **critères**.

L'ingénieur de test est amené alors à appliquer des **techniques de test.**

Il est clair qu'un algorithme :

-dont la **correction** a été montrée de cette manière

-n'est correct **que** pour les **cas particuliers** qui ont été testés.

Au point de limiter l'objectif d'un test à la **capture des défauts**

…et **non** à la preuve qu'un programme en est **exempte**.

Qu'est-ce qu'une vérification ?

Par contre, **prouver** qu'un algorithme est **correct**, signifie :

- qu'on le **garantit**
- pour **toutes** les **données valides**.

Prouver la correction revient donc à considérer :

-le comportement **général** d'un algorithme,

-plutôt que son comportement par rapport à un **ensemble limité** d'entrées.

Notons, surtout, qu'on ne peut pas parler de la **correction** d'un algorithme dans l'**absolu**.

Désormais, on parle d’algorithme :

- **correct**
- **vis-à-vis** de sa spécification.

Le problème de correction ne peut se concevoir que **par rapport** à un ensemble de **propriétés**

En génie logiciel, cet ensemble de propriétés est énoncé dans une **spécification**.

II- Correction partielle et correction totale

Le problème de la correction d’un algorithme présente de nombreuses facettes qui sont toutes importantes en conception d’algorithme.

Examinons deux d’entre elles, qui sont fondamentales et qui sont au cœur du problème.

Premièrement : la correction partielle

Il serait rassurant de savoir qu'un algorithme:

- **lorsqu'il se termine**,
- produit un résultat qui est bien celui **attendu**.

Cette propriété est appelée **correction partielle**.

Notons que si un algorithme est partiellement correct, il se peut qu’il ne se termine pas **toujours**.

La **seule garantie** que l’on a, est que :

- si on obtient un résultat: **terminaison**
- alors ce résultat est **exact**.

Deuxièmement : la correction totale

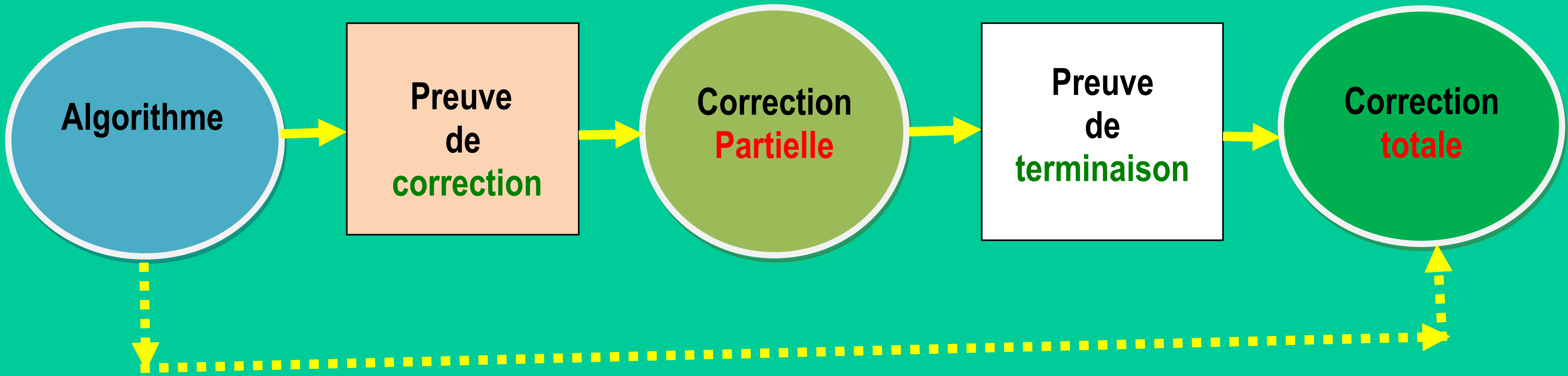
L'autre facette du problème est centrée sur la **terminaison**.

Un algorithme:

- qui est **partiellement correct**
- qui se termine **toujours**

est alors dit **totalement correct**.

En résumé



Algorithme
Preuve de correction
Correction Partielle
Preuve de terminaison
Correction totale

Comment apporter la preuve de correction ?

Preuve par induction

Preuve par assertion

III- La preuve par récurrence

Considérons l'algorithme suivant où **$** est utilisé pour indiquer un endroit particulier du code.

```
exponentiation(x)
debut --Cet algorithme produit 2^x, en supposant que x ∈N
      somme ←1
      pour i = 1 à x faire
        somme ←somme + somme
        $
      finpour
      sortir somme
fin
```

Sa **spécification** dit que :

«il doit calculer **2^x**, x étant un entier non négatif ».

Pour **prouver sa correction**, il est essentiel de le lire jusqu'au bout.

Dans ce cas simple, il est suffisant de comprendre que lorsque le point marqué **$** est atteint pour la **n**$^{\textbf{ième}}$ fois, **somme** est égale à **2**$^{\mathbf{n}}$.

Il est alors évident que dès que la fin du **pour** est atteinte, **somme** est égale à **2**$^{\mathbf{x}}$.

Ce que nous venons de montrer est une **preuve** :

- **informelle** adéquate
- de la **correction** de cet algorithme.

Mais nous n’avons prouvé qu’une **correction partielle**.

Il est facile de voir qu’il se **termine** aussi, en partant de la sémantique de la boucle **pour**.

Dans la pratique, on peut considérer qu'une telle **preuve** est suffisante.

Pour faciliter la lecture de l'algorithme, on peut compléter sa spécification comme suit :

exponentiation(x)

<u>debut</u>

--*Cet algorithme produit 2^x, en supposant que $x \in N$*

somme $\leftarrow$1

<u>pour</u> i = 1 **<u>à</u>** x **<u>faire</u>**

somme$\leftarrow$somme + somme

$ --*Quand ce point est atteint pour la $n^{ième}$ fois, somme est égale à 2^n*

<u>finpour</u>

sortir somme

<u>fin</u>

2-L'induction

La preuve précédente peut être rendue plus **formelle** en suivant les **5** étapes suivantes:

Etape 1:

La **première fois** que **$** est atteint, **somme** = 2 c'est-à-dire :

$$\textbf{somme} = 2^1.$$

Etape 2:

Supposons que **$** est atteint pour la **n**$^{\text{ième}}$ fois avec :

$$\textbf{somme} = 2^n$$

alors à la **(n+1)**$^{\text{ième}}$ fois :

$$\begin{aligned}\textbf{somme} &= 2^n + 2^n \\ &= 2 \times 2^n \\ &= 2^{n+1}.\end{aligned}$$

Etape 3:

Par **conséquent** :

$$\forall n \bullet \ \mathbf{somme} = 2^{\mathbf{n}}$$

à la $n^{\text{ième}}$ fois que **$** est atteint.

Etape 4:

Donc, si on arrive à la fin du **pour**, il y a deux possibilités:

1- soit la boucle **pour** n'a jamais été exécutée et dans ce cas :

$x = 0$

somme $= 1 = 2^{0}$

2-soit la boucle **pour** a été exécutée et dans ce cas **$** a été atteint **x** fois; d'où:

somme $= 2^{x}$.

<u>Etape 5</u>:

Dans le deuxième cas, quand la fin du <u>**pour**</u> est atteinte, 2^x est le résultat produit.

Cette **preuve** illustre la **technique de récurrence** souvent utilisée dans les preuves d'algorithmes.

Elle est appelée aussi **technique d'induction**.

Elle est très utile quand on veut prouver qu'un énoncé est vrai dans n'importe quel cas envisagé.

Son principe est simple :

-prouver que :

si un énoncé est vrai pour un **cas particulier**
alors il l'est aussi pour le **cas immédiatement supérieur**.

-il en résulte ainsi qu'il est vrai dans **tous** les cas.

Ce principe est illustré par les étapes (1), (2) et (3) de la preuve précédente.

Le but est de prouver l'énoncé (3) : c'est-à-dire que pour tout **n**, quand **$** est atteint pour la $n^{\text{ième}}$ fois :

somme = $\mathbf{2^n}$.

L'étape (1) montre que l'énoncé est vrai pour **n=1**.

L’étape (2) montre que:

- si l’énoncé est vrai pour toute valeur **particulière** de **n**
(**cas particulier**)

-**alors** il l’est aussi pour **n+1**.
(**cas immédiatement supérieur**).

En combinant (1) et (2), on prouve que l'énoncé est vrai pour n=2, pour n=3, pour n=4 et ainsi de suite.

On a donc prouvé en (3) que l'énoncé est vrai pour toute valeur de **$n \geq 1$**.

En résumé, une **preuve par induction** est constituée de trois parties:

- une **hypothèse de récurrence**,
- une **base**
- et une **étape de récurrence**.

L'**hypothèse de récurrence** décrit l'énoncé à **prouver** : c'est le but de l'étape (3) dans l'exemple.

La **base prouve** le cas de départ : c'est l'étape (1) dans l'exemple.

L'**étape de récurrence** permet d'aller :

- de la **base**
- vers des **cas progressivement supérieurs**.

C'est le but de l'étape (2) dans l'exemple.

On peut exposer de façon plus claire la méthode pour décrire la preuve précédente:

1-<u>Hypothèse de récurrence</u> :

Quand **$** est atteint pour la $n^{ième}$ fois, alors

somme = 2^n.

2-<u>Base</u> :

Quand $n = 1$, on a :

somme = 1+1 = 2^1.

3-<u>Etape de récurrence</u> :

Supposons que l'hypothèse de récurrence soit vraie pour une valeur **particulière** de **n**.

somme $=2^{n}$

La $(n+1)^{\text{ième}}$ fois que **$** est atteint :

somme = somme + somme

Par l'**hypothèse de récurrence**, la valeur précédente de **somme** est 2^n.

La nouvelle valeur de somme est donc :

$$\textbf{somme} = 2^n+2^n=2^{n+1}.$$

Ainsi, l'hypothèse de récurrence vaut pour **n+1**.

En partant :

- de la **base**
- et de l'**étape de récurrence**,

il résulte que l'**hypothèse de récurrence** vaut pour toute valeur de **n** ≥ 1.

Exemple de la factorielle

Considérons l'algorithme simple de la factorielle :

fact (x)

debut

--Pour un entier $x \geq 0$, le résultat de cet algorithme est x !

(1) **si** $x \leq 1$

(2) **alors** fact := 1

(3) **sinon** fact := x * fact (x - 1)

finsi

fin

Hypothèse de récurrence:

Avec $x = n \geq 1$, on a :

$$\mathbf{fact}(n) = n\,!$$

Base:

Pour n = **1**, le test de la ligne (1) entraîne l'exécution de la ligne (2).

Le résultat retourné par fact est 1:

fact (**1**) = **1** !

Etape de récurrence:

Supposons maintenant que **fact** retourne **n !** quand il est appelé avec un argument $n \geq 1$.

Considérons alors ce qui arrive lorsque **fact** est appelé avec la valeur **n+1** pour la variable x.

Si $n \geq 1$, alors n+1 vaut au moins 2, donc ce cas la récurrence s'applique (ligne 3).

La valeur retournée est alors :

$$x \, . \, \mathrm{fact}\,(x - 1)$$

Comme :

$$x=n+1$$

Le résultat retourné est :

$$(n+1)\times fact(n)$$

Comme par l'hypothèse de récurrence, on a:

$$fact\ (n) = n\ !$$

Et comme :

$$(n+1) \times n\ ! = (n+1)\ !$$

nous avons prouvé l'étape de récurrence c'est-à-dire que:

$$\textbf{fact}\ (n+1) = (n + 1)\ !$$

<u>Conclusion</u>:

En partant de la **base** et de l'**étape de récurrence**, il résulte que l'hypothèse de récurrence vaut pour toute valeur de **n**≥1.

Exemple de l’algorithme du pgcd

Considérons l’algorithme du pgcd d’**Euclide** :

```
pgcd(x, y)
debut
tant que y ≠ 0 faire
    calculer x%y
    x ← y
    y ← x%y
    $
fintantque
sortir x
fin
```

Son objectif est de produire pour deux entrées entières non négatives:

x et y,

leur **plus grand commun diviseur** noté pgcd(x, y).

Pour faire comprendre son algorithme, **Euclide** a dû expliquer qu'au point marqué **$** :

pgcd(x, y) = pgcd(y, x%y).

Ceci peut être montré par **induction** comme suit :

<u>Hypothèse de récurrence</u>:

Quand **$** est atteint :

g = pgcd(x, y)

= pgcd(y, x%y).

Base:

Par définition du pgcd de x et y, on a:

$$x = gx' \text{ et } y = gy'$$

où x' et y' sont **premiers entre eux** (aucun facteur commun…sauf 1)

On peut aussi écrire :

$$x = my + r \qquad\qquad m \text{ :entier}$$

où $r = x\%y$.

On a alors :

$$gx' = mgy' + r$$

d'où : $$x' = my' + r/g$$

C'est-à-dire :

$$\mathbf{r/g =} x' - my'$$

Ce qui signifie que **g** est aussi **un diviseur** de

$$r = x\%y$$

Comme, par ailleurs, on:

$$r/g + my' = x'$$

Ceci met en évidence le fait que **r**/**g** et **y'** sont **premiers entre eux**.

Car sinon, tout facteur k :

-qui divise **r/g** et **y'**

-diviserait aussi **x'**

Ce qui est contradictoire avec le fait que **x'** et **y'** sont **premiers entre eux** : propriété du pgcd.

Donc, y/**g** et **r**/**g** sont premiers entre eux.

Ce qui, par définition, signifie que **g** est le pgcd de **y** et **r**.

En conséquence, la **première fois** que **$** est atteint, on a :

pgcd(x, y) = pgcd(y, **r**)
= pgcd(y, x%y)
= **g**.

Etape de récurrence:

Si l'on admet que l'hypothèse de récurrence vaut pour **n** : la $n^{\text{ième}}$ fois où **$** a été atteint.

On a alors:

$$pgcd(x,y) = \mathbf{g}$$

A la (**n+ 1**)$^{\text{ième}}$ fois où **$** est atteint, on a :

pgcd(y, x%y) = **g**

= pgcd(x,y)

En conséquence, en partant :

-de la **base**

-et de l'**étape de récurrence**

il résulte que l'**hypothèse de récurrence** vaut pour toute valeur de n≥1.

C'est-à-dire chaque fois que **$** est atteint.

Cette preuve **par récurrence** nous a aidé à comprendre l'effet de l'algorithme à chaque fois que **$** est atteint.

Mais, attention, la sortie de la boucle:

<u>tant que</u> $y \neq 0$ **<u>faire</u>**

n'est pas évidente !

Si c'est le cas, on sait qu'en ce point de sortie, on a :

pgcd(x, y) = **g**

et y = **0**

d'où x = **g**

Et l'algorithme sort bien **g** comme prévu.

Nous venons de montrer la **correction partielle** de l'algorithme d'Euclide.

Cela signifie que **si la sortie est atteinte**, l'algorithme produit bien le résultat attendu.

La **terminaison** et par conséquent la **correction totale** de l'algorithme seront montrées par la suite.

Nous avons souligné l'importance de la compréhension d'un algorithme, quand on veut montrer sa correction.

On peut même affirmer que :

-les algorithmes **difficiles à comprendre**

-rendent nécessairement **difficile la preuve** de leur correction.

IV. La preuve par assertions

La preuve par **induction** est bien adaptée aux **algorithmes courts**.

Pour des algorithmes **conséquents**, on utilise des **méthodes plus larges**.

Ces méthode doivent pouvoir prendre en compte la **preuve** :

- chacune des **parties** de l'algorithme
- ainsi que les **liens** qui existent entre elles.

La **méthode des assertions** compte parmi de telles méthodes.

Spécification d’un algorithme

La **spécification** d’un algorithme doit comporter **deux** parties principales :

1-la **première** décrit les **entrées** sur lesquelles l’algorithme doit s’exécuter.

Par exemple, pour l'algorithme d'Euclide, ce sont des **entiers non négatifs** x et y,

2-la **deuxième** décrit le **résultat produit**.

Par exemple, c'est le calcul du **pgcd des entrées**, pour ce même algorithme d'Euclide.

Ces deux parties sont désignées respectivement par :

- **pré-conditions**
- et **post-conditions**.

La **Pre-condition** exprime les **conditions** que doivent satisfaire les **entrées valides** de l'algorithme

La **Post-condition** exprime les **conditions** qui garantissent que le **résultat** de l'algorithme est **correct**.

Autrement dit:

- les **pré-conditions** décrivent l'état du calcul **avant** l'exécution de l'algorithme

-et les **post-conditions**, l'état **après** l'exécution.

- **Spécification**

$\{x \geqslant 0 \wedge y \geqslant 0 \wedge x = x_0 \wedge y = y_0\}$

```
while (y > 0)
   {
      int Save = y;
      y = x % y;
      x = Save;
   }
```

$\{x = pgcd(x_0, y_0)\}$

Précondition

Code à vérifier

Postcondition

Qu'est-ce qu'une assertion ?

On appelle **assertion** une relation entre les variables qui est vraie, à un moment donné, au cours de l'exécution de l'algorithme.

Les **pré-conditions** et les **post-conditions** constituent des **assertions particulières.**

Autrement dit:

- la **pré-condition** est **vraie** **avant** l'exécution de l'algorithme

-la **post-condition** est **vraie** **après** son exécution.

Outre les pré-conditions et les post-conditions, un algorithme **conséquent** doit comporter d'autres **assertions intermédiaires.**

Ces **assertions intermédiaires** sont placées en des endroits clés de l'algorithme.

Dans la **pratique**, cela signifie :

- qu'il faut placer au moins une assertion
- dans **chaque boucle** de l'algorithme.

Cette **assertion** qui est vraie chaque fois qu'elle est atteinte est appelée **invariant de boucle**.

Attention !

Ne pas confondre

expression booléenne: évaluée dans une exécution du programme

propriété : utilisée dans la démonstration

assertion : utilisée dans le test de programme, évaluée systématiquement et qui lève une exception si elle est fausse

propriétés (parfois appelées **assertions** !) sans se prononcer *a priori* sur leur valeur de vérité

Une **propriété (des variables)** peut être vraie ou fausse à un point donné de l'exécution d'un programme, et selon les données initiales.

Triplet de Hoare

Une portion du programme ou **code** est **correcte** si le **triplet de Hoare**:

{ P} **code** {Q}

est **vrai**.

P : **précondition**
Q : **post-condition**

Exemples

$\{t=13\}\ \ \Delta t := 2 ;\ t' := t + \Delta t\ \ \{t'=15\}$

est une assertion **vraie**

$\{t=11\}\ \ \Delta t := 2 ;\ t' := t + \Delta t\ \ \{t'=15\}$

est une assertion **fausse**

Pré-condition : $\{\ -5 \geq t \geq 30\}$

Code : $\Delta t := 2 ;\ t' := t + \Delta t$

Post-condition: $\{t' = t + \Delta t\}$

$\{x = 4\}$ y:= sqrt(x) $\{y = 2\}$

Pré-condition: $\{x \geq 0\}$
Code: y := sqrt(x)
Post-condition: $\{y^2 = x\}$

Correction d'une séquence d'instructions

Soit la séquence:

$$S_1;\ \ S_2;\ \ \ldots;\ S_n$$

On écrit:

$\{P\}$
S_1
S_2
$\ldots$
S_n
$\{Q\}$

Pour vérifier que le triplet est correct :

1-on insère des assertions $P_1, P_2, \ldots, P_n$ décrivant l'**état des variables** à chaque étape de la séquence

2-on vérifie que les triplets :

$$
\begin{array}{lll}
\{P\} & S_1 & \{P_1\}, \\
\{P_1\} & S_2 & \{P_2\}, \\
\ldots & & , \\
\{P_{n-1}\} & Sn & \{Q\}
\end{array}
$$

sont vrais.

L'instruction S_i peut désigner l'une des instructions de base suivante:

- une **affectation**,
- une **conditionnelle**,
- ou une **boucle.**

Dans ce qui va suivre, on va **distinguer séparément** les différents cas de base.

Correction d'une affectation

On a le triplet de Hoare:

$$\{P\}$$
$$x := e$$
$$\{Q\}$$

Pour prouver que ce triplet est **correct**, il faut montrer qu'on peut :

- déduire **Q**
- à partir de **P**, en remplaçant **x** par **e** :

$$\mathbf{P} \quad \Rightarrow \mathbf{Q}_{(x=e)}$$

Exemple

$$\{x \geq 2\}$$
$$y := x^2 + 1$$
$$\{y \geq 3\}$$

On a :

$$(\mathbf{x \geq 2}) \Rightarrow \quad (x^2 + 1 \quad \geq 5)$$

$$(x^2 + 1 \quad \geq 5) \Rightarrow (y\mathbf{=}\ x^2 + 1 \geq 5) \quad \Rightarrow \quad (y \geq 3)$$

D’où :

$$(x \;\geq 2) \;\Rightarrow\; \mathbf{(y \geq 3)}$$

Correction d'une conditionnelle

On a le triplet de Hoare:

{P}
IF **C**
S_1
ELSE
S_2
{Q}

Pour prouver que le triplet est **correct**, on doit prouver que les deux triplets suivants:

$\{\mathbf{P} \wedge \mathbf{B}\}$	$\{\mathbf{P} \wedge \neg\mathbf{B}\}$
S1	S2
$\{\mathbf{Q}\}$	$\{\mathbf{Q}\}$

sont **corrects**.

Exemple

$\{x < 6\}$

if $x < 0$

$\quad y = 0$

else

$\quad y = x$

$\{0 \leq y < 6\}$

$\{x < 6 \text{ et } x < 0\}$

$y = 0$

$\{0 \leq y < 6\}$

$\{x < 6 \text{ et } \neg (x < 0)\}$

$y = x$

$\{0 \leq y < 6\}$

Correction d'une boucle

On part du triplet de base suivant:

```
{P}
INIT
WHILE  C
        CORPS
FIN
  {Q}
```

pour construire le triplet:

$\{P\}$

INIT

$\{ I \}$

WHILE C

$\{I \wedge C\}$

CORPS

$\{I\}$

$\{I \wedge \neg C\}$

FIN

$\{Q\}$

Pour prouver que le triplet est correct :

1-on met en évidence une assertion particulière I, appelée **invariant de boucle**.

L'**invariant** décrit une propriété **pendant** la boucle.

2-on doit prouver que successivement:

-**avant** la boucle :

{**P**} **INIT** { I } est correct

-**pendant** la boucle

$\{I \wedge C\}$ **CORPS** { I } est correct

-**à la fin** de la boucle :

$\{I \wedge \neg C\}$ **FIN** {**Q**} est correct

Si on a plusieurs boucles **imbriquées**, on les traite :

- **séparément**,
- en démarrant avec la boucle **la plus interne**.

Exemple d’exponentiation rapide

<u>Idée :</u>

Pour calculer $\mathbf{X}^{\mathbf{n}}$ on utilise les relations suivantes :

n pair : $n=2q \Rightarrow X^{2q} = (x^q)^2$

n impair : $n=2q+1 \Rightarrow X^{2q+1} = x.(x^q)^2$

invariant :

$$\mathbf{x^n = z.\, y^m}$$

n pair : $n=2q \quad \Rightarrow \quad \mathbf{z = 1 \quad y^m =} (x^q)^2$

n impair : $n=2q+1 \quad \Rightarrow \quad \mathbf{z =} x \,; \quad \mathbf{y^m =} (x^q)^2$

```
expoRapide(x,n):   -- la fonction retourne x^n
y:=x;  z:=1;  m:= n
#Inv: z*y^m = x^n   -- avant le while
while m>0
                #Inv 1  -- haut de la boucle
                q:=m/2;  r :=m%2
                if r = 1
                        then  z := z*y ;
                y:=y*y ;
                m:= q
                #Inv 2  -- bas de la boucle
 #Inv
return(z)
```

1-Juste **avant** la boucle :

{**P**} **INIT** { **I** } est correct

On a l'**initialisation**:

$$y = x;\ z = 1 \text{ et } m = n.$$

On remarque que pour cette **initialisation** l'invariant

$$z.\ y^m = x^n.$$

est **vérifié**.

2-Vérifions que **pendant** la boucle

$\{I \wedge C\}$ **CORPS** $\{ I \}$ est correct

C'est à dire que :

si l'invariant I est vraie **en haut** de la boucle: #Inv1,

alors elle le serait **en bas** de la boucle: #Inv2.

Si on se trouve en **haut** de la boucle, ceci signifie que la variable **m** contient un entier strictement positif :

while m>0

On a deux cas à examiner, suivant la parité de m.

Notons z'; y' et m' les valeurs des variables z, y et m en #Inv2 :

1- si m est pair, alors :

$$q = m/2 \; ; \; r=0$$

$$y' = y^2 \; ; \quad m' = q = m/2, \qquad z' = z$$

On a alors:

$$z'.\, y'^{m'} = z.\, (y^2)^{m/2}$$
$$= z\, y^m$$
$$= x^n$$

Puisque $z\, y^m = x^n$ en #Inv1 , c'est toujours le cas en #Inv2 :

$$z'\, y'^{m'} = x^n$$

2-<u>si m est impair</u> alors :

$q = (m-1)/2 \; ; \quad r = 1.$

Dans ce cas :

$z' = z.y \, , \quad y' = y^2 \; ; \quad m' = (m-1)/2$

On a alors :

$$
\begin{aligned}
z' . \, y'^{m'} &= z\, y . \, (y^2)^{(m-1)/2} \\
&= z\, y^m \\
&= x^n
\end{aligned}
$$

Comme $z\,y^m = x^n$ en #Inv1 , c'est toujours le cas en #Inv2 .

$$z'.\,y'^{m'} = x^n$$

Ainsi la propriété :

$$z\,y^m = x^n$$

est préservée en **bas** de la boucle.

I est donc un **invariant** de la boucle **while**.

3-**à la fin** de la boucle :

$\{I \wedge \neg C\}$ **FIN** {**Q**} est correct

On a montré que l'invariant est préservée I chaque itération du **while**.

On en déduit en particulier que cette propriété est vérifiée également **après la sortie** du **while**.

Comme, par ailleurs, les valeurs prises par la fonction **F** définie par :

$$F(m) = m$$

sont à chaque itération du **<u>while</u>** :

- **positives : m >0**

- **strictement décroissantes**

$$m' = [m/2] < m$$

F(m) finira par s'annuler :

$$F(m) = m$$
$$= 0$$

Or la condition

$$m = 0$$

garantit la sortie du while

…et donc la **terminaison** de l'algorithme.

Par ailleurs, comme l'**invariant** ($z\,y^m = x^n$) est vérifié, cela signifie que:

$$\mathbf{m}=0 \Rightarrow z\,y^{\mathbf{0}} = z = x^n$$

Donc, cet l'algorithme, qui retourne z, renvoie bien x^n : il est donc **totalement correct** !

Formalisation de la technique de preuve:

#Inv: invariant de la boucle while

#Inv : propriété vraie **avant** la boucle
while condition:

--montrer que si #Inv est vrai en **haut** de la boucle
[**itération du while**]
--alors #Inv est vrai en **bas** de la boucle

finWhile
-- en déduire que #Inv est vrai **après** la boucle

Exemple de la suite de Fibonacci

<u>Spécification:</u>

La suite de Fibonacci notée **(F_n)** est définie comme suit:

$F_0 = 0$;
$F_1 = 1$;
$F_2 = F_1 + F0$;
…
$F_n = F_{n-1} + F_{n-2}$

```
Fibonacci(n)
if n ≤ 1
            prev := n;
else
            { pprev := 0 ; prev := 1;
              i := 2
            }
while (i ≤ n)
            { f := prev + pprev;
              pprev := prev;    prev := f;
              i := i+1
            }
return (prev)
```

Mise en œuvre de la technique

```
Fibonacci(n)
-- {P}:   {n ≥ 0}
if n ≤ 1
        prev := n;
else
        {pprev := 0 ; prev := 1;
        i := 2
        }
        while (i ≤ n)
            {F := prev + pprev;    pprev := prev;      prev := F;
            i := i+1;
            }
-- {Q} : {prev = Fn}
return (prev)
```

Précondition

$\{P\}$: $\{n \geq 0\}$

Post-condition

$\{Q\}$: {Fibonacci(n) retourne prev = F_n}

Analyse de la condition

$\{n \geq 0 \text{ et } n \leq 1\}$

prev = n

$\{ prev = F_n \}$

Si n=0 alors **prev** =0

Comme $F_0 = 0$ alors {Fibonacci(0) retourne F_0}

Si n=1 alors **prev** =1

Comme $\mathbf{F_1 = 1}$ alors {Fibonacci(1) retourne F_1}

```
{ n > 1 }
pprev := 0      -- F0 ← 0
prev := 1       -- F1 ← 1
i := 2
while (i ≤ n)
        f := prev + pprev;      -- Fi ← Fi-1 + Fn-2
        pprev := prev;          -- Fi-2 ← Fi-1
        prev := f;              -- Fi-1 ← Fi-1 + Fi-2
        i := i + 1
{ prev = Fn }
```

$I = \{pprev = F_{i-2}, \quad prev = F_{i-1}\}$

Analyse de la boucle

$\{n > 1\}$

pprev := 0 ;
prev := 1;
i := 2;

$\{pprev = F_{i-2},\ prev = F_{i-1}\}$

correct

$\{pprev = F_{i-2}\ ,\ prev = F_{i-1},\ \ i \leq n\}$

f := prev + pprev ;

pprev := prev;

prev := f;

i := i+1;

$\{pprev = F_{i-2}\ ,\ \ prev = F_{i-1}\}$

correct

$\{pprev = F_{i-2},\ \ \ prev = F_{i-1},\ \ i = n+1\}$

$\{prev = F_n\}$

Correct

Etude de la terminaison

```
i = 2
while (i ≤ n)
        f := prev + pprev;
        pprev := prev;
        prev := f;
        i := i + 1;
```

Fonction de terminaison:

$$f(i) = n - i + 1$$

- comme $i \leq n$ on a:

$$f(i) = n - i + 1 \mathbf{> 0}$$

- comme $i' = i + 1$ à chaque itération :

  $f(i)$ est **strictement décroissante**

Donc $f(i)$ finira par s'annuler :

$$f(i) = 0$$

$$\Rightarrow n - i + 1 = 0 \Rightarrow i = n+1$$

Cette condition la garantit la sortie du **while**, donc la **terminaison** de l'algorithme.

Conclusion :

L'algorithme est donc **correct** et se **termine : sa correction est totale.**

V. La preuve de correction totale

La preuve de **correction partielle** permet d'avoir une confiance raisonnable dans le fonctionnement un algorithme.

Si un résultat est produit, alors on peut **prouver** qu'il est correct.

Cependant on sait déjà qu’une preuve de correction partielle ne garantit pas qu’un résultat soit produit.

Pour avoir une telle garantie, on a besoin d’une preuve de **correction totale**.

En d’autres termes, il faut aussi prouver que **l’algorithme se termine**.

Nous savons que l'utilisation des **boucles** est au cœur de ce problème de terminaison.

Par conséquent, si on arrive à montrer qu'une **boucle** se termine, on augmenterait notre confiance dans sa **correction totale**.

Terminaison de boucle

```
INIT
while C
CORPS
FIN
```

Un fois qu'on a prouvé que le triplet était **correct**, il faut encore montrer que la boucle se **termine**.

Pour prouver la terminaison, on cherche une **fonction de terminaison** F ou **convergent** :

1-définie sur base des variables de l'algorithme et à valeur dans $\mathbb{N}$ (≥ 0)

2-telle que la valeur de **F** **décroît strictement** suite à l'exécution du corps de la boucle

3-telle que la condition d'arrêt **C** implique **F > 0**

Quel est le raisonnement ?

Puisque la valeur de F **décroît strictement**, elle finira :

- par atteindre 0
- et donc à **infirmer C**.

Exercice d'application

Soit l'algorithme suivant qui calcule n !

(1) lire (n)
(2) i := **2**
(3) fact := 1
(4) **tant que** i $\leq$ n **faire**
(5)   fact := fact * i
(6)   i := i + 1
    fintantque
(7) écrire (fact)

Montrer la correction totale de l'algorithme précédent qui calcule n ! pour les entiers n $\geq$ 1.

Pour établir la **correction totale**, il faut montrer :

1-que l'algorithme est **partiellement correct**,
2-que la boucle **tant que** des lignes (4) à (6) doit terminer.

1-Comment prouver la correction partielle ?

-L'**invariant** de boucle est le suivant :

$$j=i \quad \Rightarrow \text{fact} = (j-1)!$$

Pourquoi ?: à chaque fois qu'on atteint le test de la boucle :

$$i \leq n$$

avec la variable i = j, alors:

$$\text{fact} = (j-1)\,!$$

(1)lire (n)

(2) i := 2

(3) fact := 1

#inv: j=i $\Rightarrow$ fact = (j-1)!

(4) **tant que** i $\leq$ n **faire**

#inv1: i=j $\Rightarrow$ fact = (j-1)! **-- en “haut” de boucle**

(5) fact := fact * i

(6) i := i + 1

#inv2: i=j $\Rightarrow$ fact = (j-1)! **-- en “bas” de boucle**

fintantque

(7) écrire (fact)

Avant la boucle

(1) lire (n)
(2) i := 2
(3) fact := 1
#inv: j=i $\Rightarrow$ fact = (j-1)!

Comme on a:

$$J = i = 2 \Rightarrow (j-1) = 1 = 1!$$

$$fact = 1$$

Il ressort immédiatement que:

$$fact = (j-1)!$$

En haut de la boucle

(4) **tant que** $i \leq n$ **faire**

#inv1: $i=j \Rightarrow fact = (j-1)!$

On suppose qu'en haut de boucle, l'invariant est vrai:

$i=j \Rightarrow fact = (j-1)!$

En bas de la boucle

#inv1: $i=j \Rightarrow fact = (j-1)!$

(5) fact := fact * i

(6) i := i + 1

#inv2: $i'=j' \Rightarrow fact' = (j'-1)!$

<u>fintantque</u>

On a:

$fact' = fact \times i$

$i' = i+1$

$j' = i' \Rightarrow fact' = fact \times i = (j-1)! \times j = j! = (j'-1)!$

Comme la boucle **<u>tant que</u>** termine quand :

$$i = n+1$$

Puisque i est un entier et il est incrémenté de 1 à chaque exécution de la boucle.

Ainsi, quand la ligne (7) est atteinte, on a:

$$i= n+1 \quad \Rightarrow \quad \textcolor{red}{fact} = (n+1-1)! = \textcolor{red}{n!}$$

Ainsi, l'algorithme est correct

2-Comment prouver la terminaison ?

Proposant la fonction de terminaison suivante :

$$F(i) = n-i.$$

F est fonction entière positive car la **condition** d'itération de la boucle est :

$$i \leq n$$

Remarquons qu'à chaque itération :

- **i** := i+1
- **n** reste inchangé.

Ainsi F décroît de 1:

$$F(i+1) = F(i) - 1$$

Quelle que soit la valeur de n, F atteindra forcément la valeur **-1**.

Quand F devient négative :

$$F = n\text{-}i \leq -1$$

On a :

$$n\text{-}i \leq -1 \Rightarrow i \geq n+1.$$

Ainsi, la condition de la boucle $i \leq n$ sera fausse: la boucle **termine**.

Correction totale de l'algorithme d'Euclide

module pgcd(x, y)
{Cet algorithme produit le plus grand commun diviseur des entiers négatif x et y}
tant que y ≠ 0 **faire**
    calculer le reste de x/y
    remplacer x par y
    remplacer y par le reste
{A ce point pgcd(x, y) est égal au pgcd des entrées initiales}
fintantque
sortir x
finmodule

La correction partielle de cet algorithme a été établie précédemment.

Aucune indication concernant sa terminaison n’a été donnée.

Pour prouver sa **terminaison**, il faut trouver le sens dans lequel sa boucle évolue.

En fait, la variable y :

- diminue d'au moins 1
- à chaque passage dans la boucle.

C'est vrai parce que y devient égal au reste de:

x / y.

Ce reste x%y est tel que:

$$0 \leq x\%y \leq y - 1.$$

Donc, la nouvelle valeur de y devient égale à une valeur comprise entre :

0 et **y–1**

C'est pourquoi y devient finalement égal à 0 et que la boucle **termine**.

D'où la **terminaison** de l'algorithme d'Euclide et donc sa **correction totale**.