UNIVERSITÉ DE LORRAINE

TELECOM nancy
Ingénieurs du numérique • Inspiring your digital future

Cours MALG & MOVEX

# Vérification mécanisée de contrats (II) (The ANSI/ISO C Specification Language (ACSL))

Dominique Méry
Telecom Nancy, Université de Lorraine
(25 février 2025 at 10:14 A.M.)

**Année universitaire 2024-2025**

# Plan


1. Programs as Predicate Transformers
2. Annotations
3. Contracts
   - Logic Specification
   - Gestion et utilisation des étiquettes pré-définies
     - Validation des annotations (type HOARE)

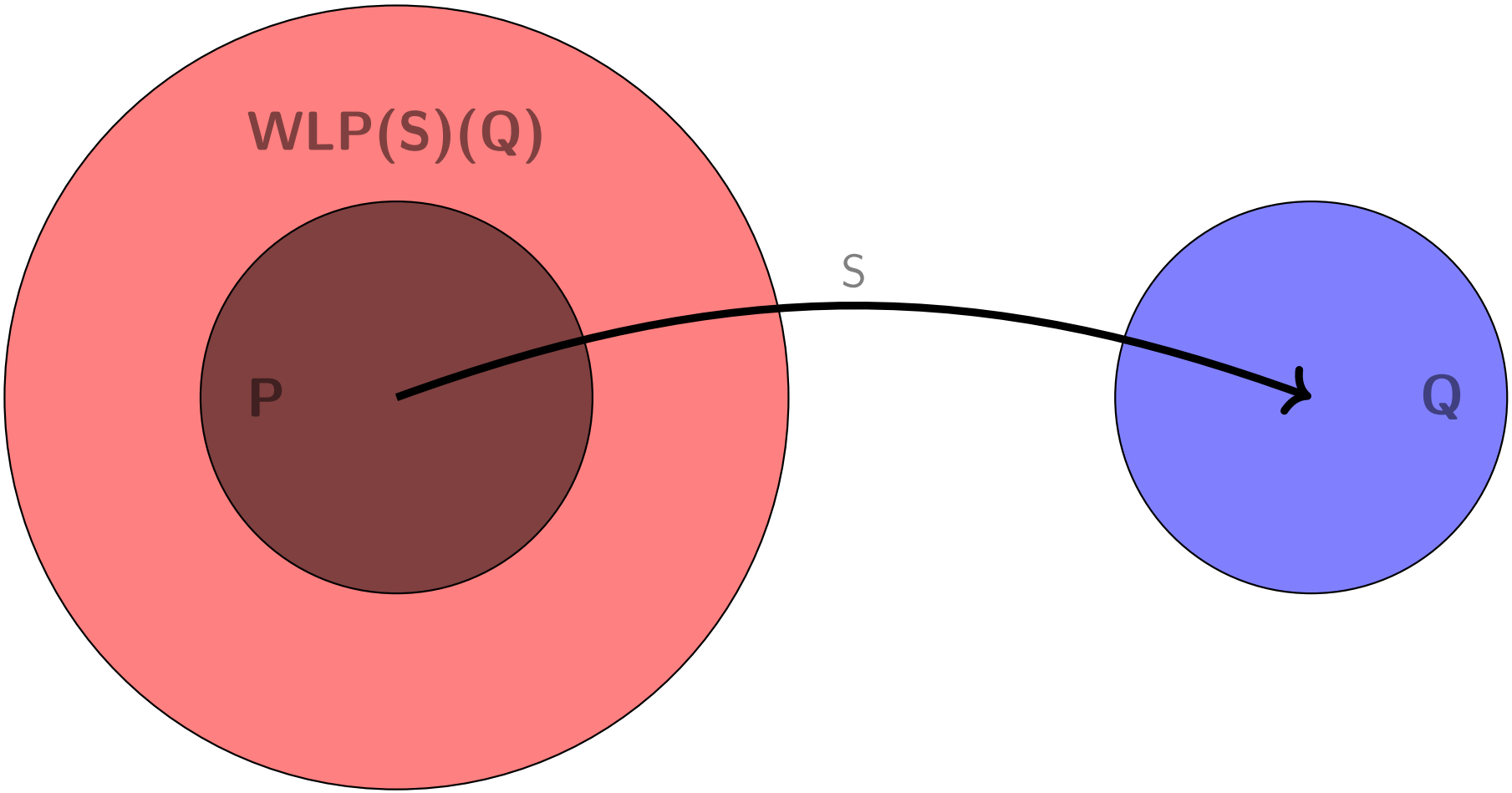
# Sommaire


1. Programs as Predicate Transformers
2. Annotations
3. Contracts
   - Logic Specification
   - Gestion et utilisation des étiquettes pré-définies
     - Validation des annotations (type HOARE)

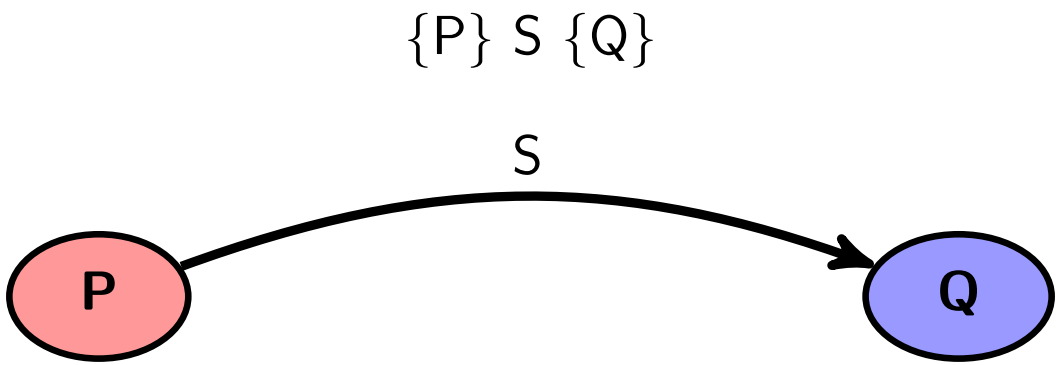
# Sommaire


1. Programs as Predicate Transformers
2. Annotations
3. Contracts
   - Logic Specification
   - Gestion et utilisation des étiquettes pré-définies
     - Validation des annotations (type HOARE)

# Asserted Program {P} S {Q}

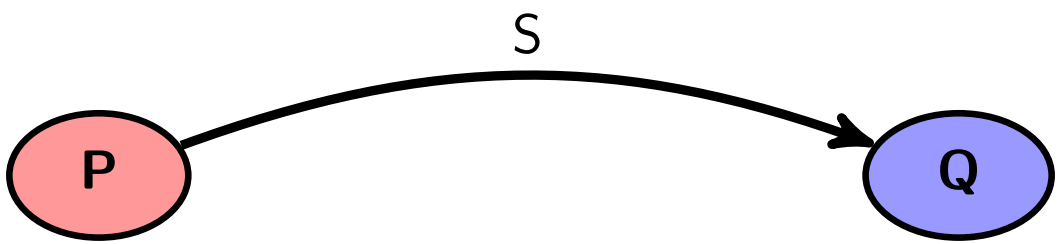


# Asserted Program {P} S {Q}



# Asserted Program {P} S {Q}

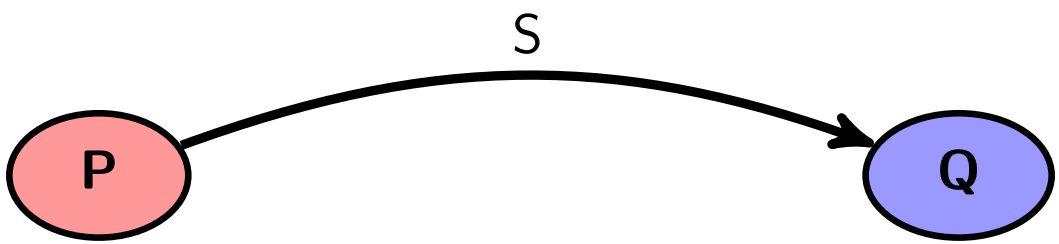
{P} S {Q}




$$P \Rightarrow WLP(S)(\mathsf{Q})$$

# Asserted Program {P} S {Q}

{P} S {Q}




$$P \Rightarrow WLP(S)(\mathrm{Q})$$

Computing WLP(S)(Q) ?

# Writing a simple contract

variables $x$
requires $x >= 0 \wedge x <= 10;$
ensures $\begin{cases} x\%2 = 0 \Rightarrow 2 \cdot \text{result} = x+; \\ x\%2 \neq 0 \Rightarrow 2 \cdot \text{result} = x-1; \end{cases}$

begin
$int\ y;$
$y = x/2;$
$return(y);$
end

- result is the value retuirned by the command `rezturn(y)`.
- `return(y)` is equivalent to result:= y.

(Writing a simple contract.)

Listing 1 – project-divers/annotation.c

```
/*@ requires x <= 0 && x >= 10;
  @ assigns \nothing;
  @ ensures  x % 2 == 0 ==> 2*\result == x;
  @ ensures  x % 2 != 0 ==> 2*\result == x-1;
  @*/
int annotation(int x)
{
  int y;
  y  = x / 2;
  return(y);
  }
```

# Writing a simple contract

(Writing a simple contract.)

Listing 2 – project-divers/annotationwp.c

```
/*@ requires 0 <= x  && x <=  10;
  @ assigns \nothing;
  @ ensures  x % 2 == 0 ==> 2*\result == x;
  @ ensures  x % 2 != 0 ==> 2*\result == x-1;
  @*/
int annotation(int x)
{
/*@ assert x % 2 == 0 ==> 2* (x / 2) == x; */
/*@ assert   x % 2 != 0 ==> 2* (x / 2) == x-1; */
  int y;
/*@ assert x % 2 == 0 ==> 2* (x / 2) == x; */
/*@ assert   x % 2 != 0 ==> 2* (x / 2) == x-1; */
  y  = x / 2;
/*@ assert x % 2 == 0 ==> 2*y == x; */
/*@ assert   x % 2 != 0 ==> 2*y == x-1; */
  return(y);
/*@ assert x % 2 == 0 ==> 2*y == x; */
/*@ assert   x % 2 != 0 ==> 2*y == x-1; */

  }
```

## Property to check

$$x \geq 0 \wedge x \leq 10 \Rightarrow \begin{cases} x\%2 \neq 0 \;\Rightarrow\; 2\cdot(x/2) = x-1 \\ x\%2 = 0 \;\Rightarrow\; 2\cdot(x/2) = x \end{cases}$$

# Writing a simple contract

(Checking the precondition.)

Listing 3 – project-divers/annotation0.c

```
g/*@ requires x >= 0 && x <  0;
  @ assigns \nothing;
  @ ensures \result == 0;
  @*/
int annotation(int x)
{
  int y;
  y = y / (x-x);
  return(y);
  }
```

# Writing a simple contract

(Checking the precondition.)

Listing 4 – project-divers/annotation0wp.c

```
/*@ requires x >= 0 && x <  0;
  @ assigns \nothing;
  @ ensures \result == 0;
  @*/
int annotation(int x)
{
  /*@ assert y / (x-x) == 0; */
  int y;
  /*@ assert y / (x-x) == 0; */
  y  =  y / (x-x);
  /*@ assert y == 0; */
  return(y);
  /*@ assert y == 0; */
  }
```

## Property to check

$$x \geq 0 \wedge < 0 \Rightarrow y/(x-x) = 0$$

# Transformations of annotated programs (1)

$//@$ assert $P(v0, v)$ :
$S1; S2$
$//@$ assert $Q(v0, v)$ :

- Applying the property : $wp(S1; S2)(A) = wp(S1)(wp(S2)(A))$
-

$//@$ assert $P(v0, v)$ :
$S1;$
$//@$ assert $wp(S2)(Q(v0, v))$ :
$S2;$
$//@$ assert $Q(v0, v)$ :

$//@$ assert $P(v0, v)$ :
$//@$ assert $xp(S1)(wp(S2)(Q(v0, v)))$ :
$S1;$
$//@$ assert $wp(S2)(Q(v0, v))$ :
$S2;$
$//@$ assert $Q(v0, v)$ :

# Transformations of annotated programs (2)

//@ assert $P(v0, v)$ :
IF $B$ THEN
  $S1$
ELSE
  $S2$
FI
//@ assert $Q(v0, v)$ :

- Applying the property : $wp(if(B, S1, S2)(A) = b \wedge wp(S1)(A) \vee \neg B \wedge wp(S2)(A)$.
-

//@ assert $P(v0, v)$ :
IF $B$ THEN
  $S1$
ELSE
  $S2$
FI
//@ assert $Q(v0, v)$ :

//@ assert $P(v0, v)$ :
IF $B$ THEN
  $S1$
//@ assert $Q(v0, v)$ :
ELSE
  $S2$
//@ assert $Q(v0, v)$ :
FI
//@ assert $Q(v0, v)$ :

# Transformations of annotated programs (2)

$//@$ assert $P(v0, v)$ :
IF $B$ THEN
  $S1$
$//@$ assert $Q(v0, v)$ :
ELSE
  $S2$
$//@$ assert $Q(v0, v)$ :
FI
$//@$ assert $Q(v0, v)$ :

# Transformations of annotated programs (2)

//@ assert $P(v0, v)$ :
IF $B$ THEN
  $S1$
//@ assert $Q(v0, v)$ :
ELSE
  $S2$
//@ assert $Q(v0, v)$ :
FI
//@ assert $Q(v0, v)$ :

//@ assert $P(v0, v)$ :
IF $B$ THEN
//@ assert $B \wedge wp(S2)(Q(v0, v))$ :
  $S1$
//@ assert $Q(v0, v)$ :
ELSE
//@ assert $\neg B \wedge wp(S2)(Q(v0, v))$ :
  $S2$
//@ assert $Q(v0, v)$ :
FI

# Transformations of annotated programs (2)

//@ assert $P(v0, v)$ :
IF $B$ THEN
  $S1$
//@ assert $Q(v0, v)$ :
ELSE
  $S2$
//@ assert $Q(v0, v)$ :
FI
//@ assert $Q(v0, v)$ :

//@ assert $P(v0, v)$ :
IF $B$ THEN
//@ assert $B \wedge wp(S2)(Q(v0, v))$ :
  $S1$
//@ assert $Q(v0, v)$ :
ELSE
//@ assert $\neg B \wedge wp(S2)(Q(v0, v))$ :
  $S2$
//@ assert $Q(v0, v)$ :
FI

//@ assert $P(v0, v)$ :
IF $B$ THEN
//@ assert $b \wedge wp(S1)(Q(v0, v))$ :
  $S1$
//@ assert $Q(v0, v)$ :
ELSE
//@ assert $\neg b \wedge wp(S2)(Q(v0, v))$ :
  $S2$
//@ assert $Q(v0, v)$ :
FI
//@ assert $Q(v0, v)$ :

# Transformations of annotated programs (2)

//@ assert $P(v0, v)$ :
IF $B$ THEN
  $S1$
//@ assert $Q(v0, v)$ :
ELSE
  $S2$
//@ assert $Q(v0, v)$ :
FI
//@ assert $Q(v0, v)$ :

//@ assert $P(v0, v)$ :
IF $B$ THEN
//@ assert $B \wedge wp(S2)(Q(v0, v))$ :
  $S1$
//@ assert $Q(v0, v)$ :
ELSE
//@ assert $\neg B \wedge wp(S2)(Q(v0, v))$ :
  $S2$
//@ assert $Q(v0, v)$ :
FI

//@ assert $P(v0, v)$ :
IF $B$ THEN
//@ assert $b \wedge wp(S1)(Q(v0, v))$ :
  $S1$
//@ assert $Q(v0, v)$ :
ELSE
//@ assert $\neg b \wedge wp(S2)(Q(v0, v))$ :
  $S2$
//@ assert $Q(v0, v)$ :
FI
//@ assert $Q(v0, v)$ :

- $b \wedge P(v0, v) \Rightarrow$
  $b \wedge wp(S1)(Q(v0, v))$
- $\neg b \wedge P(v0, v) \Rightarrow$
  $\neg b \wedge wp(S2)(Q(v0, v))$

# Transformations of annotated programs (3)

//@ assert $P(v0, v)$ :
//@ loop invariant $I(v0, v)$ :
WHILE $B$ THEN
  $S$
OD
//@ assert $Q(v0, v)$ :

- Applying the iteration rule of Hoare Logic :

# Transformations of annotated programs (3)

//@ assert $P(v0, v)$ :
//@ loop invariant $I(v0, v)$ :
WHILE $B$ THEN
  $S$
OD
//@ assert $Q(v0, v)$ :

//@ assert $P(v0, v)$ :
//@ loop invariant $I(v0, v)$ :
//@ assert $I(v0, v)$ :
WHILE $B$ THEN
//@ assert $b \wedge I(v0, v)$ :
  $S$
//@ assert $I(v0, v)$ :
OD
//@ assert $Q(v0, v)$ :

- Applying the iteration rule of Hoare Logic :

# Transformations of annotated programs (3)

//@ assert $P(v0, v)$ :
//@ loop invariant $I(v0, v)$ :
WHILE $B$ THEN
  $S$
OD
//@ assert $Q(v0, v)$ :

- Applying the iteration rule of Hoare Logic :

//@ assert $P(v0, v)$ :
//@ loop invariant $I(v0, v)$ :
//@ assert $I(v0, v)$ :
WHILE $B$ THEN
//@ assert $b \wedge I(v0, v)$ :
  $S$
//@ assert $I(v0, v)$ :
OD
//@ assert $Q(v0, v)$ :

- $b \wedge I(v0, v) \Rightarrow wp(S)(I(v0, v))$
- $P(v0, v) \Rightarrow I(v0, v))$
- $\neg b \wedge I(v0, v) \Rightarrow Q(v0, v)$

# Summary of transformations

- Checking the preservation of invariant.
- Applying the wps on assertions according to startements.

# Assertions

- Assertions at a control point of the program

```
/*@  assert pred; */

//@  assert pred;
```

- Assertions at a control point of the program components.

```
/*@  for id1,id2, ..., idn: assert pred; */
```

# Verification using WLP

(Incrementing a number)

Listing 5 – project-divers/compwp0.c

```
#define x0 5
/*@ assigns \nothing;*/
int  exemple() {
  int x=x0;
  //@ assert x == x0;
  x = x + 1;
//@ assert  x== x0+1;
return x;
}
```

(Incrementing a number)

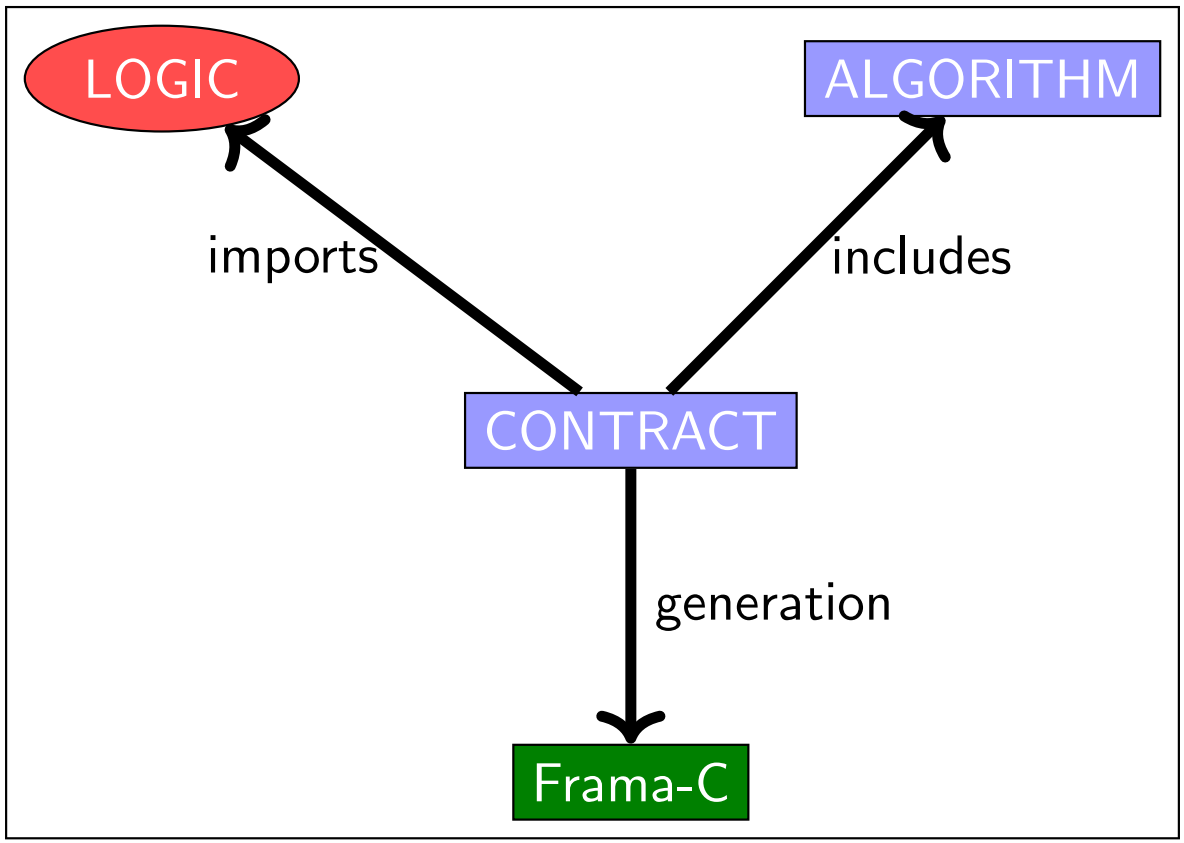
Listing 6 – project-divers/compwp0wp.c

```
#define x0 5
/*@ assigns \nothing; */
int  exemple() {
  //@ assert x0 == x0;
//@ assert  x0+1 == x0+1;
  int x=x0;
  //@ assert x == x0;
//@ assert  x+1 == x0+1;
  x = x + 1;
//@ assert  x== x0+1;
return x;
}
```

# Sommaire des annotations et autres assertions

- requires
- assigns
- ensures
- decreases
- predicate
- logic
- lemma

# Tryptic Pattern



# Defining domain properties in logical theory

- predicate

# Predicates - Logic - Lemma (3)

(Definition of function odd/even)

Listing 7 – project-divers/predicate4.c

```
//@ predicate pair(integer x) = (x/2)*2==x;
//@ predicate impair(integer x) = (x/2)*2!=x;
//@ lemma ex: \forall integer a,b; a < b ==> 2*a < 2*b;

/*@ inductive is_gcd(integer a,integer b , integer c) {
  case zero: \forall integer n; is_gcd(n,0,n);
   case un: \forall integer u,v,w; u >= v ==> is_gcd(u-v,v,w);
   case deux: \forall integer u,v,w; u < v ==> is_gcd(u,v-u,w);
   }
*/
```

# Predicates - Logic - Lemma (1)

(Predicate)

Listing 8 – project-divers/predicate1.c

```
/*@ predicate is_positive(integer x) = x > 0; */

/*@ logic integer get_sign(real x) =   @ x>0.0?1:(x<0.0?-1:0);
*/
/*@ logic integer max(int x,int y) =    x>=y?x:y;
*/
```

(Lemma)

Listing 9 – project-divers/lemma1.c

```
/*@ lemma div_mul_identity:
@ \forall real x, real y; y != 0.0 ==> y*(x/y) == x; @*/

/*@  lemma div_qr:
@ \forall int a, int b; a >= 0 && b >0 ==>
\exists  int q, int r; a == b*q +r && 0<=r && r <b; @*/
```

# Predicates - Logic - Lemma (2)

(Definition of fibonacci function)

## Listing 10 – project-divers/predicate2.c

```
/*@ axiomatic mathfibonacci{
  @ logic integer mathfib(integer n);
  @ axiom mathfib0: mathfib(0) == 1;
  @ axiom mathfib1: mathfib(1) == 1;
  @ axiom mathfibrec: \forall integer n; n > 1
  ==> mathfib(n) ==  mathfib(n-1)+mathfib(n-2);
  @ } */
```

(Definition of gcd)

## Listing 11 – project-divers/predicate3.c

```
/*@ inductive is_gcd(integer a, integer b, integer d) {
@ case gcd_zero:
@ \forall integer n; is_gcd(n,0,n);
@ case gcd_succ:
@ \forall integer a,b,d; is_gcd(b, a % b, d) ==> is_gcd(a,b,d); @}
@*/
```

# Valeur initiale de $x$ $\backslash old(x)$

- $\backslash old(x)$ désigne la valur de la variable $x$ au moment de l'appel de la fonction.
- Cette expression est utilisable uniquement dans la postcondition *ensures*

(Valeur initiale x0)

Listing 12 – project-divers/old1.c

```
/*@ requires \valid(a) && \valid(b);
    @ assigns *a,*b;
     @ ensures  *b == \old(*b)+\old(*a)+2;
     @ ensures  *a == \old(*a)+2;
     @ ensures  \result == 0;
*/
int old(int *a, int *b) {
  int x,y;
  x = *a;
  y = *b;
  x = x +1;
   x = x +1;
   y = y +x;
  *a = x;
  *b = y;
  return 0 ;
}
```

# Expression $\backslash at(e, id)$

- $\backslash at(e, id)$ désigne la valeur de $e$ au point de contrôle $id$.
- $id$ doit être rencontré avant $\backslash at(e, id)$
- $id$ est une expression parmi Pre, Here, Old, Post, LoopEntry, LoopCurrent, Init
- $\backslash old(e)$ est équivalent à $\backslash at(e, Old)$

# Exemple pour $\backslash at(e, id)$

(label Pre)

Listing 13 – project-divers/at1.c

```
/*@
   requires \valid(a) && \valid(b);
   assigns *a,*b;
   ensures  *a == \old(*a)+2;
   ensures  *b == \old(*b)+\old(*a)+2;
*/
int at1(int *a, int *b) {
//@ assert *a == \at(*a,Pre);
  *a = *a +1;
//@ assert *a == \at(*a,Pre)+1;
  *a = *a +1;
//@ assert *a == \at(*a,Pre)+2;
  *b = *b +*a;
//@ assert *a == \at(*a,Pre)+2 && *b == \at(*b,Pre)+\at(*a,Pre)+2;
  return 0;
}
```

-

# Exemple pour $\backslash at(e, id)$

(autre label)

Listing 14 – project-divers/at2.c

```
void f (int n) {
for (int i = 0; i < n; i++) {
/*@ assert \at(i,LoopEntry) == 0; */
int j=0;
while (j++ < i) {
/*@ assert \at(j,LoopEntry) == 0; */
/*@ assert \at(j,LoopCurrent) + 1 == j; */
} }
}
```

- requires
- assigns
- ensures
- decreases
- predicate
- logic
- lemma

# Contrat valide et contrat invalide

(Contrat invalide)

## Listing 15 – project-divers/anno0.c

```
int main(void){
  signed long int x,y,z;
  x = 1;
  /*@ assert x == 1; */
  y = 2;
  /*@  assert x == 1 && y == 2; */
  z = x * y;
  /*@  assert x == 1 && y == 1 && z==2; */
  return 0;
  }
```

(Contrat valide)

## Listing 16 – project-divers/anno00.c

```
int main(void){
  signed long int x,y,z; //     int x,y,z;
  x = 1;
  /*@  assert x == 1; */
  y = 2;
  /*@ assert x == 1 && y == 2; */
  z = x *y;
  /*@  assert  x == 1 && y == 2 && z == 2; */
  return 0;
  }
```

# Invariant de boucle

```
...
/*@  loop invariant I;
  @  loop assigns L;
*/
...
```

(Invariant de boucle)

## Listing 17 – project-divers/anno5.c

```
/*@ requires a >= 0 && b >= 0;
  ensures 0 <= \result;
  ensures \result < b;
  ensures \exists integer k; a == k * b + \result;
*/
int rem(int a, int b) {
  int r = a;
  /*@
    loop invariant
    (\exists integer i; a == i * b + r) &&
    r >= 0;
    loop assigns r;
   */
  while (r >= b) { r = r - b;};
  return r;
}
```

# Exemple d'invariant de boucle

(Invariant de boucle)

Listing 18 – project-divers/anno6.c

```
/*@ requires a >= 0 && b >= 0;
  ensures 0 <= \result;
  ensures \result < b;
  ensures \exists integer k; a == k * b + \result;
*/
int rem(int a, int b) {
  int r = a;
  /*@
    loop invariant
    (\exists integer i; a == i * b + r) &&
    r >= 0;
    loop assigns r;
   */
  while (r >= b) { r = r - b;};
  return r;
}
```

# Résultat de l'analyse

## Echec de la preuve

L'invariant est insuffisamment informatif pour être prouvé et il faut ajouter une information sur y.

```
frama-c -wp anno6.c
[kernel] Parsing anno6.c (with preprocessing)
[wp] Warning: Missing RTE guards
[wp] anno6.c:8: Warning: Missing assigns clause (assigns 'everything' i
[wp] 2 goals scheduled
[wp] [Alt-Ergo 2.3.3] Goal typed_f_loop_invariant_preserved : Timeout (
[wp] [Cache] found:1
[wp] Proved goals:    1 / 2
  Qed:                1  (0.57ms)
  Alt-Ergo 2.3.3:     0  (interrupted: 1) (cached: 1)
[wp:pedantic-assigns] anno6.c:1: Warning:
  No 'assigns' specification for function 'f'.
  Callers assumptions might be imprecise.
```

# Exemple d'invariant de boucle

## Analyse avec succès

L'invariant est plus précis et donne des conditions liant x et y.

(Invariant de boucle)

Listing 19 – project-divers/anno7.c

```
int f() {
int x = 0;
int y = 10;
/*@
    loop invariant
    0 <= x < 11 && x+y == 10;
  */
while (y > 0) {
  x++;
  y--;
}
 return 0;
}
```

# Résultat de l'analyse

```
frama-c -wp anno7.c
[kernel] Parsing anno7.c (with preprocessing)
[wp] Warning: Missing RTE guards
[wp] anno7.c:8: Warning: Missing assigns clause (assigns 'everything' i
[wp] 2 goals scheduled
[wp] [Cache] found:1
[wp] Proved goals:    2 / 2
  Qed:                1  (0.32ms-3ms)
  Alt-Ergo 2.3.3:     1  (6ms) (8) (cached: 1)
[wp:pedantic-assigns] anno7.c:1: Warning:
  No 'assigns' specification for function 'f'.
  Callers assumptions might be imprecise.
```

# Terminaison

- Un variant est une quantité qui décroît au cours de la boucle.
- Deux possibilités d'analyse sont possibles :
  - Terminaison d'une boucle (`variant`)
  - Terminaison de l'appel d'une fonction récursive (`decreawse`)

(Variant)

Listing 20 – project-divers/variant2.c

```
//@ loop variant e;

//@ decreases e;
```

# Terminaison de boucle

- La terminaison est assurée en montrant que chaque boucle termine.
- Une boucle est caractérisée par une expression `expvariant(x)` appelée `variant` qui doit décroître à chaque exécution du corps de la boucle S où $x_1$ et $x_2$ sont les valeurs de $X$ respectiveùent au début de la boucle S et à la fin de S :
  $\forall x_1, x_2. b(x_1) \wedge x_1 \xrightarrow{S} x_2 \Rightarrow \text{expvariant}(x_1) > \text{expvariant}(x_2)$

(Variant)

Listing 21 – project-divers/variant1.c

```
/*@ requires n > 0;
  ensures \result == 0;
*/
int code(int n) {
  int x = n;
  /*@ loop invariant x >= 0 &&  x <= n;
    loop assigns x;
    loop variant x;
   */
  while (x != 0) {
    x = x - 1;
  };
  return x;
}
```

# Exemple de variant de boucle

(Variant)

Listing 22 – project-divers/variant3.c

```
int f() {
int x = 0;
int y = 10;
/*@
    loop invariant
    0 <= x < 11 && x+y == 10;
    loop variant y;
   */
while (y > 0) {
  x++;
  y--;
}
 return 0;
}
```

# Exemple d'appel récursif

(Variant)

Listing 23 – project-divers/variant4.c

```
g/*@ requires n <= 12;
  @ decreases n;
  @*/
int fact(int n){
  if (n <= 1) return 1;
  return n*fact(n-1);
}
```

# Modèle de mémoire HOARE

- Pas de gestion de la mémoire comme les pointeurs
- Affectation à chaque variable une variable logique
- x++ avec x de type int et la C-variable est affectée à deux L-variables $x2 = x1 + 1$.

# Exemples d'annotation

(Variant)

## Listing 24 – project-divers/wp2.c

```
/*@CONSOLE
#include <LIMITS.h>
int q1() {
  int x=10,y=30,z=20;
//@ assert x== 10 && y == z+x && z==2*x;
 y= z+x;
  //@ assert  x== 10 && y == x+2*10;
 x = x+1;
//@ assert  x-1== 10 && y == x-1+2*10;
 return(0);
 }
```

(Variant)

## Listing 25 – project-divers/wp3.c

```
int q1() {
  int c = 2 ;
  /*@ assert c == 2;  */
  int x;
  /*@ assert c == 2;  */
  x = 3 * c ;
  /*@ assert x == 6;  */
  return(0);
}
```

(Variant)

## Listing 26 – project-divers/wp4.c

```
int main()
{
  int a = 42; int b = 37;
  int c = a+b; // i:1
//@assert  b == 37 ;
  a -= c; // i:2
  b += a; // i:3
//@assert  b == 0 && c == 79;
  return(0);
}
```

(Variant)

## Listing 27 – project-divers/wp5.c

```
int main()
{
  int z; // instruction 8
  int a = 4; // instruction 7
//@assert  a == 4 ;
  int b = 3; // instyruction 6
//@assert  b == 3 && a == 4;
  int c = a+b; // instruction 4
/*@ assert  b == 3 && c == 7 && a == 4 ; */
  a += c; // instruction 3
  b += a; // instruction 2
//@ assert  a == 11 && b == 14 && c == 7 ;
//@ assert  a +b  == 25 ;
  z = a*b; // instruction 1
//@assert  a == 11 && b == 14 && c == 7 && z == 154;
  return(0);
}
```

(Variant)

## Listing 28 – project-divers/wp6.c

```
int main()
{
  int a = 4;
  int b = 3;
  int c = a+b; // i:1
  a += c; // i:2
  b += a; // i:3
//@assert  a == 11 && b == 14 && c == 7 ;
  return(0);
}
```

# Listing 29 – project-divers/wp7.c

```
/*@ ensures x == a;
  ensures y == b;
 */
void swap1(int a, int b) {
  int x = a;
  int y = b;
  //@ assert x == a && y == b;
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
  //@ assert x == a && y == a;
}


void swap2(int a, int b) {
  int x = a;
  int y = b;
  //@ assert x == a && y == b;
  x = x + y;
  y = x - y;
  x = x - y;
  //@ assert x == b && y == a;
}

/*@ requires \valid(a);
  requires \valid(b);
  ensures *a == \old(*b);
  ensures *b == \old(*a);
 */
void swap3(int *a, int *b) {
  int tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
}
```