Program verification with Hoare Logic

Pre/Post Specifications

- **Pre-condição** - restrição aos valores dos parâmetros de input e do estado inicial do programa
- **Pós-condição** - restrição aos valores do output dos programas e ao estado final.

Hoare triple

- **{P} S {Q}** → Se uma expressão S é executada num estado de um programa que satisfaz as pre-condições P, então a execução de S termina e o estado final do programa satisfaz as pós condições Q. (No dafny as restrições são vistas no momento de compilação e não em run time)

Completing Hoare triples

- **{?} S {Q}** → Encontrar a *weakest pre-condition* para que o triple é true. Denotado por *wp(S, Q)*.
- **{P} S {?}** → Encontrar a *strongest pre-condition* para que o triple é true. Denotado por *sp(S, Q)*.
- **{P} ? {Q}** → Encontrar a mais simples e eficiente expressão para a qual o triplo é true.

Proving

Skip

{P} skip {Q} temos de provar que **P=>Q**

Exemplo: {x = 0} skip {x ≥ 0}

temos de provar que: x = 0 => x ≥ 0 (para todo x pertencente a Z), que é verdade, então o triple é verdadeiro.

Assignments

{P} x:=E {Q} temos de provar que **P=>Q[E/x]** (Q com x substituido por E)

Exemplo: {x = y} x := x+1 {x = y+1}

temos de provar que: (x = y) => (x+1 = y+1), que é verdade, então o triple é verdadeiro.

Conditionals

{P} if C then S else T {Q} temos de provar **{P⋀C} S {Q} e {P⋀¬C} T {Q}**

Exemplo: {true} if y < x then x := y else skip {x ≤ y}

(a) {true ⋀ y<x} x := y {x≤y} <=> {y<x} x:=y {x≤y} <=> (y<x ⋀ y≤y) <=> true

(b) {true ⋀ y≥x} skip {x≤y} <=> {y≥x} skip {x≤y} <=> (y≥x => x≤y) <=> true

Loops

Loops

{P} while C do S {Q}

(1) ⚠ Encontrar o **invariant I** do ciclo - Expressão booleana que se mantém antes e depois de todo o ciclo e de cada iteração.

(2) ⚠ Encontrar o **variante V** do ciclo - um inteiro que diminui em cada iteração e não é negativo (necessário provar a sua terminação)

(3) Provar:

- **P=>I** (O invariante inicialmente é válido)
- **I ∧ ¬C => Q** (Quando o ciclo termina, a pós condição é atingida)
- **I ∧ C => v≥0** (O variante não é negativo)
- **{I ∧ C ∧ v = V} S {I ∧ v<V}** (o invariante é preservado e o variante diminiu estritamente)

Exemplo: {n≥0} while n>0 do n:=n-1 {n=0}

(1) I = (n≥0) (baseado na pré-condição - obrigação)

(2) v=n

(3) Provar:

- P=>I, Garantido por (1)
- (I ∧ ¬C => Q) <=> (n ≥ 0 ∧ n ≤ 0 => n = 0) <=> true
- I ∧ C => v ≥ 0 <=> (n ≥ 0 ∧ n > 0 => n ≥ 0) <=> true
- {I ∧ C ∧ v=V} S {I ∧ v<V}
  <=> {n ≥ 0 ∧ n > 0 ∧ n=V} n:=n-1 {n ≥ 0 ∧ n<V}
  <=> (n>0 ∧ n=V => n-1 ≥ 0 ∧ n-1 < V)
  <=> (n > 0 ∧ n=V => n ≥ 1 ∧ n-1 < n)
  <=> true

Composition

{P} S;T {Q}

1. Encontrar a condição **R** que é mantida entre S e T (⚠ Hint - se escolhermos R = wp(T,Q) então 3 está provado; se escolhermos R = sp(P,S) então 2. está provado. Se T for um ciclo devemos usar o invariante (I))
2. provar **{P} S {R}**
3. provar **{R} T {Q}**

Exemplo: {true} x := 1; y := 0 {x = 1 ∧ y = 0}

(1) R = wp(y:=0, x=1 ∧ y=0) = (x=1 ∧ 0=0) = (x=1)

(2) {true} x:=1 {x=1} <=> (true => 1=1) <=> true

(3) Provado pela escolha do R

WP calculation rules (Dijkstra)

ID	Statement	Rule
R1'	skip	wp(skip, P) = P
R2'	assignment	wp(x := E, Q) = Q[E/x]
R3'	composition	wp(S;T, Q) = wp(S, wp(T, Q))

ID	Statement	Rule
R4'	conditional	wp(if C then S else T, Q) = C ∧ wp(S,Q) / ⌐C ∧ wp(T,Q)
R5'	loop	wp(while C do S, Q) = P0 / P1 / ..., with P0= ⌐C ∧ Q, Pk = C ∧ wp(S,Pk-1), k>0
R6'	assertion	wp(assert A, Q) = A ∧ Q