[Aller en bas du fichier](#id.otks08h0sqoo)

**Sémantique**

Enseignant : Eric VIOLARD

Mail : [violard@unistra.fr](mailto:violard@unistra.fr)

Page Web : <http://icps.u-strasbg.fr/~violard/>

Politique de notation : ---------------------- (voir cours)

Date des contrôles : ----------------------- (voir cours)

**Sommaire**

[“Raisonner pour programmer”](#h.7twux19b1qvk)

[1. Quelques problèmes classiques](#h.nzv4nzd8xtee)

[2. “La plus petite somme”](#h.ncp44dgufmel)

[« Sémantique dénotationnelle »](#h.jdgfymdni11d)

[1. Le langage « exp » des expressions arithmétiques de LISP](#h.v429rc2aoa1c)

[2. Le langage “nat” des suites binaires](#h.onv6ri1bu8ij)

[« Sémantique d’un langage impératif »](#h.l5uqca8ouexv)

[1. Définitions sémantiques](#h.lcbd3yqdo9ik)

[2. Sémantique de l’itération](#h.8w5x3wcoejgd)

[« Sémantique des programmes récursifs »](#h.p7g785sl2ycd)

[1. Application du théorème](#h.ylsxglljdui)

[2. D’autres programmes déterministes](#h.38w3p62mgpoi)

[TD “Logique de Hoare”](#h.hfb011z1w8m5)

[1. Quelques extraits de programme](#h.ep54e53nh0aq)

[2. Itération et invariant](#h.fggkwtmh3cih)

[3. Recherche d’invariant](#h.n50xssay9ajr)

[« Weakest pré-conditions »](#h.o3mb013v1181)

[1. Notion de spécification](#h.a7rf2neugdh2)

[2. Sémantique axiomatique](#h.l4a66dvawvtw)

[Sémantique d’une spécification algébrique](#h.z881kl2keavj)

[1. Notion de Sigma-algèbre](#h.s3baekcb1aub)

[2. Notion de modèle](#h.7s1ddxtzolgl)

[3. Modèle initial](#h.37d37sbncgyw)

**Cours**

[TD1 du mardi 17 janvier 2012](#id.3c6xl8eyhxb0)

[TD2 du mardi 24 janvier 2012](#id.ljlps2io2a9w)

[TD3 du mardi 31 janvier 2012](#id.b0oefjdq9g99)

[TD4 du mardi 7 février 2012](#id.5lqz2ksfyjq)

[TD5 du mercredi 22 février 2012](#id.wc13j9wd4zwb)

[TD6 du vendredi 24 février 2012](#id.xg9f6qlank7m)

[TD7 du lundi 19 mars 2012](#id.4j23s4cpocb8)

*(TD du mardi 17 janvier 2012)*

# “Raisonner pour programmer”

## 1. Quelques problèmes classiques

1. Le plus petit nombre (x) d’un tableau t de n nombres

**Données:**

t : tableau de n nombres

**Résultat:**

un nombre x qui est le plus petit nombre du tableau

**énoncé qui qualifie le résultat**

x est tel que

**énoncé qui définie le résultat:**

x(a) = a

x(a::t) = a si a < x(t)

x(t) sinon

Programme impératif en C :

int min(int\* t, int n)

{

int i=0;

int x=t[0];

for(i=1;i<n;i++)

if(t[i]<x) x=t[i];

return x;

}

2. Le quotient (q) de la division euclidienne de 2 nombres entiers a et b

**données:**

a et b deux nombres

**résultat:**

q un nombre qui est le quotion de la division euclidienne de a par b

**qualification du résultat:**

q est tel que:

q\*b <= a

(q+1)\*b>a

**qualification du résultat bis:**

q est tel que:

ou

**définition du résultat:**

si b>a q(a,b)=0

sinon q(a,b)=1+quotient(a-b,b)

**programmation impérative:**

int quotient (int a,int b)

{

int q=0;

while(a>=b)

a=a-1;q++;

return q;

}

3. Le plus grand commun diviseur (p) d’un tableau t de n nombres

**données:**

t un tableau de n nombres

**résultat:**

p un nombre qui est le PGCD de t

**qualification du résultat:**

**définition du résultat:**

p(a) = a

p(a::t) = pgcd(a,p(t))

Th d’Euclide pour calculer le pgcd

pgcd(a,b) = pgcd(b, a mod b) si a >b >= 0

d’où

pgcd(a,b) = a si b = 0

pgcd(b, a mod b) sinon

**programmation impérative:**

int gcd(int a, int b)

{

int r;

while (b!=0)

{

r = a % b;

a = b;

b = r;

}

return a;

}

int pgcd(int t[n])

{

int i; int p;

p = t[0];

for(i=1; i <n; i++)

p = gcd(t[i],p);

return p;

}

4. La racine carrée (r) d’un nombre carré n

**données:**

n un nombre carré

**résultat:**

r un nombre qui est la racine carré de n

**qualification du résultat:**

r est tel que r\*r = n et r > 0

**définition du résultat:**

la suite des carrés peut être calculé par



**programmation impérative:**

int sqroot(int n)

{

int c=0;

int i=0;

for(i=1;c!=m;i++)

c=c+2\*(i-1) +1;

return i-1;

}

5. Le tri d’un tableau t de n nombres (on nomme u le tableau trié)

**données:**

t un tableau de n nombres

**résultat:**

u un tableau contenant les mêmes éléments que t mais triés

**qualification du résultat:**

u est tel que :

**définition du résultat:**

on définit une suite de tableaux

soit k, l’indice du plus petit élément de u(j)[j..n-1]

**programmation impérative:**

## 2. “La plus petite somme”

**donnée:**

t un tableau de n nombre

**résultat:**

s un nombre qui est la plus petite somme d’éléments consécutifs dans t

**qualification du résultat:**

s est tel que:

(en convenant que

**Définition du résultat:**

s(a) = a si a <0

0 sinon

s(a::t) = min(s(t),y(a::t))

avec y(t) = la plus petite somme d’éléments consécutifs dans t mais commencant par le premier élément

y(a) = a si a<0

0 sinon

y(a::t) = min(0, a + y(t))

avec y qui bufferise pour que ca reste linéaire

y en O(n)

s pareil du coup

**programmation impérative:**

int y[n]

int somme\_inter(int t[n])

{

int i;

y[0] = min(t[0], 0);

for(i = 1; i < n; i++)

{

y[i] = min(0, t[i] + y[i-1];

}

}

int s[n];

int petite\_somme(int t[n])

{

int i;

s[0]=min(t[0],0);

for(i=1;i<n;i++)

s[i]=min([s[i-1],y[i]);

}

*(TD du mardi 24 janvier 2012)*

# « Sémantique dénotationnelle »

## 1. Le langage « exp » des expressions arithmétiques de LISP

① Le domaine sémantique du langage « exp » est *R U { ∞ }* (parce que on a des entiers et des réels, donc on va tout utiliser en tant que réel)

On a besoin d’un élément particulier pour donner une sémantique au résultat de la division par 0

② ⟦ \_ ⟧exp : expression syntaxiquement correct → *R U { ∞ }*

e → [[ e ]]exp

⟦nbr⟧ = *nbr*

⟦ (+ e1 e2) ⟧ = ⟦e1⟧ + ⟦e2⟧ si ⟦e1⟧ et ⟦e2⟧

= *∞* sinon

⟦ (/ e1 e2) ⟧ = ⟦e1⟧ / ⟦e2⟧ si ⟦e1⟧ et ⟦e2⟧ et ⟦e2⟧

= *∞* sinon

sémantique structurelle car la définition de la fonction est déduit des règles de grammaire

sémantique compositionelle car la sémantique d’une expression ne dépend que de la sémantique de ses sous-expressions

③ [[ ( \* ( + 2 1 ) ( / 6 3 ) ) ]]expr = [[ + 2 1 ]]expr \* [[ / 6 3 ]]expr

= ([[ 2 ]]expr + [[ 1 ]]expr) \* ([[ 6 ]]expr / [[ 3 ]]expr )=

= (2+1) \* (6/3)

= 3\*2

= 6

[[ (\* (+ 2 1) ( / 6 0) ) ]]exp = ∞

car [[( / 6 0) ) ]]exp  = ∞ ( car ([[ 0 ]]exp = 0)

④ [[ (+ e1 (+ e2 e3 )]] = [[e1]] + ([[e2]] + [[e3]]) si ⟦e1⟧ et ⟦e2⟧ et ⟦e3⟧

= ([[e1]] + [[e2]]) + [[e3]]

= [[(+ (+ e1 e2) e3 )]]

= ∞ sinon

= [[ (+ (e1 e2) e3 ]]exp

## 2. Le langage “nat” des suites binaires

① le domaine sémantique est ℕ

La fonction sémantique est définie par :

[[ b ]]nat = [[ b ]]bin

[[ nb ]]nat = 2 \* [[n]]nat + [[b]]bin

② [[110]]nat = 2 \* [[ 11 ]] + [[ 0 ]]

= 2 \* ( 2 \* [[ 1 ]] + [[ 1 ]] ) + [[ 0 ]]

= 2 \* 2 + 2 + 0

= 6

③ [[succ(0)]]nat = [[ 1 ]]nat (def de recurrence)

= [[ 1 ]]bin

= 1 = 0 + 1 = [[ 0 ]]bin + 1 = [[ 0 ]]nat +1

[[succ(1)]]nat = [[ 10 ]]nat (def de recurrence)

= 2 \* [[ 1 ]]bin + [[ 0 ]]bin

= 2 \* 1 + 0 = 1 + 1 = [[ 1 ]]bin + 1 = [[1]]nat +1

Supposons la propriété vraie pour un certain n de “nat” et considérons la suite binaire n b

[[succ(n0)]]nat = [[ n1 ]]nat (def de recurrence)

= 2 \* [[ n ]]nat + [[ 1 ]]nat = ( 2 \* [[ n ]]nat +0) + [[ 1 ]]bin

= ( 2 \* [[ n ]]nat + [[ 0 ]]bin ) + 1

= ( 2 \* [[ n ]]nat + [[ 0 ]]nat ) + 1 = [[ n0 ]]nat + 1

[[succ(n1)]]nat = [[succ(n)0]]nat (def de reccurence)

= 2 \* [[succ(n)]]nat + [[0]]nat = 2 \* ( [[n]]nat + 1 ) + [[ 0 ]]bin (Hyp)

= ( 2 \* [[n]]nat + 1) + 1 + 0 = ( 2 \* [[ n ]]nat + [[ 1 ]]bin ) + 1

= ( 2 \* [[n]]nat + ⟦ 1 ⟧nat ) + 1 = ⟦ n1 ⟧nat + 1

④

add(0,m) = m

add(n,1) = succ(n)

add(x,y) = add(y,x)

add(x0, y1) = add(x,y)1

add(x0, y0) = add(x,y)0

add(x1, y1) = succ(add(x,y))0

Correction :

add(0,m) = m

add(n,0) = n

add(n, 1) = succ(n)

add(1,m) = succ(m)

add(n0, m0) = add(n,m)0

add(n0, m1) = add(n,m)1

add(n1, m0) = add(n,m)1

add(n1, m1) = succ(add(n,m))0

Démontrer : [[ add(n,m) ]] = [[n]] + [[m]]

Cas de base

[[add(0,m)]] = [[m]] = [[0]] + [[m]]

[[ add(n,0) ]] = [[n]] = [[n]] + [[0]]

[[ add(n,1) ]] = [[succ(n)]] = [[n]] + 1 = [[n]] + [[1]]

[[add(1,m)]] = [[succ(m)]] = [[m]] + 1 = [[1]] + [[m]]

Supposons la propriété vraie pour un certain n et un certain m et considérons les suites binaires n0, n1, m0, m1 :

[[ add(n0, m0)]] = [[add(n,m)0]] = [[add(n,m)]]\*2 + 0 = 2 \* [[n]] + 2\*[[m]] = [[n0]] + [[m0]]

[[ add(n0, m1) ]] = [[add(n,m)1 ]] = ...

[[ add(n1, m0) ]] = [[add(n,m)1 ]] = ...

[[add(n1, m1)]] = [[succ(add(n,m))0]]

= 2 \* [[succ(add(n,m))]] + [[0]]

= 2 [[add(n,m)]] + 2

= (2 [[n]] + 1) + (2 [[m]] +1)

= [[n1]] + [[m1]]

*(TD du mardi 31 janvier 2012)*

# « Sémantique d’un langage impératif »

## 1. Définitions sémantiques

|  |
| --- |
| **Rappels :**  ∀s ϵ S :  ⟦ x:= exp ⟧ (s) = (s | x → ⟦ exp ⟧s )  ⟦ A; B ⟧ (s) = ⟦ B ⟧ ( ⟦ A ⟧ (s) )  ⟦ skip ⟧ (s) = s  ⟦ if b then A else B ⟧ (s) =  ⟦ A ⟧ (s) si ⟦ b ⟧s = vrai  ⟦ B ⟧ (s) si ⟦ b ⟧s = faux |

①

- x:=1; y:=2; z:=x; x:=y; y:=z

∀s ϵ S : ⟦ x:=1; y:=2; z:=x; x:=y; y:=z ⟧ (s) = ⟦ y:=2; z:=x; x:=y; y:=z ⟧ (s | x →1)

= ⟦ z:=x; x:=y; y:=z ⟧ ((s | x →1 ) | y → 2)

= ⟦ z:=x; x:=y; y:=z ⟧ (s | x →1 , y → 2)

= ⟦ x:=y; y:=z ⟧ ((s | x →1 , y → 2) | z → ⟦ x ⟧(s | x →1 , y → 2) )

= ⟦ x:=y; y:=z ⟧ ( s | x → 1, y → 2 , z → 1)

= ⟦ y:=z ⟧ (( s | x → 1, y → 2 , z → 1) | x → ⟦ y ⟧(s | x →1 , y → 2, z → 1) )

= ⟦ y:=z ⟧ (( s | x → 1, y → 2 , z → 1) | x → 2)

= ⟦ y:=z ⟧ ( s | x → 2, y → 2 , z → 1)

= (( s | x → 2, y → 2 , z → 1) | y → ⟦ z ⟧ ( s | x → 2, y → 2 , z → 1))

= (( s | x → 2, y → 2 , z → 1) | y → 1)

= (s | x → 2, y → 1, z → 1)

- ⟦ if x<y then skip else x:=y ⟧

∀s ϵ S : ⟦ if x<y then skip else x:=y ⟧(s) = ⟦ skip ⟧(s) si ⟦ x<y ⟧s = vrai

⟦ x:=y ⟧(s) si ⟦ x<y ⟧s = faux

= s si s(x) < s(y)

(s | x → s(y)) si s(x) >= s(y)

- ⟦ x:=1; n:=3; do n times x:=2\*x ⟧

∀s ϵ S : ⟦ x:=1; n:=3; do n time x:=2\*x ⟧ = ⟦ n := 3; do n times x := 2\*x ⟧ ( ⟦ x := 1 ⟧ (s))

= ⟦ do n times x := 2\*x ⟧( ⟦ n := 3; ⟧ ( ⟦ x := 1 ⟧ (s)))

= ⟦ do n times x := 2\*x ⟧ ( s | x → 1, n → 3)

= ⟦ x := 2 \* x ⟧ ⟦n⟧(s | x → 1, n → 3) (s | x → 1, n → 3)

= ⟦ x := 2 \* x ⟧ 3 ( s | x → 1, n → 3)

= ⟦ x := 2 \* x ⟧ ( ⟦ x := 2 \* x ⟧ ( ⟦ x := 2 \* x ⟧ (( s | x → 1,n → 3))))

= ⟦ x := 2 \* x ⟧ ( ⟦ x := 2 \* x ⟧ (( s | x → 1,n → 3) | x → ⟦ 2\*x ⟧ (s | x → 1, n → 3))

= ⟦ x := 2 \* x ⟧ ( ⟦ x := 2 \* x ⟧ (s | x → 2, n → 3))

= ⟦ x := 2 \* x ⟧ ( s | x → 4, n → 3)

= ( s | x → 8, n → 3)

③

⟦ repeat A until b ⟧ = ⟦ A; while not b do A ⟧

⟦ x1, x2 := exp1, exp2 ⟧ = (s | x1 → ⟦ exp1 ⟧ s , x2 → ⟦ exp2 ⟧ s)

## 2. Sémantique de l’itération

① Sémantique des premières affectations et du corps de l’itération :

∀s ϵ S :

⟦ i := 0; f :=1 ⟧(s) = ⟦ f := 1 ⟧ ( ⟦ i := 0 ⟧ (s))

= (s | f → 1, i → 0)

⟦ i:= i + 1; f := f\*i; ⟧ = ⟦ f := f \* i ⟧ ( ⟦ i = i + 1 ⟧ (s))

= ⟦ f = f \* i ⟧ ( s | i → s(i) + 1 )

= ( s | i → s(i) + 1 ) | f → ⟦ f \* i ⟧ (s | i → s(i) + 1 ))

= ( s | i → s(i) + 1, f → s(f) \* ( s(i) + 1 ) )

Wj+1 = if i < n then

{ i:= i + 1; f := f\*i;

if i < n then

{ i:= i + 1; f := f\*i;

...

if i < n then

{i:= i + 1; f := f\*i;

bottom}

…

}

(avec j + 1 if)

∀s ϵ S :

wj+1(s) = non défini si s(i) < s(n) et s(i)+1 < s(n) et …. et s(i)+j < s(n) donc s(i)+j <s(n)

(s | i → s(i) + j, f → s(f)\*(s(i)+1\*...(s(i)+j)) si s(i) < s(n) et s(i)+1 < s(n) et …. et s(i)+j>=s(n) donc s(i)+j = (n)

…

(s | i → s(i) +1, f → s(f)\*(s(i)+1) si s(i) < s(n) et s(i) + 1 >= s(n) donc s(i)+1 = s(n)

s si s(i) >= s(n)

wj+1(s) = non défini si s(i)+j < s(n)

(s | i → s(i) + j, f → s(f)\*(s(i)+1\*...\*s(n)) si s(i) < s(n) <= s(i) + j

s si s(i) >= s(n)

quand j → +

w(s) = (s | i → s(i) + j, f → s(f)\*(s(i)+1\*...\*s(n)) si s(i) < s(n)

s si s(i) >= s(n)

⟦ i:= 0; f := 1 ; while i<n do { i:=i+1; f:=f\*i } ⟧ (s)

= ⟦ while i < n do {i:=i+1; f:=f\*i} ⟧ ( ⟦ i:=0;f:=1 ⟧ (s))

=w(s|i → 0,f → 1 ) = (s| i → s(n); f → 1 \* (0+1) \* … \* s(n) ) si s(n) >0

s si 0>= s(n)

= (s|i → s(n), f → s(n) ! ) si s(n) >0

s si s(n) <= 0

(2)

Wi+1 = if n<>m then

{ n := n -1; m := m +1;

if n<>m then

{ n := n -1; m := m +1;

….

if n<>m then

{ n:= n -1, m := m + 1, bottom }

Wi+1(s)= non défini si s(n) != s(m) && s(n)-1 !=s(m)+1 && … && s(n) -i !=s(m)+i

(s | n → s(n)-i , m → s(m) +i) si s(n) != s(m) && s(n)-1 != s(m)+1 && … && s(n)-i = s(m)+i

…

(s | n → s(n) -1, m → s(m)+1 ) si s(n) != s(m) && s(n)-1 = s(m) +1

s si s(n) = s(m)

Wi+1(s)= non défini si s(n)-s(m)!= 0,2,...,2i

(s|n → (s(n) +s(m) )/2, m → (s(n) + s(m))/2) si (s(n) - s(m) )/2 = 1

…

(s | n → s(n) + s(m) /2, m → s(n) + s(m) /2) si s(n) -s(m) /2 = 1

s si s(n)=s(m)

Wi+1(s)= non défini si s(n) - s(m) != 0,2,...,2i

(s | n → s(n) + s(m) /2, m → s(n) + s(m) /2) si 0 < s(n)-s(m)/2 <= i && s(n) - s(m) mod 2 =0

s si s(n) = s(m)

quand i → + infini

Wi+1 = non défini si s(n) - s(m) mod 2 != 0

(s | n--> s(n) + s(m) /2, m → s(n) + s(m) /2 si s(n) - s(m) mod 2 =0 && s(n) > s(m)

s si s(n) = s(m)

*(TD du mardi 7 février 2012)*

Wi+1= if k < n then

{

k:= k+1;x:=x+k;

if k<n then

{

k:=k+1;x:=x+k;

…

if k<n then

{

k:=k+1;x:=x+k;

bottom

}

}

}

wi+1(s) = non défini si s(k)<s(n) et s(k)+1<s(n) et …. et s(k)+i<s(n) donc s(k)+i<s(n)

(s|k→s(k)+i, x → s(x)+(s(k)+1)+(s(k)+2)+...+(s(k)+i)

si s(k)<s(n) et s(k)+1<s(n) et … et s(k)+>i=s(n) donc s(k)+i = s(n)

…

(s|k → s(k)+1, x → s(x)+(s(k)+1 si s(k)<s(n) et s(k)+1 donc s(k)+1=s(n)

s si s(k) >= s(n)

wi+1(s) = | non défini si s(k)+i < s(n)

| (s|k->s(n),x → s(x) + (s(k)+1) + (s(k)+2) + … + s(n) si s(k) < s(n) <=s(k)+ i

| s si s(k) >= s(n)

La limite des fonctions sémantiques est :

wi+1(s) = | (s|k->s(n),x |-> s(x) + (s(k)+1) + (s(k)+2) + … + s(n) si s(k) < s(n)

| s si s(k) >= s(n)

appliquée à (s | k → 0 , x → 0) on obtient:

w(s | k → 0, x → 0) = | (s | k → s(n) , x → 0 + (0+1) + (0+2) + … +(0+s(n))) si 0 < s(n)

| s si 0 >= s(n)

donc ce programme calcule la somme des n premiers entiers

# « Sémantique des programmes récursifs »

## 1. Application du théorème

x(k) = if k=0 then 0

else x(k-1) + 2\*k-1

① Préciser le cpo E considéré et la fonctionnelle F déduite du programme.

le cpo considéré est l’ensemble E des fonctions partielles de N dans N muni de la relation d’ordre (ordre d’inclusion des domaines des fonctions partielles)

La fonctionnelle F est définie par :

F : E → E

f → F(f) définie par F(f) : N → N

k → 0 si k=0

f(k-1)+2k-1 sinon

② Montrer que la fonctionnelle F est croissante

Soient et tels que

Il faut montrer que ⇔ |

Soit |

* si k=0 F(g)(k) = 0 = F(f)(k)
* si k>0

puisque

si a

donc

donc

d’autre part :

CGFD

③ Déterminer la limite de la suite

La sémantique du programme “x” est définie par le plus petit point fixe de F

c’est à dire la limite de la suite de fonctions partielles.

* calculer les premiers termes : f0, f1, f2, f3 et f4
* Donner le terme général

...

non défini sinon

* Faire tendre i vers

Cette limite est la fonction f qui à tout associe

## 2. D’autres programmes déterministes

① Déterminer la sémantique du programme récursif :

prod(x, y) = if x=0 then 0

else if x mod 2 = 0 then 2\*prod(x/2, y)

else 2\*prod((x-1)/2, y) + y

Le cpo considéré est l’ensemble E des fonction partielles de dans N

F : E → E

f → F(f) définie par F(f) : N² → N

(x,y) → 0 si (x=0)

2\*prod(x/2,y) si (x mod 2 = 0)

2\*prod((x-1)/2,y)+y sinon

La sémantique du programme “prod” est définie par le plus petit point fixe de F c-a-d la limite de la suite de fonction partielle

c’est-à-dire

non défini sinon

non défini sinon

La limite de cette suite de fonctions partielles est la fonctions qui a tout couple (x,y) associe x\*y.

② Même question pour le programme :

f91(x) = if x>100 then x-10

else f91(f91(x+11))

Le cpo considéré est l’ensemble E des fonctions partielles de Z dans Z

la fonctionnelle F définie par :

F : E → E

f → F(f) définie par F(f) : Z → Z

x → x-10 si x > 100

f(f(x+11)) sinon

La limite des fonctions fi est la fonction f défini par

*(TD du mercredi 22 février 2012)*

# TD “Logique de Hoare”

## 1. Quelques extraits de programme

(1) {x>=0}y:=x+1{y>0}

On utilise l’axiome A1 de l’affectation.

A1 : { q[x\exp] } x:=exp { q } où q[x\exp] est prédicat q mais dans lequel toutes les occurences x ont été substituées par exp.

D’après A1 {x+1>0}y:=x+1{y>0} (en remplacant y par x+1 dans la post-condition y>0)

D’après R4 avec {x>=0} ⇒ {x+1>0}

donc : {x>=0}y:=x+1{y>0}

(2) {vrai} a:=x; b:=x; { a = b }

A1 : {a = x} b:=x; { a = b }

A1 : {x=x} a:=x; {a = x}

R1 : {x=x} a:=x; b:=x; { a = b }

R4 avec {vrai} ⇒ {x=x} donc {vrai} a:=x; b:=x; { a = b } est un théorème.

(3) {a>0} if a>b then b:=a else skip {b>0}

D’après A1 : {a>0}b:=a{b>0} est un axiome

D’après R4 avec a>0 et a>b ⇒ a>0

{a>0 et a>b}b:=a{b>0} est un théorème

D’après A2 {b>0}skip{b>0}

D’après R4 avec a>0 et not(a>b) ⇒ b>0

{ a>0 et not(a>b) } skip {b>0} est un théorème

D’après R2 : {a>0}if a>b then b:=a else skip{b>0} est un théorème.

(4) {a>0}if a>b then b:=a{b>0}

D’après A1 : {a>0}b:=a{b>0} est un axiome

D’après R4 avec a>0 et a>b ⇒ a>0

{a>0 et a>b}b:=a{b>0} est un théorème.

D’après R4(cas particulier) avec a>0 et not(a>b) ⇒ b>0

{a>0}if a>b then b:=a{b>0} est un théorème

(5) {x>a}x:=x+1;x:=x+x;{x> 2a+2}

Résumé de la preuve :

{x>a} ⇒ {x+1+x+1 > 2a+2}

x:=x+1;

{x + x > 2a+2}

x:=x+x;

{x> 2a+2}

## 2. Itération et invariant

(1) Programme de gauche :

on doit montrer que

{ x+y\*b = a\*b}

x:=x+b;

y:=y-1;

{x+y\*b = a\*b} est un théorème

Résumé de la preuve

{ x+y\*b = a\*b} ⇒ {x+b+(y-1)\*b = a\*b}

x:=x+b;

{x+(y-1)\*b = a\*b}

y:=y-1;

{x+y\*b = a\*b}

(2)

D’après R3 :

{ x+y\*b = a\*b}

while y<>0 do

{

x:=x+b;

y:=y-1;

}

{x+y\*b = a\*b et non(y!=0)}

D’après R4 avec x=0 et y = a ⇒ x+y\*b=a\*b

et x+y\*b = a\*b et non(y!=0) ⇒ x = a\*b

{ x=0 et y = a } ⇒ { x+y\*b = a\*b}

while y<>0 do

{

{ x+y\*b = a\*b} ⇒ {x+b+(y-1)\*b = a\*b}

x:=x+b;

{x+(y-1)\*b = a\*b}

y:=y-1;

{x+y\*b = a\*b}

}

{x+y\*b = a\*b et non(y!=0)} ⇒ {x = a\*b}

Résumé de la preuve :

{ x=0 et y = a }

while y<>0 do

{

x:=x+b;

y:=y-1;

}

{x = a\*b}

(1) Programme de droite :

on doit prouver que

{x+a\*y=a\*b}

y:=y-1;

x:=x+a;

{x+a\*y=a\*b}

Résumé de la preuve :

{x+a\*y=a\*b} ⇒ {x+a+a\*(y-1)=a\*b}

y:=y-1;

{x+a+a\*y=a\*b}

x:=x+a;

{x+a\*y=a\*b}

(2)

{x=0 et y=b} ⇒ {x+a\*y=a\*b}

while y<>0 do {

y:=y-1;

x:=x+a;

}

{x+a\*y=a\*b et non y!=0} ⇒ {x=a\*b}

## 3. Recherche d’invariant

Pour trouver l’invariant on sépare la condition finale en p et non p.

Programme de droite :

{a=n+m} se réécrit sous la forme p et non b avec p l’invariant recherché ici et b la condition de l’itération ici b!=0.

Résumé de la preuve:

{vrai} → {n+m = n+m}

a:= n; b:=m;

{a+b=n+m}}

while b<>0 do {

{a+b=n+m /\ b!=0} => {a+1+b-1=n+m}

a:= a+1;

{a+b-1=n+m}

b:= b-1;

{a+b=n+m}

}

{a+b = n+m /\ non(b!=0)} → {a=n+m}

{c=n²} x réécrit {c=i² /\ (i!=n)}

Résumé de preuve :

{vrai} → {0=0²}

c:=0; i:=0;

{ c=i²}

while i<>n do {

{c=i²/\ i!=n} → {i+2i+1= (i+1)²}

c:=c+2\*i+1;

{c=(i+1)²}

i:=i+1;

{c=i²}

}

{c=i²/\ non(i:=n)}

*(TD du vendredi 24 février 2012)*

# « Weakest pré-conditions »

## 1. Notion de spécification

① [

② ou n est un entier

③

## 2. Sémantique axiomatique

|  |
| --- |
| **Rappels :**  wp(x := exp, post) = post[x\exp]  wp(A;B, post) = wp(A, wp(B, post))  wp(if b then A else B, post) = b ⇒ wp(A,post) ⇒ post  wp(while b do A, post) = où P0 = et Pi+1 = wp(A, Pi) i>=0 |

Démontrer :

① wp(t:=x, t=a) = (x=a) par definition

② wp(z:=x,

= ( ) = (

Calculer :

① wp(i:=i+1; j:=j-1, i+j=0)

= wp(i:=i+1, wp(j:=j-1, i+j=0))

= wp(i:=i+1; i+j-1=0)

= (i+1+j-1=0)

= (i+j=0)

② wp( t[t[i]]:=i, t[i] = i )

= ( t[i] = i )[ t[t[i]] \ i )

= ( t[i] = i et i=i ) ou

( t[i] i et i=i )

= ( t[i] = i )

Calculer la plus petite pré-condition : wp( t[t[i]] = i, t[i]=j) = (t[i]=j)[ t[t[i]]\i ]

= ( t[i]=i et i=j)

=( t[i] i et t[i]=j)

③ while n<>m do { n:=n-1;m:=m+1}

wp( while n<>m do {n:=n-1;m:=m+1}, vrai )

= P0 = (nm) vrai = (n=m)

P1 = (nm) wp({n:=n-1;m:=m+1}, n=m)

= (nm) (n-1 = m+1) = (n = m + 2)

P2 = (n=m+4)

…

Pk = (n=m+2k)

Calculer :

① wp( a:=x; b:=y; while b<>1 do { a:=a+x; b:=b-1 }, a =x\*y)

wp(a:=x, wp(b=y, X))

X = wp(while b<>1 do { a:=a+x; b:=b-1 })

P0 = (a =x\*y) = (b<>1)

P1 = ...

...

=

② wp(while k<n do { k:=k+1; x:=x+k }, x= )

X = wp(while k<n do { k:=k+1; x:=x+k }, x= )

P0 =

P1 =

③ wp(x:=0; while k<n do { k:=k+1; x:=x+k }, x= )

*(TD du lundi 19 mars 2012)*

# Sémantique d’une spécification algébrique

## 1. Notion de Sigma-algèbre

① La Num est défine par :

NatNum = IN

BoolNum = { vrai, faux }

zeroNum =

vNum = vrai fNum = faux

succNum = IN → IN

n → n+1

addNum = IN \* IN → IN

(n,m) → n+m

plus-grand-que : IN \* IN → { vrai, faux }

(n,m) → vrai si n > m

faux sinon

(2) La Num4 est défine par :

NatNum4 = {0, 1, 2, 3,4}

BoolNum4 = { vrai, faux }

zeroNum4 = 0 { 0,1 ,2, 3 }

vNum4 = vrai fNum4 = faux

succNum4 = {0, 1, 2, 3,4} → {0, 1, 2, 3,4}

n → (n+1) mod 4

addNum = {0, 1, 2, 3,4} \* {0, 1, 2, 3,4} → { 0, 1, 2, 3}

(n,m) → (n+m) mod 4

plus-grand-que : { 0, 1, 2, 3} \* { 0, 1, 2, 3} → {vrai, faux}

(n,m) → vrai si n>m

faux sinon

(3) Exemple de termes clos :

zero add(zero,zero) add(succ(zero), succ(succ(zero))) plus-grand-que(succ(zero), add(zero,zero))

Soient x, y, z des variables de sorte Nat

Exemple de termes : x add(x,succ(y))

**Tout terme clos est un terme, un terme clos est un terme sans variable.**

Algèbre des termes clos : est définie par :

= ensemble des termes clos de sorte Nat = {zero, succ(zero), add(zero,zero), …}

= ensemble des termes clos de sorte Bool = { vrai, faux , plus-grand-que(zero,zero) }

t → succ(t)

(t1, t2) → add(t1,t2)

plus-grand-que :

(t1, t2) → plus-grand-que(t1, t2)

④ succ(succ(succ(succ(zero))))) = 0

plus-grand-que (succ(succ(succ(succ(zero))))),zero) = plus-grand-que(zero,zero)=FAUX

## 2. Notion de modèle

(1) Num4 ne valide pas l’axiome plus-grand-que(succ(m),zero) = v (on remplace m par succ(succ(succ(zero)))) donc ce n’est pas un modèle.

Num est un modèle.

(2) L’axiome des termes clos quotientée par les axiomes clos est définie par :

= ensemble des classes d’équivalence des termes clos de sorte Nat

= { ZERO, SUCC(ZERO), SUCC(SUCC(ZERO)), … }

ZERO = { zero, add(zero, zero), add(add(zero,zero),zero), …}

SUCC(ZERO) = { succ(zero), add(zero, succ(zero)), add(succ(zero), zero), ...}

Boolclos = ensemble des classes d’équivalence des termes clos de sorte Bool

= { V, F }

V = { v, plus-grand-que(succ(zero), zero)..}

F = { f, plus-grand-que(zero, succ(zero))...}

zeroclos = ZERO

vclos = V fclos = F

succclos : Natclos → Natclos

ZERO → SUCC(ZERO)

SUCC(ZERO) → SUCC(SUCC(ZERO))...

addClos :

(ZERO, ZERO) → ZERO

(ZERO, SUCC(ZERO)) → SUCC(ZERO)

plus-grand-que : Natclos \* Natclos → Boolclos

(ZERo, ZERO) → F

(SUCC(ZERO),ZERO) → V

Clos et Num sont isomorphes.

## 3. Modèle initial

(1) On peut choisir l’algèbre Num définie par :

PNum = {0,1,2,3}

pos\_ecrouNum = 3

visserNum = {0,1,2,3} → {0,1,2,3}

n |→ (n - 1) mod 4

devisserNum = {0,1,2,3} → {0,1,2,3}

n |→ (n + 1) mod 4

Cela vérifie tous les axiomes donc c’est un modèle.

(2)

Si on enlève les deux derniers axiomes, Num n’est plus un modèle initial parce qu’il y a des confusions : par exemple l’élément 3 qui est l’interprétation des termes clos :

pos\_ecrou et visser(visser(visser(visser(pos\_ecrou)))) alors qu’on ne peut pas prouver l’égalité de ces 2 termes clos.

(3) PAlpha = { TRUC, CTRU, UCTR, RUCT }

pos\_ecrouAlpha = TRUC

visserAlpha : PAlpha → PAlpha

TRUC → CTRU

CTRU → UCTR

UCTR → RUCT

RUCT → TRUC

devisserAlpha : PAlpha → PAlpha

CTRU → TRUC

UCTR → CTRU

RUCT → UCTR

TRUC → RUCT

(4) TRUC <-> 3

CTRU <-> 2

...

-- Fin du document ---

*(Ne pas supprimer cette ligne et ne rien ajouter après ! Merci)*