ÉPREUVE COMMUNE DE TIPE - Partie D

Preuves de programmes

Temps de préparation : ……..….2 h 15 minutes
Temps de présentation devant le jury : …….10 minutes
Entretien avec le jury : ……..10 minutes

GUIDE POUR LE CANDIDAT :

Le dossier ci-joint comporte au total : 10 pages

Travail suggéré :
Il est suggéré au candidat de ne pas chercher à effectuer une présentation exhaustive de ce document, mais plutôt de présenter succinctement les règles de déduction des théorèmes et le principe des preuves d'arrêt, puis de focaliser le reste de la présentation sur un point particulier de la preuve formelle du programme `PGCD`, comme par exemple :

- montrer que l'une des propositions (PGCD(a,b) = PGCD(*A*,*B*)), ou bien ((a>0) ∧ (b>0)), ou encore (max(a,b)>0) sont bien des invariants de la boucle, et sont satisfaits avant l'exécution de celle-ci ;
- établir la preuve d'arrêt de la boucle.

CONSEILS GENERAUX POUR LA PREPARATION DE L'EPREUVE :

* Lisez le dossier en entier dans un temps raisonnable.
* Réservez du temps pour préparer l' exposé devant le jury.

- Vous pouvez écrire sur le présent dossier, le surligner, le découper … mais tout sera à remettre au jury en fin d'oral.
- En fin de préparation, rassemblez et ordonnez soigneusement TOUS les documents (transparents, *etc.*) dont vous comptez vous servir pendant l'oral, ainsi que le dossier, les transparents et les brouillons utilisés pendant la préparation. En entrant dans la salle d' oral, vous devez être prêts à débuter votre exposé.
- A la fin de l' oral, vous devez remettre au jury le présent dossier, les transparents et les brouillons utilisés pour cette partie de l' oral, ainsi que TOUS les transparents et autres documents présentés pendant votre prestation.

INTRODUCTION

Un programme P définit un algorithme qui applique un ensemble de données L sur un ensemble de résultats R. Autrement dit, pour tout $d \in L$ le programme P définit soit une séquence de calcul finie qui donne un résultat $P(d) \in R$, soit une séquence de calcul infinie : le programme "boucle" pour l'entrée d.

D'autre part le programme P a été écrit pour calculer une certaine fonction f de $D \subseteq L$ dans R. P est correct si et seulement si il calcule bien la fonction f, c'est-à-dire si $\forall\ d \in D$, $P(d)$ est défini (c'est-à-dire P ne boucle pas pour l'entrée d) et est égal à $f(d)$. La méthode usuelle pour vérifier qu'un programme est correct est la méthode des tests : on choisit un échantillon fini de données $d_1, \ldots, d_n$, on fait exécuter le programme P pour chacune d'entre elles et on vérifie que $P(d_1) = f(d_1), \ldots, P(d_n) = f(d_n)$. L'insuffisance de cette méthode est évidente dès que l'échantillon testé ne recouvre pas l'ensemble D des données (et c'est pratiquement toujours le cas, même quand D est fini) : il est possible, pour une donnée d n'appartenant pas à l'échantillon testé, que P fournisse un résultat incorrect même si il a fourni un résultat correct pour chaque d_i de l'échantillon. Par suite, la méthode des tests ne permet jamais de prouver qu'un programme est correct. Elle permet tout au plus de prouver éventuellement qu'il est incorrect, si pour une valeur d_i de l'échantillon $P(d_i) \neq f(d_i)$.

La méthode des preuves de programme est, d'un point de vue théorique, bien plus satisfaisante : elle consiste à prouver mathématiquement que le programme P est correct, c'est-à-dire à démontrer un théorème équivalent à : $\forall\ d \in D$, $P(d) = f(d)$.

La méthode de Hoare présentée dans ce document effectue la preuve de la correction d'un programme en deux parties :

- preuve de correction partielle : si le programme s'arrête, il fournit un résultat correct ;
- preuve d'arrêt : le programme s'arrête pour toute donnée $d \in D$.

Cette méthode sera présentée par un exemple, en prouvant un programme simple.

FORMALISATION D'UN PROGRAMME ET HYPOTHESES SIMPLIFICATRICES

Nous définirons ici un programme comme un bloc d'instructions, c'est-à-dire une suite, délimitée par des accolades, d'instructions séparées par des points-virgules :

$$P : \{I_1; \ldots; I_n\}$$

Un programme travaille avec un ensemble V de variables. Dans un souci de simplification, nous ne considérerons que des variables de types simples (*i.e.* entiers, réels, booléens,

caractères) à l'exclusion de tous types structurés (tableaux, chaînes de caractères, listes chaînées, etc.).

Toujours pour simplifier, nous considérerons que les seules instructions élémentaires sont des affectations, de la forme

`x:=`<expression>

ou `x` est un élément de *V* et <expression> représente toute expression arithmétique ou logique qui peut être construite avec diverses constantes, et des variables de *V*, en utilisant des opérateurs arithmétiques et logiques classiques (les quatre opérations, "et" et "ou" logiques, négation logique, opérateurs de comparaisons, etc.). Le résultat de l'exécution de cette affectation est que la variable `x` prend la valeur de l'expression.

Nous n'envisagerons, à l'exclusion de toutes autres, que deux types d'instructions non élémentaires :

- les instructions conditionnelles "si alors sinon", de la forme

  `si` <condition> `alors` <Bloc d'instructions 1> `sinon` <Bloc d'instructions 2>

où <condition> représente toute expression logique construite avec des constantes et des variables de *V* et <Bloc d'instructions 1> (respectivement <Bloc d'instructions 2>) est le bloc d'instructions qui est exécuté par le programme si <condition> est vraie (respectivement fausse) ;

- les boucles "tant que", de la forme

  `tant que` <condition> `faire` <Bloc d'instructions>

où <condition> représente toute expression logique construite avec des constantes et des variables de *V* et <Bloc d'instructions> est le bloc d'instructions qui est exécuté par le programme de manière répétitive tant que <condition> est vraie.

L'EXEMPLE UTILISE

La méthode de preuve de programme sera présentée en effectuant la preuve du programme suivant :

```
Div(a,b) : { r:=a ; q:=0 ;
             tantque r ≥ b faire
                { r:=r-b ;
                  q:=q+1 }
           }
```

a, b, q, r sont des variables entières. a et b sont initialisées (par le passage des paramètres du programme) à des valeurs entières que nous noterons A et B. Div(A, B) calcule, pour $A \in \mathbb{N}$ et $B \in \mathbb{N}^*$ le quotient q et le reste r de la division euclidienne de A par B. On a donc pour Div : $D = \mathbb{N} \times \mathbb{N}^*$, $R = \mathbb{N} \times \mathbb{N}$ et f est l'application de $\mathbb{N} \times \mathbb{N}^*$ dans $\mathbb{N} \times \mathbb{N}$ qui associe à chaque couple (A,B) de $\mathbb{N} \times \mathbb{N}^*$ le couple (q, r) de $\mathbb{N} \times \mathbb{N}$ tel que $A = Bq + r$ et $r < B$.

PREUVES DE CORRECTION PARTIELLE

Il s'agit, dans le cas de Div, de démontrer l'énoncé suivant :

"Si, avant exécution de la première instruction de Div les variables a et b sont initialisées à des valeurs $A \in \mathbb{N}$ et $B \in \mathbb{N}^*$, alors si Div se termine, après son exécution les valeurs de q et r satisfont les conditions $A = Bq + r$, $r < B$, $q \geq 0$, $r \geq 0$ ".

On notera cet énoncé ($\wedge$ représente le "et " logique) :

$$(a = A) \wedge (b = B) \wedge (A \geq 0) \wedge (B > 0)\ [\mathrm{Div}]\ (A = Bq + r) \wedge (q \geq 0) \wedge (r \geq 0) \wedge (r < B).$$

De façon générale, les énoncés que l'on cherchera à prouver au cours d'une preuve de correction partielle seront de la forme E [P] S, où E et S sont des conditions et où P est une instruction, ou une séquence d'instructions, ou un bloc d'instructions (et donc en particulier un programme), la signification étant : " Si E est vraie avant l'exécution de P, alors, si P se termine, S est vraie après exécution de P ". On appellera E précondition et S postcondition de P.

Règles de notation et d'interprétation des énoncés logiques

Toute démonstration consiste à déduire un théorème à partir d'un certain nombre d'axiomes en utilisant des règles de déduction. Dans les preuves de correction partielle, axiomes et théorèmes seront essentiellement des énoncés de la forme E [P] S. Au cours des déductions logiques destinées à démontrer un théorème, les opérateurs logiques utilisés, à savoir la négation, la conjonction (*i.e.* le "et" logique), et la disjonction (*i.e.* le "ou" logique), seront notés différemment selon qu'ils s'appliquent aux théorèmes et/ou axiomes utilisés pour cette démonstration ou bien aux conditions utilisées dans ces mêmes théorèmes et/ou axiomes (en particulier dans les préconditions et les postconditions). Entre théorèmes et/ou axiomes la conjonction sera notée "et", et la disjonction "ou", tandis qu'entre conditions, on notera " $\wedge$" pour la conjonction, "$\vee$ " pour la disjonction, et "$\neg$" pour la négation. De même, l'implication

logique entre deux théorèmes et/ou axiomes H et C sera notée "Si H alors C ", tandis que l'implication logique entre deux conditions C_1 et C_2 sera notée "$C_1 \Rightarrow C_2$".

L'interprétation de ces énoncés devra se faire en appliquant dans l'ordre suivant les opérateurs ci-dessous (qui sont donc classés par ordre de priorité décroissante) :

- tout d'abord, au sein des préconditions et postconditions (et dans cet ordre) les "$\neg$", puis les "$\wedge$", puis les "$\vee$" ;
- ensuite, les opérateurs de type "[P]" entre préconditions et postconditions ;
- enfin (et dans cet ordre) les "et" et les "ou" entre théorèmes ou axiomes.

En cas d'ambiguïté, le parenthésage explicitera l'ordre d'application des opérateurs pour une interprétation correcte de l'énoncé.

Axiomes et règles de déduction pour les preuves de correction partielle

On donne ci-dessous une liste d'axiomes et de règles de déduction permettant de prouver des théorèmes de la forme E [P] S. Pour prouver des relations logiques entre conditions, on utilisera les résultats classiques de logique formelle ainsi que les propriétés des domaines de définition des variables du programme (des entiers dans le cas de Div).

- Axiomes de l'affectation :

Etant donnée une instruction d'affectation x:=<expr> et une post condition S, on a l'axiome E [x:=<expr>] S, où E est obtenue à partir de S par substitution, à toutes les occurrences de la variable x, de l'expression <expr>.

E est la plus faible condition que doivent satisfaire les variables avant l'exécution de l'affectation pour que S soit vraie après.

Exemples : $(xy \geq 0)\ [z:=x*y]\ (z \geq 0)$

$(q+1 \geq 0)\ [q:=q+1]\ (q \geq 0)$

$((x+y)^2 = y)\ [x:=x+y]\ (x^2 = y)$

Remarque importante : grâce à la règle ci-dessus, on peut trouver une précondition d'une instruction d'affectation, étant donnée une postcondition, mais l'inverse n'est pas possible. Par suite, les preuves de programmes se feront dans ce sens : on partira de la fin (postcondition) et on remontera au début (précondition).

- Règle de composition des instructions :

Si E [I_1] F et F [I_2] S, alors E [I_1 ; I_2] S

- Règle du bloc d'instruction : (P représente une séquence d'instructions I_1 ; . . . ; I_n)

Si E [P] S alors E [{P}] S

- Règle de la conditionnelle :

Si $E \wedge C$ [B_1] S et $E \wedge \neg C$ [B_2] S alors E [si C alors B_1 sinon B_2] S

- Règle de la boucle "tant que" :

Si $E \wedge C$ [B] E alors E [tant que C faire B] $E \wedge \neg C$

La condition E est appelée un invariant de la boucle.

En plus des règles ci-dessus, qui permettent donc de construire des énoncés de type E [P] S où P représente un programme tel que nous l'avons formalisé précédemment, on peut donner les règles suivantes, utiles à la déduction de théorèmes :

- Règle de la précondition :

Si E [P] S et ($E' \Rightarrow E$) alors E' [P] S

- Règle de la postcondition :

Si E [P] S et ($S \Rightarrow S'$) alors E [P] S'

- Règle du "et" :

Si E [P] S et E [P] S' alors E [P] $S \wedge S'$

- Règle du "ou" :

Si E [P] S et E' [P] S alors $E \vee E'$ [P] S

Preuve de la correction partielle de `Div`

On veut prouver :

$(a = A) \wedge (b = B) \wedge (A \geq 0) \wedge (B > 0)$ [Div] $(A = Bq + r) \wedge (q \geq 0) \wedge (r \geq 0) \wedge (r < B)$.

On remarque que :

$$(a = A) \wedge (b = B) \wedge (A \geq 0) \wedge (B > 0) \Rightarrow (a = A) \wedge (b = B) \wedge (a \geq 0) \wedge (b > 0)$$

et

$$(a = A) \wedge (b = B) \wedge (a = bq + r) \wedge (q \geq 0) \wedge (r \geq 0) \wedge (r < b)$$
$$\Rightarrow (A = Bq + r) \wedge (q \geq 0) \wedge (r \geq 0) \wedge (r < B).$$

D'après les règles de la précondition et de la postcondition, il suffit de prouver

$(a = A) \wedge (b = B) \wedge (a \geq 0) \wedge (b > 0)$

$$[\text{Div}]\ (a = A) \wedge (b = B) \wedge (a = bq + r) \wedge (q \geq 0) \wedge (r \geq 0) \wedge (r < b), \quad \textbf{(1)}$$

ce que nous démontrerons en deux étapes :

1°) $(a = A) \wedge (b = B)$ [Div] $(a = A) \wedge (b = B)$; **(2)**

2°) $(a \geq 0) \wedge (b > 0)$ [Div] $(a = bq + r) \wedge (q \geq 0) \wedge (r \geq 0) \wedge (r < b)$. **(3)**

En effet, en utilisant les règles de la précondition et du "et ", on peut prouver facilement que si E [P] S et E' [P] S' sont des théorèmes, alors $E \wedge E'$ [P] $S \wedge S'$ est aussi un théorème. Donc, de **(2)** et **(3)**, on déduira **(1)**, ce qui prouvera la correction partielle de Div.

1°) Preuve de **(2)** : $(a = A) \wedge (b = B)$ [Div] $(a = A) \wedge (b = B)$.

Montrons d'abord que $(a = A) \wedge (b = B)$ est un invariant de la boucle.

On a (règle de l'affectation) : $(a = A) \wedge (b = B)$ [q:=q+1] $(a = A) \wedge (b = B)$

$(a = A) \wedge (b = B)$ [r:=r−b] $(a = A) \wedge (b = B)$

donc (règle de composition) : $(a = A) \wedge (b = B)$ [r:=r−b; q:=q+1] $(a = A) \wedge (b = B)$.

Or $(a = A) \wedge (b = B) \wedge (r \geq b) \Rightarrow (a = A) \wedge (b = B)$,

donc (règle de la précondition) :

$(a = A) \wedge (b = B) \wedge (r \geq b)$ [r:=r−b; q:=q+1] $(a = A) \wedge (b = B)$,

dont on déduit (règle de la boucle) :

$(a = A) \wedge (b = B)$ [tant que r≥b faire {r:=r−b; q:=q+1}]

$(a = A) \wedge (b = B) \wedge (r < b)$ **(4)**

En appliquant de nouveau les règles de l'affectation et de la composition, on trouve

$(a = A) \wedge (b = B)$ [r:=a; q:=0] $(a = A) \wedge (b = B)$ **(5)**

A partir de **(4)** et **(5)**, on a (règle de composition, puis règle du bloc d'instructions) :

$(a = A) \wedge (b = B)$ [Div] $(a = A) \wedge (b = B) \wedge (r < b)$,

et enfin, d'après la règle de la postcondition, on déduit **(2)**.

2°) Preuve de **(3)** : $(a \geq 0) \wedge (b > 0)$ [Div] $(a = bq + r) \wedge (q \geq 0) \wedge (r \geq 0) \wedge (r < b)$.

a) Etablissons d'abord que $(a = bq + r) \wedge (q \geq 0) \wedge (r \geq 0)$ est un invariant de la boucle.

On a (règle de l'affectation) :

$(a = b(q+1) + r) \wedge (q+1 \geq 0) \wedge (r \geq 0)$ [q:=q+1] $(a = bq + r) \wedge (q \geq 0) \wedge (r \geq 0)$

et

$(a = b(q+1) + r-b) \wedge (q+1 \geq 0) \wedge (r-b \geq 0)$ [r:=r−b]

$(a = b(q+1) + r) \wedge (q+1 \geq 0) \wedge (r \geq 0)$,

ce qui donne (règle de la composition) :

$(a = b(q+1) + r-b) \wedge (q+1 \geq 0) \wedge (r-b \geq 0)$ [r:=r−b; q:=q+1]

$(a = bq + r) \wedge (q \geq 0) \wedge (r \geq 0)$. **(6)**

Etudions la précondition ainsi obtenue :

(i) $(a = b(q+1) + r-b) \wedge (q+1 \geq 0) \wedge (r-b \geq 0) \Leftrightarrow (a = bq + r) \wedge (q \geq -1) \wedge (r \geq b)$;

(ii) Puisque $(q \geq 0) \Rightarrow (q \geq -1)$ et $(r \geq 0) \wedge (r \geq b) \Rightarrow (r \geq b)$,

on a $(a = bq + r) \wedge (q \geq 0) \wedge (r \geq 0) \wedge (r \geq b) \Rightarrow (a = bq + r) \wedge (q \geq -1) \wedge (r \geq b)$.

L'énoncé **(6)** devient donc (règle de la précondition)

$$(a = bq + r) \wedge (q \geq 0) \wedge (r \geq 0) \wedge (r \geq b)\ [r:=r-b;\ q:=q+1]$$
$$(a = bq + r) \wedge (q \geq 0) \wedge (r \geq 0),$$

dont on déduit (règle de la boucle)

$$(a = bq + r) \wedge (q \geq 0) \wedge (r \geq 0)\ [\texttt{tant que } r \geq b \texttt{ faire } \{r:=r-b;\ q:=q+1\}]$$
$$(a = bq + r) \wedge (q \geq 0) \wedge (r \geq 0) \wedge (r < b), \qquad \textbf{(7)}$$

ce qui établit que $(a = bq + r) \wedge (q \geq 0) \wedge (r \geq 0)$ est un invariant de la boucle.

b) Utilisons maintenant l'invariant de la boucle mis en évidence comme postcondition pour les premières instructions de la séquence.

On a (règle de l'affectation)

$$(a = b.0 + r) \wedge (0 \geq 0) \wedge (r \geq 0)\ [q:= 0]\ (a = bq + r) \wedge (q \geq 0) \wedge (r \geq 0)$$

et $(a = b.0 + r) \wedge (0 \geq 0) \wedge (r \geq 0) \Leftrightarrow (a = r) \wedge (r \geq 0)$.

Appliquons de nouveau la règle de l'affectation :

$$(a = a) \wedge (a \geq 0)\ [r:=a]\ (a = r) \wedge (r \geq 0)$$

et $(a = a) \wedge (a \geq 0) \Leftrightarrow (a \geq 0)$.

D'après la règle de composition, on a

$$(a \geq 0)\ [r:=a;\ q:= 0]\ (a = bq + r) \wedge (q \geq 0) \wedge (r \geq 0) \qquad \textbf{(8)}$$

La règle de composition, appliquée à **(7)** et **(8)**, et la règle du bloc d'instruction donnent

$$(a \geq 0)\ [\texttt{Div}]\ (a = bq + r) \wedge (q \geq 0) \wedge (r \geq 0) \wedge (r < b)$$

et la règle de la précondition permet d'établir **(3)**.

PREUVES D'ARRET

Preuve d'arrêt d'un programme

Un programme tel qu'on l'a défini dans ce document ne peut produire une séquence de calcul infinie que s'il exécute une infinité de fois le corps d'une boucle. Prouver qu'un tel programme s'arrête pour toute donnée $d \in D$ revient donc à prouver que chaque boucle du programme ne peut être exécutée qu'un nombre fini de fois, pour tout $d \in D$.

Soit une boucle "`tant que` C `faire B`" (l'arrêt de l'exécution du bloc d'instructions `B` ayant préalablement été prouvé). Soient x_1, .., x_n les variables du programme. Notons W_E l'ensemble des valeurs du vecteur $w = (x_1, .., x_n)$ telles que les variables vérifient une condition E.

Supposons que la condition E est invariante pour la boucle considérée et satisfaite avant l'exécution de cette boucle. Si $w = w_0 \in W_{E \wedge C}$ avant l'exécution de la boucle, alors $w = w_1 \in W_E$ après la première exécution du corps de boucle `B`. Si $w_1 \in W_{E \wedge (\neg C)}$ la boucle s'arrête, sinon $w = w_2 \in W_E$ après la deuxième exécution du corps de boucle `B`. etc…

Pour prouver l'arrêt de la boucle, il suffit de montrer que toute suite w_0, w_1, … ainsi construite est finie. Une méthode pour démontrer ceci consiste à définir une application m de $W_{E \wedge C}$ dans $\mathbb{N}$ telle que l'on puisse montrer que :

$$E \wedge C \wedge (m(w) = m_0)\ [\mathtt{B}]\ \neg C \vee (m(w) < m_0),\ \forall\ m_0 \in \mathbb{N}.$$

Cet énoncé exprime que si $w = w_i$ avant l'exécution de `B`, alors après exécution de `B`, ou bien la boucle s'arrête, ou bien $m(w_{i+1}) < m(w_i)$. Dans ce cas, l'arrêt de `B` ayant été prouvé, on peut en effet affirmer, pour toute suite w_0, w_1, .., que la suite $m(w_0)$, $m(w_1)$, .., strictement décroissante dans $\mathbb{N}$, est finie. Ceci démontre que la suite w_0, w_1, … est finie et que la boucle s'arrête.

Preuve d'arrêt de `Div`

Le programme `Div` comporte une seule boucle :

```
tant que r ≥b faire {r:=r-b; q:=q+1}.
```

L'exécution du corps de cette boucle s'arrête toujours.

La condition (`b` = B) est un invariant de cette boucle, et est satisfaite avant son exécution (voir preuve de correction partielle, 1°)).

Il en est de même pour la condition (`r` ≥ 0) (voir preuve de correction partielle, 2°)).

Considérons l'ensemble $W_{E\wedge C}$, avec $E = (\texttt{b} = B) \wedge (\texttt{r} \geq 0)$ et $C = (\texttt{r} \geq \texttt{b})$.

Soit l'application m de $W_{E\wedge C}$ dans $\mathbb{N}$ telle que $m(w) = \texttt{r}$. Le choix de cette fonction ramènera donc la preuve de l'arrêt à la démonstration que la variable `r` prend des valeurs successives strictement décroissantes.

D'après le paragraphe précédent, il nous reste à prouver que $\forall\, m_0 \in \mathbb{N}$

$$(\texttt{b} = B) \wedge (\texttt{r} \geq 0) \wedge (\texttt{r} \geq \texttt{b}) \wedge (\texttt{r} = m_0)\ [\texttt{r:=r-b; q:=q+1}]\ (\texttt{r} < \texttt{b}) \vee (\texttt{r} < m_0). \qquad \textbf{(9)}$$

Or d'après la règle de l'affectation

$$(\texttt{r-b} < m_0)\ [\texttt{r:=r-b; q:=q+1}]\ (\texttt{r} < m_0),\ \forall\, m_0 \in \mathbb{N},$$

ce qui conduit, en appliquant les règles de la précondition et de la postcondition à

$$(\texttt{b} = B) \wedge (\texttt{r} \geq 0) \wedge (\texttt{r} \geq \texttt{b}) \wedge (\texttt{r} = m_0) \wedge (\texttt{r-b} < m_0)\ [\texttt{r:=r-b; q:=q+1}]$$
$$(\texttt{r} < \texttt{b}) \vee (\texttt{r} < m_0),\ \forall\, m_0 \in \mathbb{N}. \qquad \textbf{(10)}$$

Or $(\texttt{b} = B) \wedge (\texttt{r} = m_0) \wedge (\texttt{r-b} < m_0) \Leftrightarrow (\texttt{b} = B) \wedge (\texttt{r} = m_0) \wedge (m_0 - B < m_0)$

$$\Leftrightarrow (\texttt{b} = B) \wedge (\texttt{r} = m_0) \wedge (B > 0).$$

La condition $(B > 0)$ est toujours vraie car $(A,B) \in D = \mathbb{N}\times\mathbb{N}^*$, donc

$$(\texttt{b} = B) \wedge (\texttt{r} = m_0) \wedge (\texttt{r-b} < m_0) \Leftrightarrow (\texttt{b} = B) \wedge (\texttt{r} = m_0).$$

Ceci permet de déduire directement **(9)** à partir de **(10)**, et achève donc la preuve de l'arrêt de `Div`.

CONCLUSION

L'exemple précédent illustre à quel point il est difficile et fastidieux de prouver un programme, même très court et très simple. De fait, l'ampleur du travail que constitue la preuve d'un programme déjà écrit limite énormément la portée d'une telle méthode. Le véritable intérêt de ce qui précède apparaît dans une démarche plus efficace : concevoir en même temps le programme et sa preuve. Par exemple, spécifier une boucle par un invariant avant de l'écrire, déterminer le test d'arrêt de cette boucle au vu de sa postcondition, puis écrire le corps de la boucle de façon à ce qu'il préserve l'invariant et ne soit exécuté qu'un nombre fini de fois.

Donnons un petit exemple de cette méthode de programmation avec un programme `PGCD` ayant comme arguments deux nombres entiers non nuls A et B et tel que l'exécution de `PGCD`(A,B) calcule le plus grand commun diviseur de A et B en s'appuyant sur les propriétés suivantes :

- si $A = B$, PGCD(A,B) = $A = B$;
- si $A < B$, PGCD(A,B) = PGCD(A,$B–A$) ;
- si $A > B$, PGCD(A,B) = PGCD($A–B$,B) ;

Le programme aura deux variables `a` et `b` initialisées, par passage de paramètre, à A et à B. Après cette initialisation, la condition PGCD(`a`,`b`) = PGCD(A,B) est satisfaite. Si la suite du programme est une boucle d'invariant PGCD(`a`,`b`) = PGCD(A,B) dont le test d'arrêt est `a` = `b`, on aura, après exécution de cette boucle PGCD(A,B) = PGCD(`a`,`b`) = `a` = `b`. Le résultat recherché sera donc obtenu à la fin de cette boucle avec la valeur de l'une ou l'autre des deux variables `a` et `b`. En utilisant les propriétés du PGCD données ci-dessus, on peut proposer comme corps de cette boucle, l'instruction

```
si a>b alors a:=a–b sinon b:=b–a,
```

qui laisse bien, d'après les propriétés ci-dessus, la proposition PGCD(A,B) = PGCD(`a`,`b`) invariante, tout en diminuant la valeur de l'une ou l'autre des deux variables (ce qui est *a priori* une condition favorable pour la recherche de la suite décroissante pour la preuve d'arrêt).

De façon plus formelle, on peut vérifier que la boucle

```
tant que a≠b faire
   {si a>b alors a:=a–b sinon b:=b–a}
```

a bien comme invariant PGCD(`a`,`b`) = PGCD(A,B) en démontrant (règle de la boucle) :

(PGCD(`a`,`b`) = PGCD(A,B)) $\wedge$ (`a` $\neq$ `b`) [`si a>b alors a:=a–b sinon b:=b–a`]

PGCD(`a`,`b`) = PGCD(A,B) **(11)**

En utilisant la règle de la conditionnelle **(11)** se déduit de **(12)** et **(13)** ci-dessous,

(PGCD(`a`,`b`) = PGCD(A,B)) $\wedge$ (`a` $\neq$ `b`) $\wedge$ (`a>b`) [`a:=a–b`] (PGCD(`a`,`b`) = PGCD(A,B)) **(12)**

(PGCD(`a`,`b`) = PGCD(A,B)) $\wedge$ (`a` $\neq$ `b`) $\wedge$ $\neg$(`a>b`) [`b:=b–a`] (PGCD(`a`,`b`) = PGCD(A,B))**(13)**

qui se démontrent par utilisation des règles de l'affectation et de la précondition.

Enfin, il reste à prouver l'arrêt du programme en déterminant une fonction m, des variables du programme, qui décroît à chaque exécution de la boucle. Informellement, il est clair que les variables `a` et `b` restent strictement positives au cours de l'exécution de la boucle. Il en découle qu'à chaque exécution de la boucle, c'est soit la valeur de `a`, soit la valeur de `b` qui décroît. Une fonction m qui convient est alors

$$m : (\texttt{a},\texttt{b}) \rightarrow \max(\texttt{a},\texttt{b}),$$

qui est donc strictement positive, et qui décroît à chaque exécution de la boucle, ce qui assure l'arrêt de celle-ci.

Formellement, on peut prouver que la proposition $(\mathtt{a} > 0) \wedge (\mathtt{b} > 0)$ est vérifiée avant l'exécution de la boucle, puis, de la même manière que pour la proposition **(11)**, qu'elle est un invariant de celle-ci. On montre de même que $(\max(\mathtt{a},\mathtt{b}) > 0)$ est vérifiée avant l'exécution de la boucle et est un invariant de celle-ci.

En reprenant les notations du document et en prenant

$$E = (\mathtt{a} > 0) \wedge (\mathtt{b} > 0) \text{ et } C = (\mathtt{a} \neq \mathtt{b})$$

on définit donc $m : W_{E \wedge C} \rightarrow \mathbb{N}^*$ telle que $m(w) = \max(\mathtt{a},\mathtt{b})$, et il faut donc prouver

$(\mathtt{a} > 0) \wedge (\mathtt{b} > 0) \wedge (\mathtt{a} \neq \mathtt{b}) \wedge (\max(\mathtt{a},\mathtt{b}) = m_0)$

$[\mathtt{si\ a>b\ alors\ a:=a-b\ sinon\ b:=b-a}]\ (\mathtt{a} = \mathtt{b}) \vee (\max(\mathtt{a},\mathtt{b}) < m_0),\ \forall\ m_0 \in \mathbb{N}^*.$

Or $(\max(\mathtt{a},\mathtt{b}) < m_0) \Rightarrow (\mathtt{a} = \mathtt{b}) \vee (\max(\mathtt{a},\mathtt{b}) < m_0)$, donc en appliquant la règle de la postcondition, ceci se ramène à prouver

$(\mathtt{a} > 0) \wedge (\mathtt{b} > 0) \wedge (\mathtt{a} \neq \mathtt{b}) \wedge (\max(\mathtt{a},\mathtt{b}) = m_0)$

$[\mathtt{si\ a>b\ alors\ a:=a-b\ sinon\ b:=b-a}]\ (\max(\mathtt{a},\mathtt{b}) < m_0),\ \forall\ m_0 \in \mathbb{N}^*,$

ce qui peut se faire en appliquant, comme pour la proposition **(11)**, la règle de la conditionnelle, puis, pour prouver chacune des propositions dont elle découle, comme pour les propositions **(12)** et **(13)**, les règles de l'affectation et de la précondition.