# Vérification déductive de programmes

## VVL

C. Dubois

ENSIIE

# But de la vérification déductive

Etant donnée une spécification formelle S d'un programme P,

donner une preuve mathématique rigoureuse que toute exécution de P satisfait S

P est correct par rapport à S

Que faut-il pour vérifier formellement P ?

- une spécification formelle de P (rigoureuse, mathématique),
- une méthode de preuve de correction (logique de Hoare par exemple, calcul de la *weakest pre-condition*),
- des règles de preuve pour faire des preuves rigoureuses, mathématiques : un calcul.

# Limitations de la vérification déductive

- pas de notion absolue de correction
  La correction est toujours relative à une spécification donnée
- Difficile et coûteux d'écrire des spécifications formelles
  En pratique, on ne spécifie pas formellement toutes les fonctionnalités mais les propriétés de sûreté comme la bonne formation des données (accès hors des bornes, déréférencement du pointeur null, etc.), l'absence d'exceptions non détectées, les parties critiques du logiciel etc.
- Coûteux également de faire des preuves (mais on gagne sur le temps de test)

# Limitations de la vérification déductive

- Il existe des propriétés difficiles ou impossibles à spécifier ressources comme le temps et la mémoire (possible) le comportement de l'utilisateur, l'environnement en général.
- Des programmes vérifiés formellement peuvent planter à l'exécution
  - bugs dans le compilateur ($\rightarrow$ vérification du compilateur C Compcert)
  - bugs dans l'environnement d'exécution ( $\rightarrow$ vérification NICTA d'un micro-kernel SEL4)
  - bugs dans le matériel ($\rightarrow$ preuve de hardware Intel par ex.)

# Design by contract
# Programmation par contrats

# Contrat

- Notion analogue à celle de contrats entre partenaires pour une affaire.
- Qu'est-ce-qu'un contrat ?

  Définition du contrat (Article 1101 du Code civil)

  `Le contrat est une convention par laquelle une ou plusieurs personnes s'obligent, envers une ou plusieurs autres, à donner, à faire ou à ne pas faire quelque chose.`

  Le contrat est ainsi source d'obligations entre les personnes.
  Il spécifie les obligations mais aussi les bénéfices attendus.

- Qui sont les partenaires du contrat ?

**Fournisseur** implémenteur, en C une procédure, en Java, une classe ou une méthode

**Client** principalement un appelant (un objet ou une procédure), ou un utilisateur humain pour un `main()`

**Contrat** une ou plusieurs paires de clauses ensures/requires qui définissent les obligations mutuelles du client et de l'implémenteur

- Contrat : fortement utilisé pour spécifier dans le monde des langages impératifs et OO.
  notion introduite d'abord par B. Meyer en 1992 dans Eiffel, JML (Java), ACSL (C, CEA/LRI, inspiré de JML)

# Contrats pour procédures

- But : Spécification of des procédures/fonctions
- Approche : donner des assertions (i.e. propriétés) su la procédure
  - Précondition
    - doit être vraie à l'entrée
    - assurée par l'appelant
  - Postcondition
    - doit être vraie à la sortie de la procédure
    - assurée par la procédure elle-même si elle termine
    - aucune garantie si la précondition n'est pas satisfaite
    - terminaison peut être garantie ou non

- Precondition(State) $\Rightarrow$ Postcondition(procedure(State))

  Etat ?
  $\Rightarrow$ *snapshot* du système, à chaque point de programme durant l'exécution, ie en C les valeurs des variables et des paramètres

  Pour les programmes OO, plus compliqué (+ valeurs des attributs statiques des classes, les valeurs des variables d'instances des objets . . . )
- Notation :

  `{Precondition} procedure {Postcondition}`

  ou

  ```
  requires Precondition
  ensures Postcondition
  procedure
  ```
- Pre et Post : formules de la logique du 1er ordre.
  Formules qui utilisent les variables globales et les paramètres de la fonction comme variables libres.

```
/**
* @param a an integer
* @returns integer square root of a
**/
int root (int a) {
int i = 0;
int k = 1;
int sum = 1;
while (sum <= a) {
k = k+2;
i = i+1;
sum = sum+k;
}
return i;
}
```

# Spécification de root

Les types sont garantis par le compilateur : a : integer et root : integer (le résultat)

1. `root` vue comme une fonction partielle
   Precondition : $a \geq 0$
   Postcondition : $root * root \leq a < (root + 1) * (root + 1)$
2. `root` vue comme une fonction totale
   Precondition : *true*
   Postcondition :
   $(a \geq 0) \Rightarrow root*root \leq a < (root+1)*(root+1)$
   $\wedge$
   $(a < 0 \Rightarrow root=0)$

- Trouver la plus faible précondition
  i.e. une précondition impliquée par toutes les autres
  le plus large domaine de définition pour la fonction
- Trouver la plus forte postcondition
  i.e. une postcondition qui implique toutes les autres
  une postcondition la plus complète

$\Rightarrow$ `root` (fonction totale) :
`true` est la plus faible précondition possible.
La postcondition peut être renforcée :

$$(root \geq 0) \wedge$$
$$((a \geq 0) \Rightarrow root*root \leq a < (root+1)*(root+1) \wedge$$
$$(a < 0 \Rightarrow root=0))$$

# Exemples de contrats en ACSL

ACSL = langage d'annotations pour programmes C

```
/*@
requires a >= 0;
ensures
    \result * \result <=  a < (\result + 1)*(\result + 1);
*/
int root(int a){
...
}
```

```
/*@ requires \valid(t+(0..n-1)) && n>0;
    ensures \forall integer i; 0<=i<n ==> \result <= t[i];
    ensures \exists integer k;0<=k<n && \result == t[k];
 */
int getMin(int t[], int n) {
...
}
```

Les postconditions sont souvent des prédicats qui portent sur deux états

Exemple : Dans une classe Java counter (dans le langage JML)

```
/*@ requires count >= 0
     ensures count > \old(count)
@*/
increment() { ..
  }
```

# Vérification déductive

# Sémantique axiomatique (logique de Hoare)

Floyd (1967), Hoare(1969), Dijkstra (1978)

Pour les langages impératifs

Sémantique orientée vers la *preuve de programmes*

Avec cette méthode sémantique, un programme est vu comme un transformateur de propriétés logiques

# Le langage

Syntaxe (abstraite) d'un "petit" langage impératif :

Expressions

$$a ::= n \mid x \mid a + a \mid a * a \mid a - a \mid a \div a$$

Conditions

$$b ::= \mathit{true} \mid \mathit{false} \mid a = a \mid a \leq a \mid \ldots$$

Commandes/instructions

$$c ::= \text{skip} \mid x := a \mid c; c \mid \text{if } b \text{ then } c \text{ else } c \mid \text{while } b \text{ do } c$$

# Triplets de Hoare

## Syntaxe

Triplet de Hoare : {P} i {Q} avec $P$, $Q$ des assertions logiques et $i$ une instruction

$P$ est appelée la précondition, $Q$ la postcondition.
Elles constituent la spécification du programme (contrat)

Assertions logiques : formules du premier ordre, avec comme formules atomiques les expressions du langage de programmation

Remarque importante : identification entre les variables du programme et les variables des assertions (si x est une variable du programme, au point de l'assertion, x signifie *valeur de x en ce point*)

Exemple assertion : $x > 0$
$\forall i, 0 \leq i < n \Rightarrow min \leq t(i)$

## Sémantique des triplets

{P} i {Q} se lit

*pour toute exécution, si la propriété P est vraie pour les valeurs des variables du programme avant l'exécution de i et* ***si l'exécution termine*** *alors la propriété Q est vraie après l'exécution de i*

$\rightarrow$ correction partielle

## Sémantique des triplets : un peu plus formellement,

- Un état (mémoire) d'un programme associe une valeur à chaque variable du programme. Un état peut être vu comme une fonction des variables vers les valeurs (ici entières).

```
z := 0;
x := 1;
y := 2; (1)
z := x + y; (2)
```

  Au point de programme (1), l'état $\sigma_1$ est $\{z \mapsto 0; x \mapsto 1; y \mapsto 2\}$
  Au point de programme (2), l'état $\sigma_2$ est $\{z \mapsto 3; x \mapsto 1; y \mapsto 2\}$

- Soit $\sigma$ un état mémoire, soit $P$ une assertion logique, on dit que $\sigma$ satisfait $P$ quand l'interprétation de $P$ est *vraie* lorsque les variables libres de $P$ ont leur valeur courante (dans l'état).
  On écrit alors $\sigma \models P$.
  $\sigma_1$ ne satisfait pas l'assertion $z \geq x$ (qui est fausse en 1),
  $\sigma_2$ satisfait l'assertion $z \geq x$ (qui est vraie en 2).

## Sémantique des triplets : un peu plus formellement,

- {P} i {Q} est vraie (on note $\models \{P\}\ i\ \{Q\}$ ) ssi pour tout état initial $\sigma$ qui valide $P$ ($\sigma \models P$), si l'exécution de $i$ dans $\sigma$ termine, soit $\sigma'$ l'état final résultant, alors $\sigma'$ satisfait $Q$ ($\sigma' \models Q$ )
- Quelques triplets de Hoare valides

  $\models$ {true} x := 5 { x=5 }

  $\models$ { x = y } x := x + 3 { x = y + 3 }

  $\models$ { x $>$ 0 } x := x * 2 { x $>$ -2 }

  $\models$ { x=a } if (x $<$ 0) then x := -x else skip { x=abs(a) }

  $\models$ { i=1} while !(i<=0) do i := i +1 done { i=0 }

  Mais {x$<$0} x := x-3 { x$>$0 } n'est pas valide.
  Mais { i=1 } while (i<=10) do i := i +1 done { i=10} n'est pas valide.

Pour correction totale : autre forme de triplet :

$\models [P]i[Q]$ ssi

*pour toute exécution, si la propriété P est vraie pour les valeurs des variables du programme avant l'exécution de i alors l'exécution termine et la propriété Q est vraie après l'exécution de i*

Correction totale = correction partielle + preuve de terminaison

Dans la suite, nous considérerons la correction partielle

Pourquoi ne pas insister sur la terminaison ?

- Dans certains on ne veut pas la terminaison
- On simplifie le raisonnement
- Si nécessaire, on peut prouver la terminaison séparément. Obligations de preuve supplémentaires

Les postconditions sont souvent des prédicats qui portent sur deux états

Triplets : on utilise une variable logique (n'apparaît pas dans le pgm) qui désignera la valeur de la variable avant l'exécution du programme

$$\{\mathit{count} = \mathbf{n} \wedge \mathit{count} \geq 0\}\ \mathit{increment}\ \{\mathit{count} > \mathbf{n}\}$$

# Exemple de spécification et de programme (Fact)

$\{x = n \wedge n > 0\}$
y := 1 ;
while not(x = 1) do (y := y * x ; x := x - 1) od
$\{y = n! \wedge n > 0\}$

Ici `n` est variable logique (elle permet de se référer à la valeur initiale de `x`)

Comment décider si un triplet de Hoare est valide ?

Se référer à la sémantique opérationnelle du langage : un peu difficile

Utiliser un système de déduction pour démontrer qu'un triplet est correct : Logique de (Floyd-) Hoare

Appliquer un algorithme qui permet de *tester* qu'un triplet de Hoare est valide : calcul de la plus faible précondition (weakest precondition)

## Le langage d'assertions

- Ici le langage des prédicats (logique du 1er ordre).
  $P ::= b | P \wedge P | P \vee P \mid \neg P | P \Rightarrow P | \forall x.P | \exists x.P$
- Toutes les expressions booléennes du langage impératif sont des assertions
- Mélange des variables du programme et de variables logiques.
- Implicitement les variables prennent des valeurs entières.
- On dispose d'une sémantique pour ce langage d'assertions

# Plus faible précondition - Weakest Precondition

But : automatiser les preuves en logique de Hoare

Définition (Plus faible précondition)
Soit $i$ un programme et $Q$ une formule (assertion). On notera $WP(i, Q)$ la plus faible précondition telle que si $i$ termine, alors $i$ termine dans un état qui satisfait $Q$

C'est une précondition qui amène à Q : $\{WP(i, Q)\}\ i\ \{Q\}$ est valide
C'est la plus faible : toute précondition qui amène à Q est plus forte (elle implique donc la plus faible)
Pour tout prédicat $P$, si $\{P\}\ i\ \{Q\}$ alors $P \Rightarrow WP(i, Q)$

**Théorème (correction partielle avec WP) :**
**Pour prouver $\{P\}\ i\ \{Q\}$ il suffit de prouver $P \Rightarrow WP(i, Q)$.**

# Weakest precondition computation

Start from the postcondition and go backwards.

Work backwards, computing weakest preconditions one statement at a time, to derive the weakest precondition for the program as a whole

```
stmt1                                   {c0} stmt2 {c1}
stmt2                                   {c1} stmt2 {c2}
...                          ---->

stmtn  {cn}                             {cn-1} stmtn  {cn}
```

cn = postcondition

c0 = weakest precondtion

$$Pre \Rightarrow c_0 \; ? \; ? \; ? \; ? \; ? \; ?$$

More formally, we define a function WP which takes 2 arguments :

- a program/procedure/function or a fragment of it, *Pr* and
- a formula *Q* which is the postcondition of *Pr*.

*WP* is defined by case on statements.

# WP : the empty statement case

$\{Q\}$ *skip* $\{Q\}$ is a valid triple, for any predicate $Q$.

skip does not change the state ....

$$\boxed{WP(skip, Q) = Q}$$

# WP : the assignment case

$$WP(x := e, Q) = Q[x \leftarrow e]$$

- $Q[x \leftarrow e]$ : it is the formula $Q$ where all the free occurrences of $x$ are replaced by $e$.
  e.g. if $Q$ is $x > 1$, then $Q[x \leftarrow x + 1]$ is $x + 1 > 1$.
- Assignments change the state so we expect Hoare triples for assignments always to reflect that change.
- $WP(x := x + 1, x > 5) = (x + 1) > 5 = x > 4$
- {x $>$ 4} x := x+1 {x $>$ 5} is a valid Hoare triple.

Substitutions

$x[x \leftarrow y] = y$

$z[x \leftarrow y] = z$

$x > y[x \leftarrow y] = y > y$

$(\forall x, x \ mod \ 2 = 0 \Rightarrow x * y = z)[x \leftarrow y] = \forall x, x \ mod \ 2 = 0 \Rightarrow x * y = z$

$(\forall z, z \ mod \ 2 = 0 \Rightarrow x * x = z)[x \leftarrow y] = \forall z, z \ mod \ 2 = 0 \Rightarrow y * y = z$

$(\forall y, x \ mod \ 2 = 0 \Rightarrow x * y = z)[x \leftarrow y] =$
Aie .....
renommage du $y$ lié pour éviter la capture de $y$
$(\forall t, x \ mod \ 2 = 0 \Rightarrow x * t = z)[x \leftarrow y] =$
$(\forall t, y \ mod \ 2 = 0 \Rightarrow y * t = z)$

# WP : the sequence case

$$WP(stm_1; stm_2, Q) = WP(stm_1, WP(stm_2, Q))$$

Compute weakest precondition for 2nd stmt, use as postcondition for 1st stmt

We consider the sequence $stm_1; stm_2$ with the post-condition $Q$.

We work backwards.

We first compute $WP(stm_2, Q)$. Let us call $P$ the predicate $WP(stm_2, Q)$
Then we compute $WP(stm_1, P)$.
The result is the weakest precondition for the sequence $stm_1; stm_2$ with respect to teh post-condiiton $Q$.

Example :
$WP(x := x + 1; x := x + 2, x > 5)$=
$WP(x := x + 1, WP(x := x + 2; x > 5))$=
$WP(x := x + 1, (x + 2) > 5) = WP(x := x + 1, x > 3) =$
$(x + 1) > 3 = x > 2$

Same method for a sequence of 3 statements, 4 statements etc.

# Exercice

Compute the weakest precondition for the following sequence of assignment statements, for the postcondition $b < 0$
a := 2 * b + 1 ; b := a - 3

*Substitute the right hand side of the second assignment in the postcondition to get a postcondition for the first assignment, then substitute the right hand side of the first assignment in that postcondition to get the precondition.*

- $a - 3 < 0$ equivalent to $a < 3$ : weakest precondition for assignment 2
- With this as a postcondition for assignment 1, we obtain $2 * b + 1 < 3$ as the weakest precondition for assignment 1 and then the for sequence.
- It can be simplified into $b < 1$

{$b < 1$} a := 2 * b + 1 ; b := a - 3 {$b < 0$} is a valid Hoare triple.

# Exercice

Requires $x = a \wedge y = b$
Ensures $x = b \wedge y = a$

```
temp := x;
x:= y;
y := temp
```

Let *Prog* this program. We want to prove that the program is correct wrt the contract.

We have to prove that $\{x = a \wedge y = b\}\ Prog\ \{x = b \wedge y = a\}$ is valid.

1. First we compute $W = WP(Prog, x = b \wedge y = a)$, the weakest pre-condtion.
2. Then we'll have to prove that $(x = a \wedge y = b) \Rightarrow W$.

# Conditional

{ ? ? ? } if x $>$ 0 then y := x else y := -x { y $>$ 5 }

Possible preconditions ?
$x > 10$, $x > 5$, $x < -5$

What is the weakest precondition ?
$x < -5 \vee x > 5$.

# WP : the conditional statement case

$$WP(IF\ b\ THEN\ i_1\ ELSE\ i_2\ ENDIF, Q) = (b \Rightarrow WP(i_1, Q)) \wedge (\neg b \Rightarrow WP(i_2, Q))$$

- When a conditional is executed, either $i_1$ or $i_2$ is executed.
- Therefore, if the conditional is to establish $Q$, both $i_1$ and $i_2$ must establish $Q$.
- The choice between $i_1$ and $i_2$ depends on evaluating $b$ in the initial state, so we can also assume $b$ to be a precondition for $i_1$ and $\neg b$ to be a precondition for $i_2$.
- Thus $b$ must imply the weakest precondition of $i_1$ for the postcondition $Q$. And $\neg b$ must imply the weakest precondition of $i_2$ for the postcondition $Q$.

Example : compute the weakest precondition of
*IF* $x > 2$ *THEN* $y := 1$ *ELSE* $y := -1$ *ENDIF*
for the postcondition $y > 0$.

$WP(IF\ x > 2\ THEN\ y := 1\ ELSE\ y := -1\ ENDIF, y > 0) =$
$(x > 2 \Rightarrow WP(y := 1, y > 0)) \wedge (\neg x > 2 \Rightarrow WP(y := -1, y > 0)) =$
$(x > 2 \Rightarrow 1 > 0) \wedge (\neg x > 2 \Rightarrow -1 > 0) \Leftrightarrow$
$(x > 2 \Rightarrow true) \wedge (x \leq 2 \Rightarrow false) \Leftrightarrow$
$(x > 2)$

(if you are not convinced by the last step,
$(x > 2 \Rightarrow true) \wedge (x \leq 2 \Rightarrow false) \Leftrightarrow (\neg x > 2 \vee true) \wedge (\neg x \leq$
$2 \vee false) \Leftrightarrow \neg x \leq 2 \Leftrightarrow x < 2$)

Thus if we execute *IF*.... in a state where $x$ has a value $> 2$ at the end we are ensured to get a $y$ whose value is $> 0$.

What do you think about the following ?

requires $x > 0$
ensures $y > 0$
*IF* $x > 2$ *THEN* $y := 1$ *ELSE* $y := -1$ *ENDIF*

- As computed previously the weakest precondition is $x > 2$.
- But the precondition $x > 0$ does not imply the weakest precondition $x > 2$.
- Thus the program is not correct with respect to its specification.

Exercice 1 : $WP(IF\ b\ THEN\ i\ ENDIF, Q) = ????$

*IF b THEN i ENDIF* is the same instruction as
*IF b THEN i ELSE skip ENDIF*

$WP(IF\ b\ THEN\ i\ ENDIF, Q) =$
$WP(IF\ b\ THEN\ i\ ELSE\ skip\ ENDIF, Q) =$
$(b \Rightarrow WP(i, Q)) \wedge (\neg b \Rightarrow WP(skip, Q)) =$
$(b \Rightarrow WP(i, Q)) \wedge (\neg b \Rightarrow Q)$

Exercice 2 : Prove *IF* $x < y$ *THEN* $m := y$ *ELSE* $m := x$ *ENDIF* computes in $m$ the maximun of $x$ and $y$.

- Give the postcondition of the program and the precondition if any.
  No precondition (or precondition equals *true*) - Postcondition :
  $m \geq x \wedge m \geq y \wedge (m = x \vee m = y)$
- Compute the weakest precondition

```
IF x<y
  THEN  {y ≥ x ∧ y ≥ y ∧ (y = x ∨ y = y)}
          m := y
        {m ≥ x ∧ m ≥ y ∧ (m = x ∨ m = y)}
  ELSE  {x ≥ x ∧ x ≥ y ∧ (x = x ∨ x = y)}
          m := x
        {m ≥ x ∧ m ≥ y ∧ (m = x ∨ m = y)}
ENDIF {m ≥ x ∧ m ≥ y ∧ (m = x ∨ m = y)}
```

  So (after simplifications) the weakest precondition is equivalent to

$$(x < y \Rightarrow y \geq x) \wedge (x \geq y \Rightarrow x \geq y)$$

  which is a valid formula.

- Conclude. The precondition (*true*) implies the weakest precondition.

Exercice 3 :

```
Requires true
Ensures m ≥ 0
m := x;
if m < 0 then m := - m endif;
```

Let *Prog* this program. We want to prove that the program is correct wrt the contract.

# The loop case (1)

WP(while e do i,Q) = no simple formula !

Indeed, while b do i is equivalent to if b then i else skip endif ; while b do i else skip.

**WP(while b do i, Q)** =
WP(if b then i else skip endif ;while b do i else skip, Q)
= (b $\Rightarrow$ WP(i, **WP(while b do i, Q)**)) $\wedge$ ($\neg$b $\Rightarrow$ Q)

Recursive equation (we know how to solve it in domain theory but it is not simple ! !)

Impossible to compute in general, and not usable in practice

Is there anything easier to calculate and use automatically ?
$\longrightarrow$ Verification Conditions (VCs)

# The loop case (2)

The **user** needs to introduce a loop invariant *Inv*

*If executing the body once preserves the truth of the invariant, then executing the body any number of times also preserves the truth of the invariant*

$Inv$
```
while b loop
```
$Inv \wedge b$
```
   i
```
$Inv$
```
endloop
```
$(\neg\ b) \wedge Inv$

Importance of loop invariants not only when proving : developing loop invariants are a powerful way to design and understand algorithms

***Good invariants are hard to find.***
***In general you need to know what a loop does***
***in order to find its invariant***

Some tools are proposed for inferring/guessing invariants.

# Guessing Loop Invariants

Some tricks

- Usually has same form as postcondition
- But depends on loop index j in some way
- Negation of loop condition $\wedge$ invariant $\Rightarrow$ postcondition
- Good guess : replace N with j in postcondition

Consider selection sort :

```
selectionSort(int[] t) {
int min, temp, bar=0;
while (bar < t.length - 1) {
min = indexOfSmallest(t, bar);// find min
temp = t[bar]; t[bar] = t[min]; t[min] = temp;// swap
bar = bar + 1;}}
```

Loop invariant : region before *bar* is sorted
$\forall i.\forall j.0 \leq i < bar \wedge 0 \leq j < bar \wedge i \leq j \Rightarrow t[i] \leq t[j]$

Example
[746832] (first entry, bar=0)
[246837] (second entry, bar=1)
[236847]
[234867]
[234687]
[234678]

region not known to be sorted
region known to be sorted

# Annotated programs and verification conditions

We start with an annotated program (loop invariant and function specifications)

VC(i,Q) verification condition for i wrt Q (postcondition).

VC(i,Q) stronger than WP(i,Q)

But we have :
for P $\in$ *Pre*($i$, $Q$), P $\Rightarrow$ VC(i,Q) and VC(i,Q) $\Rightarrow$ WP(i,Q)
with *Pre*($i$, $Q$)= set of assertions $R$ such that $\{R\}\ i\ \{Q\}$

In practice, it's enough !

VC(skip, Q)=Q

VC(x :=e,Q) = Q[x/e]

VC(i1 ;i2,Q) = VC(i1,VC(i2,Q))

VC(if e then i1 else i2,Q) = (e $\Rightarrow$ VC(i1,Q)) $\wedge$ ($\neg$ e $\Rightarrow$ VC(i2,Q))

VC(while e do invariant Inv i ,Q) =
Inv

*( the invariant must be satisfied at the entry of the loop )*

$\wedge(\forall x_1 \ldots x_m.Inv \wedge e \Rightarrow VC(i, Inv))$

*( the invariant is preserved by an iteration)*

$\wedge(\neg e \wedge Inv \Rightarrow Q)$

*The postcondition must be satisfied at the end of the loop*

($x_1, \ldots x_n$ variables modified in the loop)

Finally :
VC'({P} i {Q}) = P $\Rightarrow$ VC(i,Q)

# How to show Inv is a loop invariant ?

$Inv$
`while b loop`
$Inv \wedge b$
`i`
$Inv$
`endloop`
$(\neg\ b) \wedge Inv$

We have to show that the Hoare triple $\{Inv \wedge b\}\ i\ \{Inv\}$ is valid.

How ?
With the 2-step methodology for example (if there is no loop in $i$) : compute the weakest precondition $W$ of $i$ wrt the postcondition $Inv$ and show that the formula $Inv \wedge b \Rightarrow W$ is true.

# Exemple : retour sur le programme (Fact)

$\{x = n \wedge n > 0\}$
y := 1 ;
while not(x = 1) do (y := y * x ; x := x - 1) od
$\{y = n! \wedge n > 0\}$

Ici `n` est variable logique (elle permet de se référer à la valeur initiale de `x`)

On prend pour invariant de boucle l'assertion
$y * x! = n! \wedge n > 0 \wedge x > 0$. Elle est notée Inv ci-dessous.

VC'({$x = n \wedge n > 0$} Fact {y=n ! $\wedge n > 0$})=

x=n $\wedge$ n>0 $\Rightarrow$ VC(y :=1 ;while ..., y=n ! $\wedge n > 0$) =

x=n $\wedge$ n>0 $\Rightarrow$ VC(y :=1, VC(while ..., y=n ! $\wedge n > 0$)) =

x=n $\wedge$ n>0 $\Rightarrow$ VC(y :=1 ; Inv $\wedge$ ($\forall$ x, y. Inv $\wedge$ not(x=1) $\Rightarrow$ y*x*(x-1) !=n ! $\wedge$ x-1>0 $\wedge$ n>0) $\wedge$ ($\neg$not(x=1) $\wedge$ Inv $\Rightarrow$ y=n ! $\wedge$ n>0)=

x=n $\wedge$ n>0 $\Rightarrow$ (Inv[y $\leftarrow$ 1] $\wedge$ ($\forall$ x, y. Inv $\wedge$ not(x=1) $\Rightarrow$ y*x*(x-1) !=n ! $\wedge$ x-1>0 $\wedge$ n>0) $\wedge$ ($\neg$ not(x=1) $\wedge$ Inv $\Rightarrow$ y=n ! $\wedge$ n>0)[y$\leftarrow$ 1]) =

x=n $\wedge$ n>0 $\Rightarrow$ ((1*x !=n ! $\wedge$ x>0 $\wedge$ n>0) $\wedge$ ($\forall$ x, y. Inv $\wedge$ not(x=1) $\Rightarrow$ y*x*(x-1) !=n ! $\wedge$ x-1>0 $\wedge$ n>0) $\wedge$ ($\neg$not(x=1) $\wedge$ 1*x !=n ! $\wedge$ x>0 $\wedge$ n>0 $\Rightarrow$ 1=n ! $\wedge$ n>0))

La formule est grosse mais elle se décompose en petits morceaux qui peuvent se démontrer séparément.

## Quelques règles utiles pour recoller les triplets de Hoare entre eux

Les règles suivantes sont des règles de la logique de Hoare et se comprennent sans problème en revenant à la sémantique d'un triplet de Hoare

- si $\{P\}\ i1\ \{Q\}$ est valide et si $\{Q\}\ i2\ \{R\}$ est valide alors $\{P\}\ i1; i2\ \{R\}$ est valide
- si $\{P\}\ i\ \{Q\}$ est valide et $Q$ implique $Q'$ est valide alors $\{P\}\ i\ \{Q'\}$ est valide
- si $\{C \wedge Inv\}\ i\ \{Inv\}$ est valide (i.e si $Inv$ est un invariant de la boucle) alors $\{Inv\}\ whileCdoiend\ \{Inc \wedge \neg C\}$ est valide

VC can be adapted to prove the termination of a loop if it is annotated by a variant (total correction)

VC(while e do Invariant $Inv$, Variant $V$ i ,$Q$) =
$Inv$
$\wedge(\forall x_1 \ldots x_m.Inv \wedge e \Rightarrow VC(i, Inv))$
$\wedge(\neg e \wedge Inv \Rightarrow Q)$
$\wedge\forall x_1 \ldots x_m.(Inv \Rightarrow V \in N)$
*The variant is a positive quantity*
$\wedge(\forall x_1 \ldots x_m.Inv \wedge e \wedge V = V_0 \Rightarrow VC(i, V < V_0))$
*... that strictly decreases in an interation*
($x_1, \ldots x_n$ variables modified in i)

## Générateur de condition de vérification VC et triplet de Hoare

- On vérifie que $\{VC(i, Q)\}\ i\ \{Q\}$
  ce qui signifie, informellement, VC(i,Q) est une précondition qui permet d'atteindre Q au final.

  *Démonstration : par induction sur i (essayez !)*
- Théorème fondamental :
  Si on peut prouver la condition de vérification générée par VC'({P} i {Q}) alors le triplet est dérivable (i.e. on a {P} i {Q})

  *Démonstration : elle se fait par induction sur i en montrant qu'il existe une dérivation dont les conditions d'applicabilité sont exactement celles calculées par VC'({P} i {Q})*.
  (essayez !)

Grâce aux programmes annotés et à *VC*, on obtient une méthode réaliste pour faire de la preuve de programmes impératifs :

1. Ecrire un programme contenant nécessairement des assertions sur les boucles et (éventuellement) des assertions sur les points difficiles de la preuve
2. Transmettre ce code annoté à un outil qui engendre les conditions de vérification (i.e. qui implante la fonction VC ou VC')
3. Prouver ces conditions de vérification (de préférence avec un prouveur automatique (simplify, Z3, Zenon par exemple) ou interactif (Coq, Isabelle) par exemple)

# Vérification modulaire d'un programme

- Vérification fonction par fonction, classe par classe, méthode par méthode.
- Soit $f$ une fonction et $P_f/Q_f$ son contrat ($P_f$ pré-condition, $Q_f$ post-condition).
  On prouve le triplet $\{P_f)\}\ f\ \{Q_f\}$.
  Soit $g$ (de contrat $P_g/Q_g$) une fonction qui appelle $f$
  Dans la preuve de $g$, comment va-t-on traiter l'appel à $f$ ?
  1. Avant l'appel à $g$, il faut vérifier que la précondition de $f$, soit $P_f$ est satisfaite
     $\longrightarrow$ génération de la condition de vérification correspondante
  2. Après l'appel à $f$, la postcondition de $f$ est supposée vraie pour continuer la preuve de correction de $g$.
- Preuve modulaire :
  - on ne vérifie pas le code de $f$ à chaque appel
  - on a juste besoin du contrat de $f$ lors de l'appel à $f$

Pour aller plus loin sur les conditions de vérification :

- Les conditions de vérification sont conservées après une optimisation à la compilation
- Elles peuvent servir pour vérifier la correction d'une optimisation. Utilisables aussi bien sur du code source et du code bas niveau
- Conservation des obligations de preuve entre JML/Java et BML/bytecode Java
  *Gilles Barthe, Benjamin Grégoire, Mariela Pavlova : Preservation of Proof Obligations from Java to the Java Virtual Machine. IJCAR 2008 : 83-99*