Objetivo

Dada una especificación S de un programa P, proveer una demostración rigurosa de que cualquier ejecución de P cumple con S

Dada una especificación S y un programa P generar un conjunto de fórmulas lógicas que, de ser derivables en base a un conjunto de axiomas y reglas de inferencia, implican que P cumple con S. Los predicados pueden ser demostrados de manera manual, con demostración semiautomática o con demostradores automáticos de fórmulas No poder demostrar la fórmula no implica necesariamente que el programa contenga un error. Es decir, el verificador puede devolver proven (true o false) o unknown.

Ejemplo de un contrato

```
//@ requires x > 0
//@ ensures result*result == x
float raizCuadrada(int x) {
    return Math.sqrt(x);
}
```

Para esto necesitamos traducir el programa y el contrato a una lógica en común. Una posibilidad es representar la semántica del programa con axiomas. El programa es un teorema del conjuntos de axiomas y reglas de inferencia

Hoare

Triplas y lógica

{A} codigo {B}: Si el programa comienza con estado A y la ejecución del código termina, entonces el estado final cumple B. (Notar correctitud parcial: SI el codigo termina entonces . . . , total se obtiene probando que termina).

```
float hypo(float leg_a, float leg_b) {
  // @requires: leg_a > 0
  // @requires : leg_b > 0
  // @ensures: result > 0
   return Math.sqrt(leg_a**2 + leg_b**2)
}
```

```
{leg_a > 0 & leg_b > 0}
ret math.sqrt(leg_a**2 + leg_b**2)
{result >0}
```

provee un conjunto de reglas deductivas para probar la correctitud de programas respecto a la estructura de sentencias de programas imperativos. La lógica de Hoare modela cómo cada operación modifica el estado del sistema en término de predicados lógicos.

Lenguaje y reglas

```
skip    // nada
x:=e    // asignación
s1;s2   // secuencia

if (cond) {s1} else {s2}     // condicional
while (cond) {s}             // iteración
```

- Backward (bw): sirve para calcular la Weakest Precondition (para {P} S {Q} si $\forall P'(P' \implies P)$ entocnes P es la WP) > una tripla es válida si la precondición es más fuerte que la WP ($P \implies WP$)
- Forward (fw): sirve para calcular la Strongest Postcondition (para {P} S {Q} si $\forall Q'(Q \implies Q')$ entonces Q es la SP) > una tripla es válida si la postcondición es más debil que la SP ($SP \implies Q$) (Es más dificil de computar)

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}$$

$$\frac{\{A\}\ s1\ \{C\}\quad \{C\}\ s2\ \{B\}}{\{A\}\ \texttt{s1;s2}\ \{B\}}$$

$$\frac{\{A\ \&\&\ cond\}\ s1\ \{B\}\quad \{A\ \&\&\ !cond\}\ s2\ \{B\}}{\{A\}\ \texttt{if(cond) \{s1\} else \{s2\}}\ \{B\}}$$

$$\frac{\{A\ \&\&\ cond\}\ body\ \{A\}\quad (A\ \&\&\ !cond) \Rightarrow B}{\{A\}\ \texttt{while(cond) \{body\}}\ \{B\}}$$

$$\frac{}{\{[b/x]A\}\ \mathbf{x := b}\ \{A\}}\ \text{(Bw)}$$

$$\frac{}{\{A\}\ \mathbf{x := b}\ \{\exists x' \mid A[x'/x]\ \&\&\ x{=}{=}b[x'/x]\}}\ \text{(Fw)}$$

A[b/x] -> reemplazar x en A por b

Regla **forward**:

{ A } **x := E** {∃x' | A[x'/x] && x==E[x'/x]}

- Intuición: x' es el valor anterior de x. (\old(x))

{ x>=3 } **x := x+2** {∃x' | (x>=3)[x'/x] && x == (x+2)[x'/x]}

{ x>=3 } **x := x+2** {∃x' | x'>=3 && x == x'+2}

{ x>=3 } **x := x+2** {∃x' | x'>=3 && x-2== x'}

{ x>=3 } **x := x+2** { x-2>=3 }

{ x>=3 } **x := x+2** { x>=5 }

Regla **backward**:

{ B[x→E] } x:= E {B}

- Intuición: Para que B valga luego de la asignación, la precondición debe tomar en cuenta la relación previa entre x y E

▪Ejemplo:

{?} **x:=4** {x==4}
{x==4[x→4]} **x:=4** {x==4}
{4==4} **x:=4** {x==4}
{true} **x:=4** {x==4}

▪Otro ejemplo:

{?} **x:= x+2** {x>=5}
{x>=5[x→x+2]}
x:=x+2{x>=5}
{x+2>=5} **x:=x+2** {x>=5}
{x>=3} **x:=x+2** {x>=5}

Verfication Condition

Dado un contrato M $(pre_M, post_M)$, calcular una formula lógica que permita inferir la correctitud del programa. (Si la fórmula es verdadera, el programa cumple el contrado)

- Forward: Dado $\{pre_M\}$ S $\{post_M\}$ calculo $SP(S, pre_M)$ y verifico que $SP(S, pre_M) \implies \{post_M\}$, es decir que la postcondición más fuerte que puedo armar con la pre_M sea más fuerte que lo que pide la $post_M$ (post del contrato)
- Backward: Dado $\{pre_M\}$ S $\{post_M\}$ calculo $WP(S, post_M)$ y verifico que $\{pre_M\} \implies WP(S, post_M)$, es decir que la precondición del contrato sea más restrictiva que la WP

Calcular WP

- WP(`skip`, B) $=_{def}$ B
- WP(`x:=E`, B) $=_{def}$ B[x→E]
- WP(`s1;s2` , B) $=_{def}$ WP(s1, WP(s2, B))
- WP(`if(E){s1}else{s2}`, B) $=_{def}$
  E=> WP(s1,B) &&
  !E => WP(s2,B)

Los ciclos!

- WP_k(`while(E) {S}`, B) = $\vee_{i=0} H_i(D)$
  - H_0(...) = def(E) ∧ ¬E ∧ B,
  - H_1(...) $=_{def}$ def(E) ∧ E ∧ wp(S, ¬E ∧ B))
    = def(E) ∧ E ∧ wp(S, H_0 (B)).
  - ....
  - $H_{k+1}(Q) \equiv$ def(E) ∧ E ∧ wp(S, H_k (B)) para k ≥ 0.
- Calcularla de forma precisa puede ser imposible en general…

Ejemplo

- WP(skip, B) $=_{def}$ B
- WP(`x:=E`, B) $=_{def}$ B[x→E]
- WP(`s1;s2`, B) $=_{def}$ WP(s1, WP(s2, B))
- WP(`if(E){s1}else{s2}`, B) $=_{def}$
  E=> WP(s1,B) &&
  !E => WP(s2,B)

```
bool P(bool a, bool b)
requires true
ensures c==a || b
{
  if (a)
    c=true
  else
    c=b
}
```

WP(`if(a)`..., c==a||b) =

a=> WP(`c=true`, c==a||b) &&

!a => WP(`c=b`, c==a||b)

= (a => true==a||b) && (!a => b== a||b)

Conjetura lógica a probar:
preM => WP(P, c==a||b)
true =>(a=> true==a||b) && (!a => b==a||b) ✓

Solucion ciclos: loop unroling Es una solución unsafe porque puede dar una precondición más debil que la WP

Loop unrolling

DEPARTAMENTO DE COMPUTACION

```
public m(int n)
requires x>0 && n>0;
ensures x>=n
{
  for(int i=0; i< n; i++)
  {
     x=x+1;
  }
}
```

```
public m'(int n)
requires x>0 && n>0;
ensures x>=n
{
  if(0<n) {
    x=x+1;
    if(1<n) {
      x=x+1
      if (2<n) assume false; // aceptar programa
      else skip
    else skip
  }
  else skip
}
```

- La WP (m, x>=n) real es **x>=0**
- Con 2 unrolls, WP(m' , x>=n) = **(n<=2 => x>=0)**
  - Vale que x>0 && n>0 => (n<=2 => x>=0)
  - Pero unrolling produjo algo más débil…
    - Hint: Que pasa si la pre es n = 3 y x=-1?

por lo tanto se cumple la VC,1 y daria que el programa con el contrato es correcto

En este ejemplo $pre_M \implies WP'$ y $WP \implies WP'$ por ser más debil, pero pre_M no implica la WP. Dando erroneamente que el programa con la especificación es correcto.

Solucion ciclos: invariantes de ciclo No garantiza terminación

{ A }
Init;
{ I }
while (Cond) {
  { I & Cond}
  S;
  { I}
}
{ I & !Cond => B}
{B}

Probar
- {A} Init {Inv}
- {Cond && Inv} S {Inv}
- (!Cond && Inv) => B

Entonces
- {A} while (Cond) {S} {B}

- First consider *partial correctness*
  - The loop may not terminate, but if it does, the postcondition will hold
- {P} while B do S {Q}
  - Find an invariant Inv such that:
    - P ⇒ Inv
      - The invariant is initially true
    - { Inv && B } S {Inv}
      - Each execution of the loop preserves the invariant
    - (Inv && ¬B) ⇒ Q
      - The invariant and the loop exit condition imply the postcondition

Extensión de lenguaje

```
WP(assume E, B) == (E => B)
WP(assert E, B) == (E && B)
// para todo valor de x vale B
WP(havoc x, B) == \forall x. B
```

$$\frac{\{\phi \wedge B\}\ p\ \{\phi\}}{\{\phi\}\ \textbf{while}\ B\ \textbf{do}\ p\ \textbf{od}\ \{\phi \wedge \neg B\}}$$

```
While (I) B do S end ==
     assert I
     havoc S
     assume I
     if (B) then
          S
          assert I
          assume false
     endif
```

- assert I: Comprueba que el invariante se mantienen al principio
- havoc S: Olvidar la información de las variables afectadas, usar solo la información proporcionada por el invariante
- assert I (dentro del if): Chequear que el invariante se conserva después de ejecutar el cuerpo del bucle

Tratamiento de llamadas

[x/j] a x asignale j

```
void foo() {
…
j = -4
h = m(j);
…
}
```

```
//@ requires true;
//@ ensures \retvalue == |x |;

public void m(int x) {
  if(x<0) return –x;
   return x;
}
```

Genera en el método

```
public void m(int x) {
//@ assume true;
  if(x<0) return –x;
   return x;
//@ \retvalue == |x |;
}
```

Y en el llamador…

```
…
j = -4
//@ assert true[x/j];
h = m(j);
//@ assume \retvalue==|x|[x/j][retvalue/h];
…
```

2 Opciones:

- Inlining
- Usar el contrato

```
//@ requires P;
//@ ensures Q;
//@ modifies M // lo que se modifica
public void m() {
  …
}
```

Genera en el método

```
public void m(params) {
//@ assume P;
…
//@ assert Q;
}
```

Y en el llamador…

```
//@ assert P[params/args];
v = m(args); (havoc M)
//@ assume Q[params/args][result/v];
```

Invariante de representación

Es un predicado que deben cumplir todas las instancias de la clase. Al llamar a un método el invariante vale antes y despues (durante la ejecución puede romperse siempre y cuando se recomponga para el final).

```
class Racional
{
   private int numerador;
   private int denominador;
   //@ invariant mcd(numerador, denominador)==1 && denominador>0;

   //@ requires d>0;
   public Racional(int n, int d){
     int mcd = Math.gcd(n,d);
     numerador = n/mcd;
     denominador = d/mcd;
   }
}
```

- Los invariantes deben valer a la entrada y salida del método
  - A veces esto es muy restrictivo
- Helper methods:
  - Tienen que ser privados
  - No requieren que valga el invariante
  - Ni establecer el invariante

```
class Racional
{
  private int numerador;
  private int denominador;
  invariant mcd(numerador, denominador)== 1
  && denominador>0;

  public void suma(Racional r){
    numerador n = numerador*r.denominador
    + r.numerador * denominador;
    denominador =denominador*r.denominador;
     reconstuir();
  }
}
```

```
/* @helper */ void reconstruir()
{
 int mcd = Math.gcd(numerador,
          denominador);
 numerador = numerador/mcd;
 denominador = denominador/mcd;
}
```

- **Axioma 1.** $wp(\texttt{x := E}, Q) \equiv \mathrm{def}(E) \wedge_L Q^x_E$.
- **Axioma 2.** $wp(\texttt{skip}, Q) \equiv Q$.
- **Axioma 3.** $wp(\texttt{S1; S2}, Q) \equiv wp(\texttt{S1}, wp(\texttt{S2}, Q))$.
- **Axioma 4.** $wp(\texttt{if B then S1 else S2 endif}, Q) \equiv$

$$def(B) \wedge_L \Big((B \wedge wp(\texttt{S1}, Q)) \vee (\neg B \wedge wp(\texttt{S2}, Q)) \Big)$$

- **Observación:** $wp(\texttt{b[i]:=E}, Q) \equiv wp(\texttt{b:=setAt(b,i,E)}, Q)$

$wp(b[i] := E, Q)$

$$\begin{aligned}
&\equiv wp(b := setAt(b,i,E), Q)\\
&\equiv def(setAt(b,i,E)) \wedge_L Q^b_{setAt(b,i,E)}\\
&\equiv ((def(b) \wedge def(i)) \wedge_L 0 \leq i < |b|) \wedge \mathrm{def}(E)) \wedge_L Q^b_{setAt(b,i,E)}
\end{aligned}$$

Dados $0 \leq i, j < |b|$, recordemos que:

$$\boxed{setAt(b,i,E)[j] = \begin{cases} E & \text{si } i = j \\ b[j] & \text{si } i \neq j \end{cases}}$$