

# Algoritmos e Complexidade

## Introdução à Análise de Correção de Algoritmos

José Bernardo Barros  
Departamento de Informática  
Universidade do Minho

### 1 Introdução

Um programa pode ser definido como um mecanismo (ou máquina) de transformação de informação. Escrever um programa é, por isso, relacionar as entradas e saídas de tal máquina.

Por exemplo, para calcular o factorial de um número podemos escrever os seguintes programas:

```
fact 0 = 1
fact (n+1) = (n+1) * (fact n)
```

```
f = 1;
while (n>0) {
  f = f*n;
  n = n-1;}
```

Esta definição é suficientemente abrangente para poder incluir vários paradigmas de programação. É aliás uma forma de distinguir entre os dois grandes grupos de linguagens de programação:

- Nas linguagens de programação **declarativas** a ênfase é posta na explicitação da relação existente entre as saídas (*output*) e as entradas (*input*). A forma como tal transformação é feita não está explicitada no programa; é antes uma característica de cada uma das linguagens em causa.
- Nas linguagens de programação **imperativas** um programa descreve as transformações a que a informação de entrada é sujeita até ser transformada na informação de saída. Não é por isso geralmente fácil determinar a relação existente entre os estados iniciais e finais da informação.

Uma das desvantagens da programação imperativa é a fraca ligação que existe entre os programas (vistos como sequências de instruções) e as suas especificações (vistas como a relação que existe entre os *inputs* e os *outputs*).

Daí que sejam necessários mecanismos exteriores à linguagem de programação nos quais seja possível expressar essa ligação. Desta forma consegue-se avaliar a adequação de um programa face a uma especificação.

Nestas notas apresenta-se, de uma forma muito introdutória, um desses mecanismos – *triplos de Hoare*. Veremos como estes podem ser usados para nos pronunciarmos sobre a correcção de um algoritmo face a uma dada especificação. Veremos ainda, se bem que de uma forma muito breve, como tal formalismo pode ser usado para guiar a derivação de um algoritmo a partir de uma dada especificação.

A grande fonte de inspiração deste documento é a parte inicial de um curso leccionado por Mike Gordon [?] na Universidade de Cambridge e disponível a partir da página do autor (<http://www.cl.cam.ac.uk/~mjc/>)

## 2 Programas

Uma das características mais importantes das linguagens imperativas é a existência de **estado**. Uma visão simplista do estado de um programa é como o conjunto de variáveis (memória) a que o programa pode aceder.

Em cada estado, a cada variável está associado um valor. Podemos por isso pensar no estado como uma função que a cada variável associa o seu valor. Se  $s$  for um estado e  $v$  for uma das suas variáveis, é costume representar-se por  $\llbracket v \rrbracket_s$  o valor de  $v$  no estado  $s$ . Esta função, que associa a cada variável o seu valor num dado estado, pode ser generalizada para fazer corresponder a cada expressão o seu valor num dado estado. Por exemplo, se  $\llbracket x \rrbracket_s = 3$  e  $\llbracket y \rrbracket_s = 4$  então

- $\llbracket x + (y * x) \rrbracket_s = \llbracket x \rrbracket_s + (\llbracket y \rrbracket_s * \llbracket x \rrbracket_s) = 15$
- $\llbracket x+1 = y \rrbracket_s = \text{True}$

A linguagem de programação que vamos apresentar é muito simples. Tem no entanto os ingredientes necessários à análise de um conjunto razoável de problemas.

Tomando como base um conjunto  $V$  de variáveis de estado, e as operações usuais sobre os valores dessas variáveis, a sintaxe de tal linguagem de programação pode ser descrita por:

```
<programa> ::= V = <exp>
               <programa> ; <programa>
               if <cond> <programa>
               if <cond> <programa> else <programa>
               while <cond> { <programa> }
```

## 3 Especificações

A correcção de um programa está estritamente relacionada com a sua especificação. Por outras palavras, não se pode afirmar que um programa está ou não correcto: um programa que ordene um vector de inteiros por ordem crescente está correcto se for essa a sua especificação; o mesmo programa está incorrecto se a especificação for *inicializar o vector com zeros*.

Para especificar um programa vamos usar dois predicados que estabelecem as propriedades dos estados antes e depois da execução do programa:

- a **pré-condição** que estabelece as condições em que o programa deve funcionar;
- a **pós-condição** que estabelece aquilo que deve acontecer após a execução do programa.

Comecemos por analisar alguns exemplos de especificações de problemas simples e bem conhecidos.

**Exemplo 1 (swap)** Para especificarmos o programa que troca os valores das variáveis  $x$  e  $y$  podemos *tentar* a seguinte especificação.

<b>Pré-condição:</b> $True$
<b>Pós-condição:</b> $x = y \wedge y = x$

Duas notas sobre esta especificação:

- a pré-condição  $True$  significa que não há quaisquer restrições ao funcionamento do programa;
- a pós-condição apresentada é uma forma rebuscada de dizer que no final os valores das variáveis  $x$  e  $y$  são iguais. O que não era de todo o que tínhamos em mente.

Este exemplo mostra que por vezes a especificação de um problema precisa de relacionar valores de variáveis antes e depois da execução do programa. Há muitas formas de lidar com esta requisito. Aquela que vamos adoptar é a de, sempre que necessário, fixar os valores iniciais das variáveis. Assim, a especificação do programa que troca os valores das variáveis  $x$  e  $y$  é:

<b>Pré-condição:</b> $x = x_0 \wedge y = y_0$
<b>Pós-condição:</b> $x = y_0 \wedge y = x_0$

O uso de um predicado aparentemente mais restritivo serve apenas o propósito de fixar os valores iniciais das variáveis  $x$  e  $y$ .  $x_0$  e  $y_0$  são frequentemente referidas como variáveis lógicas uma vez que não correspondem a nenhuma variável do programa.

**Exemplo 2 (produto)** Para especificarmos um programa que calcula o produto de dois inteiros, devemos não só dizer quais os inteiros a multiplicar mas onde esse resultado será colocado. Teremos por exemplo

<b>Pré-condição:</b> $x = x_0 \wedge y = y_0 \geq 0$
<b>Pós-condição:</b> $m = x_0 * y_0$

que pode ser lido como *calcular o produto dos valores iniciais de  $x$  e  $y$  colocando o resultado na variável  $m$* . Note-se que esta especificação é omissa quanto ao que acontece com as variáveis  $x$  e  $y$ . Podemos por isso ter programas correctos em relação a esta especificação que modificam ou não o valor de alguma destas variáveis.

**Exemplo 3 (mod)** A especificação seguinte estabelece os requisitos de um programa que coloca em  $m$  o resto da divisão inteira entre os valores iniciais das variáveis  $x$  e  $y$ .

**Pré-condição:**  $x = x_0 > 0 \wedge y = y_0 \geq 0$

**Pós-condição:**  $0 \leq m < y_0 \wedge \exists_{d \geq 0} d * y_0 + m = x_0$

**Exemplo 4 (div)** A especificação seguinte estabelece os requisitos de um programa que coloca em  $d$  o resultado da divisão inteira entre os valores iniciais das variáveis  $x$  e  $y$ .

**Pré-condição:**  $x = x_0 > 0 \wedge y = y_0 \geq 0$

**Pós-condição:**  $d \geq 0 \wedge \exists_{0 \leq m < y_0} d * y_0 + m = x_0$

**Exemplo 5 (divmod)** A especificação seguinte estabelece os requisitos de um programa que coloca em  $d$  o resultado da divisão inteira entre os valores iniciais das variáveis  $x$  e  $y$  e em  $m$  o resto dessa divisão.

**Pré-condição:**  $x = x_0 > 0 \wedge y = y_0 \geq 0$

**Pós-condição:**  $0 \leq m < y_0 \wedge d \geq 0 \wedge d * y_0 + m = x_0$

**Exemplo 6 (procura)** Consideremos o problema de procurar um dado valor ( $x$ ) num vector ordenado ( $v[]$  da posição  $a$  a  $b$ ). A especificação deste problema pode ser feita com os seguintes predicados:

**Pré-condição:**  $(\forall_{a \leq i \leq b} . v[i] = v_i) \wedge (\forall_{a \leq i < b} . v_i \leq v_{i+1})$

**Pós-condição:**  $(\forall_{a \leq i \leq b} . v[i] = v_i) \wedge ((\exists_{a \leq i \leq b} . v_i = x) \Rightarrow v[p] = x)$

Vejamos com mais detalhe cada uma das conjunções acima.

Na pré-condição, o primeiro termo serve para fixarmos os valores iniciais do vector. Este mesmo termo aparece na pós-condição, obrigando por isso que os valores do vector não sejam alterados.

O segundo termo da conjunção afirma que o vector está ordenado. Uma formulação alternativa seria

$$\forall_{a \leq i, j \leq b} . i \leq j \Rightarrow v_i \leq v_j$$

Finalmente o segundo termo da pós-condição afirma que, se existir um elemento do vector igual a  $x$ , então o valor da componente índice  $p$  tem esse valor  $x$ .

Note-se que não se especifica qual será o valor de  $p$  no caso de o valor que procuramos não ocorrer no vector.

**Exercício 1** Descreva por palavras as seguintes especificações:

1. **Pré-condição:**  $x = x_0 \geq 0 \wedge e = e_0 > 0$

**Pós-condição:**  $|r * r - x_0| < e_0$

2. **Pré-condição:**  $\forall_{0 \leq i < N} A[i] = a_i$

**Pós-condição:**  $\forall_{0 \leq i < N} (A[i] = a_i \wedge A[p] \leq a_i)$

**Exercício 2** Escreva especificações (pré e pós condições) para os seguintes problemas:

1. Um programa que coloca na variável  $r$  o mínimo múltiplo comum das variáveis  $X$  e  $Y$ .
2. Um programa que recebe dois arrays  $A$  e  $B$  como parâmetros, e verifica se eles têm um elemento em comum.
3. Um programa que recebe dois arrays  $A$  e  $B$  como parâmetros, e calcula o comprimento do prefixo mais longo que os dois têm em comum.

## 4 Correção Parcial

Dados

- Um programa  $S$
- Dois predicados  $P$  e  $Q$  sobre as variáveis do programa  $S$

escrevemos

$$\{ P \} S \{ Q \}$$

e lê-se o programa  $S$  está (parcialmente) correcto face à especificação  $(P, Q)$ , com o seguinte significado:

Se, a partir de todos os estados em que  $P$  é válido, executarmos o programa  $S$ , depois dessa execução terminar, atingimos estados em que  $Q$  é válido.

Para melhor compreender este conceito de validade, vejamos um caso em que essa validade não é verificada.

**Exemplo 7** Atentemos no seguinte triplo:

$$\{ x > 0 \} x = x + y \{ x > 1 \}$$

Para mostrarmos a validade deste triplo teremos que enumerar todos os estados em que a pré-condição  $x > 0$  se verifica, e assegurarmo-nos que depois de executar o programa  $x = x + y$  a pós-condição (calculada no estado resultante) é válida.

Para mostrarmos que o triplo não é válido temos que encontrar pelo menos um destes estados iniciais (contra-exemplo) em que tal não se verifique.

Considere-se então o estado  $A$  em que  $\llbracket x \rrbracket_A = 3$  e  $\llbracket y \rrbracket_A = -5$ .

Note-se que neste estado a pré-condição é válida:

$$\llbracket x > 0 \rrbracket_A \Leftrightarrow (3 > 0) \Leftrightarrow True$$

Partindo desse estado, atingimos um estado  $B$  em que  $\llbracket x \rrbracket_B = -2$  e  $\llbracket y \rrbracket_B = -5$ . Ora neste estado a pós-condição não é válida:

$$\llbracket x > 1 \rrbracket_B \Leftrightarrow (-2 > 1) \Leftrightarrow False$$

Este exemplo evidencia que a forma de provar que um dado triplo não é válido consiste em descobrir um contra-exemplo. Para determinar que um destes triplos é válido, teríamos que enumerar todos os estados (que validam a pré-condição) e executar o programa a partir deles. Ora esta tarefa é em geral inviável e por isso teremos que estabelecer um conjunto de regras de prova que nos permitam atingir tal objectivo

Para cada um dos construtores de programas vistos na secção 2 vamos apresentar regras de prova da correção de programas que envolvam essas construções.

**Exercício 3** Pronuncie-se sobre a validade dos seguintes triplos de Hoare:

1.  $\{ i > j \} j = i + 1; i = j + 1 \{ i > j \}$

2.  $\{i \neq j\} \text{ if } (i > j) \text{ then } m = i - j \text{ else } m = j - i \{m > 0\}$
3.  $\{a > b\} m = 1; n = a - b \{m * n > 0\}$
4.  $\{s = 2^i\} i = i + 1; s = s * 2 \{s = 2^i\}$
5.  $\{\text{True}\} \text{ if } (i < j) \text{ then } min = i \text{ else } min = j \{min \leq i \wedge min \leq j\}$
6.  $\{i > 0 \wedge j > 0\} \text{ if } (i < j) \text{ then } min = i \text{ else } min = j \{min > 0\}$

#### 4.1 Restrição das Especificações

Convém notar a semelhança que existe entre a correcção parcial e a implicação de predicados.

- Quando, para dois predicados  $P$  e  $Q$  dizemos que  $P \Rightarrow Q$  é válido queremos dizer que se  $P$  é válido  $Q$  também é. Dizemos ainda que  $P$  é mais forte (ou mais restritivo) do que  $Q$ .
- Por seu lado, quando dizemos que  $\{P\} S \{Q\}$  é válido queremos dizer que se  $P$  for válido num dado estado,  $Q$  também o será *depois da execução de  $S$* .

Daqui, e da transitividade da implicação, podemos desde já enunciar duas regras de correcção, que dizem respeito à restrição de uma especificação.

**Fortalecimento da pré-condição** Se um programa  $S$  funciona em determinadas condições iniciais  $P$ , ele continuará a funcionar em condições mais restritivas.

$$\frac{R \Rightarrow P \quad \{P\} S \{Q\}}{\{R\} S \{Q\}} \quad (\text{Fort})$$

**Enfraquecimento da pós-condição** Se um programa  $S$  garante que alguma propriedade  $Q$  é válida, garantirá que qualquer condição menos restritiva também é válida.

$$\frac{\{P\} S \{Q\} \quad Q \Rightarrow R}{\{P\} S \{R\}} \quad (\text{Enfrac})$$

Estas duas regras podem ser resumidas numa só que traduz a restrição de especificações.

$$\frac{P \Rightarrow P' \quad \{P'\} S \{Q'\} \quad Q' \Rightarrow Q}{\{P\} S \{Q\}} \quad (\text{Consequência})$$

#### 4.2 Atribuição

A operação fundamental de qualquer linguagem de programação imperativa é a atribuição de um valor a uma variável.

Antes de apresentar a regra de correcção da atribuição convém relembrar o significado de tal comando. O efeito de uma atribuição  $x = E$  pode ser descrito por:

1. Começa-se por calcular o valor da expressão E no estado inicial.
2. O estado é então alterado mudando o valor da variável x para esse valor então calculado.

Esta descrição evidencia que o valor da expressão E deva ser calculado no estado inicial, motivando a seguinte regra de correcção.

#### Atribuição–1

$$\frac{}{\{ P[x \setminus E] \} \ x = E \ \{ P \}} \quad (\text{Atrib1})$$

Quando escrevemos  $P[x \setminus E]$  significamos *substituir todas as ocorrências (livres) da variável x pela expressão E*. Assim por exemplo,

- $(x + y)[x \setminus x - y]$  é a expressão  $(x - y) + y$
- 

$$(x + \sum_{y=0}^n y^2)[y \setminus y + 1]$$

é a expressão  $x + \sum_{y=0}^n y^2$  (uma vez que a variável y não está livre).

É de realçar que esta regra nos permite determinar qual é a restrição menos forte que devemos fazer para obter um dado resultado após uma atribuição.

Conjugando esta regra com a do fortalecimento da pré-condição permite-nos escrever uma regra de aplicação mais usual.

#### Atribuição–2

$$\frac{P \Rightarrow (Q[x \setminus E])}{\{ P \} \ x = E \ \{ Q \}} \quad (\text{Atrib2})$$

### 4.3 Sequenciação

Uma outra construção fundamental de programas é a de sequenciação: executar um programa após outro.

Para motivar a regra de correcção desta construção, vejamos a diferença que existe entre os seguintes comandos em *python*. Assumamos que partimos de um estado em que o valor das variáveis a e b são 10 e 6, respectivamente.

- O comando `a = a + b; b = a - b` leva-nos para um estado em que as variáveis a e b têm os valores 16 e 10.
- O comando `a, b = a + b, a - b` leva-nos para um estado em que as variáveis a e b têm os valores 16 e 4.

Isto porque enquanto que no segundo comando, os valores a atribuir são calculados num mesmo estado inicial (daí se chamar atribuição simultânea), no primeiro comando, o valor da segunda expressão é calculado num estado intermédio (correspondendo ao estado final do primeiro comando).

A regra de correcção associada à sequenciação de programas deve espelhar que o segundo programa deve ter como entrada (i.e., pré-condição) a saída (i.e., pós-condição) do primeiro.

### Sequência

$$\frac{\{P\} S_1 \{R\} \quad \{R\} S_2 \{Q\}}{\{P\} S_1 ; S_2 \{Q\}} \quad (;$$

**Exemplo 8** Vamos provar que o seguinte algoritmo troca os valores das variáveis  $x$  e  $y$ .

```
x = x + y ;
y = x - y ;
x = x - y
```

A especificação deste problema foi apresentada no Exemplo 1 da página 3.

Usando a correcção da sequenciação, temos de encontrar predicados  $R_1$  e  $R_2$  tais que:

$$\begin{aligned} & \{x = x_0 \wedge y = y_0\} \\ & x = x + y \\ & \{R_2\} \\ & y = x - y \\ & \{R_1\} \\ & x = x - y \\ & \{x = y_0 \wedge y = x_0\} \end{aligned}$$

O cálculo dos predicados  $R_1$  e  $R_2$  é feito, por essa ordem usando a primeira regra apresentada para a atribuição. Assim teremos:

- $R_1 = (x = y_0 \wedge y = x_0)[x \setminus x - y]$   
 $= x - y = y_0 \wedge y = x_0$
- $R_2 = R_1[y \setminus x - y]$   
 $= (x - y = y_0 \wedge y = x_0)[y \setminus x - y]$   
 $= x - (x - y) = y_0 \wedge x - y = x_0$   
 $= y = y_0 \wedge x - y = x_0$

Para completarmos a prova vamos usar a segunda das regras apresentadas para a atribuição. Temos então de provar que:

$$(x = x_0 \wedge y = y_0) \Rightarrow R_2[x \setminus x + y]$$

Começemos por simplificar o consequente desta implicação.

$$\begin{aligned} R_2[x \setminus x + y] &= (y = y_0 \wedge x - y = x_0)[x \setminus x + y] \\ &= y = y_0 \wedge (x + y) - y = x_0 \\ &= y = y_0 \wedge x = x_0 \end{aligned}$$

Que não é mais do que o antecedente, e por isso a implicação é válida.



**Exemplo 9** Uma forma mais habitual de resolver o mesmo problema (da troca dos valores de duas variáveis) passa por usar uma terceira para armazenar temporariamente o valor de uma delas.

```
z = x ;
x = y ;
y = z
```

Usando a correcção da sequenciação, temos de encontrar predicados  $R_1$  e  $R_2$  tais que:

$$\begin{aligned} & \{ x = x_0 \wedge y = y_0 \} \\ & z = x \\ & \{ R_2 \} \\ & x = y \\ & \{ R_1 \} \\ & y = z \\ & \{ x = y_0 \wedge y = x_0 \} \end{aligned}$$

Donde vem:

$$\begin{aligned} \bullet \quad R_1 &= (x = y_0 \wedge y = x_0)[y \setminus z] \\ &= x = y_0 \wedge z = x_0 \\ \bullet \quad R_2 &= R_1[x \setminus y] \\ &= (x = y_0 \wedge z = x_0)[x \setminus y] \\ &= y = y_0 \wedge z = x_0 \end{aligned}$$

Para completarmos a prova vamos usar a segunda das regras apresentadas para a atribuição. Temos então de provar que:

$$(x = x_0 \wedge y = y_0) \Rightarrow R_2[z \setminus x]$$

Comecemos por simplificar o conseqüente desta implicação.

$$\begin{aligned} R_2[z \setminus x] &= (y = y_0 \wedge z = x_0)[z \setminus x] \\ &= y = y_0 \wedge x = x_0 \end{aligned}$$

Que não é mais do que o antecedente, e por isso a implicação é válida.

Como podemos ver pelos exemplos apresentados, a aplicação da regra da sequenciação, quando os comandos envolvidos são atribuições, traduz-se por aplicar sucessivamente a regra da atribuição pela ordem inversa à que aparecem na sequência. Daí que, na prática, seja mais útil a seguinte regra composta.

$$\frac{\{ P \} S_1; \dots; S_n \{ Q[x \setminus E] \}}{\{ R \} S_1; \dots; S_n; x = E \{ Q \}} \quad (\text{SeqAtr})$$

#### 4.4 Condicionais

A correcção de programas que envolvam condicionais é dada pela seguinte regra.

**Condicional**

$$\frac{\{ P \wedge c \} S_1 \{ Q \} \quad \{ P \wedge \neg c \} S_2 \{ Q \}}{\{ P \} \text{ if } c \text{ S}_1 \text{ else } S_2 \{ Q \}} \quad (\text{ifThenElse})$$

Que traduz o significado intuitivo da construção **if**  $c$   $S_1$  **else**  $S_2$ : partindo de  $P$ , a pós-condição  $Q$  pode ser atingida executando um de dois comandos:

- $S_1$  no caso da condição ser verdadeira
- $S_2$  no caso da condição ser falsa

**Exemplo 10** Vamos provar que o seguinte algoritmo coloca em  $M$  o máximo entre os valores das variáveis  $x$  e  $y$ .

```
if (x > y)
  M = x ;
else
  M = y ;
```

A especificação informal feita acima pode ser feita usando os seguintes predicados.

**Pré-condição:**  $x = x_0 \wedge y = y_0$

**Pós-condição:**  $M = \max(x_0, y_0)$

Usando a correcção dos condicionais, podemos anotar o algoritmo acima com os seguintes predicados.

```
{ x = x0 ∧ y = y0 }
if (x > y)
  1 [ { x > y ∧ x = x0 ∧ y = y0 }
    M = x
    { M = max(x0, y0) }
else
  2 [ { x ≤ y ∧ x = x0 ∧ y = y0 }
    M = y
    { M = max(x0, y0) }
```

Vamos então usar a regra da atribuição para concluir a prova. Para isso temos de mostrar a validade das seguintes implicações

1.  $(x > y \wedge x = x_0 \wedge y = y_0) \Rightarrow (M = \max(x_0, y_0))[M \setminus x]$   
 $\Rightarrow x = \max(x_0, y_0)$
2.  $(x \leq y \wedge x = x_0 \wedge y = y_0) \Rightarrow (M = \max(x_0, y_0))[M \setminus y]$   
 $\Rightarrow y = \max(x_0, y_0)$

Que são consequência da definição do máximo entre dois números.

Em muitas linguagens de programação existe ainda a possibilidade de definir condicionais só com uma alternativa. A regra associada a esta construção pode ser derivada da anterior se notarmos que em caso de falha não é executado qualquer comando. Teremos então:

### Condicional-2

$$\frac{\{P \wedge c\} S \{Q\} \quad (P \wedge \neg c) \Rightarrow Q}{\{P\} \text{ if } c S \{Q\}} \quad (\text{ifThen})$$

É de realçar que esta regra traduz o comportamento esperado do programa em causa:

1. se a condição é verdadeira o predicado  $Q$  só é atingido após a execução de  $S$
2. Quando a condição é falsa, o predicado  $Q$  é uma consequência imediata da pré-condição  $P$ .

**Exercício 4** Prove cada um dos seguintes triplos de Hoare.

1.  $\{i > j\} j = i + 1; i = j + 1 \{i > j\}$
2.  $\{i \neq j\} \text{ if } (i > j) \text{ then } m = i - j \text{ else } m = j - i \{m > 0\}$
3.  $\{a > b\} m = 1; n = a - b \{m * n > 0\}$
4.  $\{s = 2^i\} i = i + 1; s = s * 2 \{s = 2^i\}$
5.  $\{\text{True}\} \text{ if } (i < j) \text{ then } min = i \text{ else } min = j \{min \leq i \wedge min \leq j\}$
6.  $\{i > 0 \wedge j > 0\} \text{ if } (i < j) \text{ then } min = i \text{ else } min = j \{min > 0\}$

## 4.5 Ciclos

Por uma questão de simplicidade vamos usar apenas uma forma de ciclos, correspondente ao que em C se codifica com um **while**.

Para provarmos a correcção (parcial) de um programa da forma

**while**  $b$   $S$

vamos precisar de encontrar um predicado, denominado **invariante do ciclo** que traduz o processo usado na obtenção do resultado. Para isso teremos de provar que é verdadeiro antes de cada iteração do ciclo e que no final do ciclo (i.e., quando a condição do ciclo é falsa) nos garante que a pós-condição é alcançada.

A regra de correcção fundamental para os ciclos é:

**Ciclo-1**

$$\frac{\{I \wedge c\} S \{I\}}{\{I\} \text{ while } c S \{I \wedge \neg c\}} \quad (\text{while-1})$$

Podemos ainda usar as regras de restrição das especificações para derivar a seguinte regra de correcção de um ciclo.

**Ciclo-3**

$$\frac{P \Rightarrow I \quad \{I \wedge c\} S \{I\} \quad (I \wedge \neg c) \Rightarrow Q}{\{P\} \text{ while } c S \{Q\}} \quad (\text{while-3})$$

Vejamos então quais as premissas a provar quando queremos mostrar a validade de um ciclo:

1.  $\mathbf{P} \Rightarrow \mathbf{I}$ : Antes da execução do ciclo, o invariante é verdadeiro.
2.  $\{\mathbf{I} \wedge \mathbf{c}\} \mathbf{S} \{\mathbf{I}\}$ : Assumindo que o invariante é válido antes de uma iteração do ciclo, ele continua válido depois dessa iteração.
3.  $(\mathbf{I} \wedge \neg \mathbf{c}) \Rightarrow \mathbf{Q}$ : Quando o ciclo termina a pós-condição é estabelecida.

**Exemplo 11** Consideremos o seguinte programa que multiplica dois números inteiros por somas sucessivas:

```

1      m = 0; d = y;
2      while (d>0) {
3          m = m + x; d = d-1;
4      }
```

Podemos, *à posteriori*, tentar caracterizar este programa pela seguinte especificação:

**Pré-condição:**  $x = x_0 \wedge y = y_0 \geq 0$   
**Pós-condição:**  $m = x_0 * y_0$

Para tentarmos descobrir o invariante deste ciclo, vamos *experimental* o programa acima para um valor inicial do estado das suas variáveis (por exemplo, para  $x = x_0 = 11$  e  $y = y_0 = 6$ ).

Linha	x	y	d	m
1	11	6	?	?
2	11	6	6	0
3	11	6	6	0
2	11	6	5	11
3	11	6	5	11
2	11	6	4	22
3	11	6	4	22
2	11	6	3	33
3	11	6	3	33
2	11	6	3	33
...	...	...	...	...
2	11	6	1	55
3	11	6	1	55
2	11	6	0	66
4	11	6	0	66

A análise deste comportamento (particularmente o do estado antes de executar cada instância da linha 2) evidencia algumas propriedades que nos podem ajudar a tentar encontrar o variante e invariante necessários:

- Os valores de  $x$  e de  $y$  permanecem inalterados.
- O valor de  $m$  cresce proporcionalmente ao decréscimo de  $d$ .

Ajudados por estas observações, podemos formular o seguinte

$$I \doteq (x = x_0) \wedge (y = y_0) \wedge x_0 * d + m = x_0 * y_0$$

O predicado  $I$  acima não é suficiente para provar a correcção; mas podemos usá-lo como primeira aproximação.

Usando as regras apresentadas, aquilo que temos de mostrar é:

1.  $(x = x_0 \wedge y = y_0 \geq 0) \Rightarrow (I[m \setminus 0, d \setminus y])$
2.  $(I \wedge d > 0) \Rightarrow (I[m \setminus m + x, d \setminus d - 1])$
3.  $(I \wedge \neg(d > 0)) \Rightarrow (m = x_0 * y_0)$

Ao tentarmos mostrar a validade destas implicações apercebemo-nos que precisamos ainda de acrescentar ao invariante a propriedade  $d \geq 0$ , pois só assim garantiremos que no final do ciclo (i.e., quando a condição do ciclo for falsa) o valor de  $d$  é nulo, estabelecendo então a pós-condição em causa.

## 5 Anotações

Ao longo dos exemplos acima fomos nos apercebendo da necessidade de intercalar predicados (que explicitam algumas propriedades) no código do programa. Estes predicados ou **anotações** são usados desde os primórdios das Ciências da Computação (c.f. Cliff Jones [?]) e podem de alguma forma sintetizar o primeiro passo de um método de verificação formal da correcção de um programa.

Vamos adoptar a seguinte disciplina de anotação. Para além das anotações correspondentes à pré e pós condições (no início e fim do programa),

- usaremos uma anotação antes de qualquer comando que não seja uma atribuição;
- usaremos uma anotação imediatamente a seguir à condição de um ciclo (que traduz o invariante desse ciclo).

Por exemplo, no programa do Exemplo 11, o programa seria anotado nos seguintes pontos.

```

{ anotação 1 }
m = 0; d = y;
{ anotação 2 }
while (d>0)
{ anotação 3 }
    {
        m = m + x; d = d-1
    }
{ anotação 4 }
```

Como veremos à frente, a partir destes programas anotados, é possível gerar (automaticamente) as condições a verificar para provar a correcção do programa face à sua especificação.

### 5.1 Cálculo das condições de verificação

Vejamos então como é que a partir de um programa anotado se podem gerar tais condições. Para isso, vamos enumerar as várias construções da nossa linguagem e mostrar quais as condições associadas.

### 5.1.1 Atribuição

Ao programa anotado (composto por uma única instrução)

$$\{ A_1 \} \quad x=E \quad \{ A_2 \}$$

corresponde a única condição de verificação

1.  $A_1 \Rightarrow (A_2[x \setminus E])$

### 5.1.2 Sequência

Dado um programa anotado que é uma sequência de instruções, dois casos são possíveis:

- a última instrução é uma atribuição (e nesse caso não é precedida por nenhuma anotação). Então, as condições de verificação associadas ao programa anotado

$$\{ A_1 \} \quad S_1 \quad \cdots \quad S_n; \quad x=E \quad \{ A_2 \}$$

são as condições de verificação associadas ao programa anotado

$$\{ A_1 \} \quad S_1 \quad \cdots \quad S_n; \quad \{ A_2[x \setminus E] \}$$

- a última instrução não é uma atribuição (e por isso mesmo é precedida por uma anotação). Então, as condições de verificação associadas ao programa anotado

$$\{ A_1 \} \quad S_1 \quad \cdots \quad S_n; \quad \{ A_2 \} \quad S_{n+1} \quad \{ A_3 \}$$

são

1. as condições de verificação associadas ao programa anotado

$$\{ A_1 \} \quad S_1 \quad \cdots \quad S_n \quad \{ A_2 \}$$

2. as condições de verificação associadas ao programa anotado

$$\{ A_2 \} \quad S_{n+1} \quad \{ A_3 \}$$

### 5.1.3 Condicionais

- As condições de verificação associadas ao programa anotado

$$\{ A_1 \} \quad \text{if } c \quad S_1 \quad \text{else } S_2 \quad \{ A_2 \}$$

são

1. as condições de verificação associadas ao programa anotado

$$\{ A_1 \wedge c \} \quad S_1 \quad \{ A_2 \}$$

2. as condições de verificação associadas ao programa anotado

$$\{ A_1 \wedge \neg c \} \quad S_2 \quad \{ A_2 \}$$

- As condições de verificação associadas ao programa anotado

$$\{A_1\} \text{ if } c \text{ } S \{A_2\}$$

são

1.  $(A_1 \wedge \neg c) \Rightarrow A_2$
2. as condições de verificação associadas ao programa anotado

$$\{A_1 \wedge c\} \text{ } S_1 \{A_2\}$$

#### 5.1.4 Ciclos

As condições de verificação associadas ao programa anotado

$$\begin{array}{l} \{A_1\} \\ \text{while } c \\ \{A_2\} \\ S \\ \{A_3\} \end{array}$$

são

1.  $A_1 \Rightarrow A_2$
2.  $(A_2 \wedge \neg c) \Rightarrow A_3$
3. as condições de verificação associadas ao programa anotado

$$\{A_2 \wedge c\} \text{ } S \{A_2\}$$

## 5.2 Cálculo das condições de verificação (versão 2)

As regras de boa anotação apresentadas atrás significam que o nosso objecto de análise deixa de ser um triplo de Hoare e passa a ser um objecto cuja sintaxe poderá ser descrita da seguinte forma:

- Um predicado inicial (pré-condição)  $A_0$
- Uma sequência de  $N$  **blocos**, separados por predicados  $A_i$  com  $1 \leq i < N$
- Um predicado final (pós condição)  $A_N$

Ou seja, é da forma

$$P \doteq \{A_0\} \text{ } B_1 \{A_1\} \text{ } B_2 \{A_2\} \cdots B_n \{A_n\}$$

As condições de verificação deste programa anotado é a união das condições de verificação dos seus constiuintes, i.e.,

$$VC(P) = \bigcup_{i=1}^N VC(\{A_{i-1}\} \text{ } B_i \{A_i\})$$

Cada um dos blocos referidos atrás é uma das seguintes alternativas:

- Um bloco constituído exclusivamente por atribuições
- Um bloco constituído por um **if then else** seguido de uma sequência de atribuições
- Um bloco constituído por um **if then** seguido de uma sequência de atribuições
- Um bloco constituído por um **while** seguido de uma sequência de atribuições

Vejamos então como calcular as condições de verificação para cada um destes tipos de blocos.

- Ao bloco composto exclusivamente por atribuições

$$\{ P \} \quad x1=E1; \quad x2=E2; \quad \dots \quad xn=En \quad \{ Q \}$$

corresponde a única condição de verificação

$$1. \quad P \Rightarrow (Q[xn \setminus En] \dots [x2 \setminus E2][x1 \setminus E1])$$

- Ao bloco constituído por um **if then else** seguido de uma sequência de atribuições

$$\{ P \} \quad \text{if } c \quad S_1 \quad \text{else} \quad S_2; \quad x1=E1; \quad x2=E2; \quad \dots \quad xn=En \quad \{ Q \}$$

corresponde a união das condições de verificação associadas aos programas anotados

1.  $\{ P \wedge c \} \quad S_1 \quad \{ Q[xn \setminus En] \dots [x2 \setminus E2][x1 \setminus E1] \}$
2.  $\{ P \wedge \neg c \} \quad S_2 \quad \{ Q[xn \setminus En] \dots [x2 \setminus E2][x1 \setminus E1] \}$

- Ao bloco constituído por um **if then** seguido de uma sequência de atribuições

$$\{ P \} \quad \text{if } c \quad S; \quad x1=E1; \quad x2=E2; \quad \dots \quad xn=En \quad \{ Q \}$$

corresponde:

1. a condição  $P \Rightarrow (Q[xn \setminus En] \dots [x2 \setminus E2][x1 \setminus E1])$
2. bem como as condições de verificação associadas ao programa anotado

$$\{ P \wedge c \} \quad S \quad \{ Q[xn \setminus En] \dots [x2 \setminus E2][x1 \setminus E1] \}$$

- Ao bloco constituído por um **while** seguido de uma sequência de atribuições

$$\begin{array}{l} \{ P \} \\ \quad \text{while } c \\ \quad \{ I \} \\ \quad \quad S; \\ \quad x1=E1; \quad x2=E2; \quad \dots \quad xn=En \{ Q \} \end{array}$$

correspondem:

1. a condição  $P \Rightarrow I$
2. a condição  $(I \wedge \neg c) \Rightarrow (Q[xn \setminus En] \dots [x2 \setminus E2][x1 \setminus E1])$
3. bem como as condições de verificação associadas ao programa anotado

$$\{ I \wedge c \} \quad S \quad \{ I \}$$



### 5.3 Exponenciação inteira

Para calcular a potência positiva de um número podemos usar um algoritmo semelhante ao que foi apresentado para multiplicar dois números naturais por somas sucessivas. Para cumprirmos os requisitos da especificação

**Pré-condição:**  $a = a_0 \wedge b = b_0 > 0$

**Pós-condição:**  $p = a_0^{b_0}$

vamos usar o seguinte programa.

```
p=1;
while (b>0) {
  p = p * a;
  b = b - 1}
```

Para isso vamos usar como invariante  $I$  o seguinte predicado

$$I \doteq p = a_0^{b_0-b} \wedge a = a_0 \wedge b \geq 0$$

Com estes ingredientes podemos então escrever o seguinte programa anotado.

```
{ a = a_0 ∧ b = b_0 > 0 }
p=1;
{ a = a_0 ∧ b = b_0 > 0 ∧ p = 1 }
while (b>0)
  { p = a_0^{b_0-b} ∧ a = a_0 ∧ b ≥ 0 }
  p = p * a;
  b = b - 1
{ p = a_0^{b_0} }
```

A partir daqui podemos gerar as seguintes condições de verificação.

1.  $(a = a_0 \wedge b = b_0 > 0) \Rightarrow (a = a_0 \wedge b = b_0 > 0 \wedge 1 = 1)$
2.  $(a = a_0 \wedge b = b_0 > 0 \wedge p = 1) \Rightarrow (p = a_0^{b_0-b} \wedge a = a_0 \wedge b \geq 0)$
3.  $(p = a_0^{b_0-b} \wedge a = a_0 \wedge b \geq 0 \wedge (b \leq 0)) \Rightarrow (p = a_0^{b_0})$
4. as condições de verificação associadas ao programa anotado

$$\begin{aligned} & \{ p = a_0^{b_0-b} \wedge a = a_0 \wedge b \geq 0 \wedge (b > 0) \} \\ & \quad p = p * a; \quad b = b - 1 \\ & \{ p = a_0^{b_0-b} \wedge a = a_0 \wedge b \geq 0 \} \end{aligned}$$

que por sua vez corresponde à condição

$$\begin{aligned} (a) \quad & (p = a_0^{b_0-b} \wedge a = a_0 \wedge b \geq 0 \wedge (b > 0)) \\ & \Rightarrow (p * a = a_0^{b_0-(b-1)} \wedge a = a_0 \wedge b - 1 \geq 0) \end{aligned}$$

Uma alternativa à anotação que escrevemos antes do ciclo seria repetir o invariante do ciclo. Esta alternativa é muito usual e transforma a segunda das condições de verificação numa implicação trivial que é muitas vezes omitida.

## 5.4 Algoritmo de Euclides

O algoritmo seguinte calcula o máximo divisor comum entre dois números inteiros positivos ( $mdc$ )

```
while (a != b)
  if (a < b)
    b = b-a;
  else
    a = a-b;
```

A especificação informal feita acima pode ser feita usando os seguintes predicados.

**Pré-condição:**  $a = a_0 > 0 \wedge b = b_0 > 0$

**Pós-condição:**  $a = mdc(a_0, b_0)$

Para provarmos a correcção deste algoritmo vamos usar como invariante o seguinte predicado:

$$I \doteq mdc(a, b) = mdc(a_0, b_0) \wedge a > 0 \wedge b > 0$$

Vamos ainda usar o seguinte teorema (de Euclides)

$$mdc(x, y) = mdc(x + y, y) = mdc(x, x + y)$$

Usando as regras apresentadas, podemos anotar o algoritmo acima.

```
{ a = a_0 > 0 ∧ b = b_0 > 0 }
while (a != b)
  { mdc(a, b) = mdc(a_0, b_0) ∧ a > 0 ∧ b > 0 }
  if (a < b)
    b = b-a;
  else
    a = a-b;
{ a = mdc(a_0, b_0) }
```

Dando origem às seguintes condições de verificação:

1.  $(a = a_0 > 0 \wedge b = b_0 > 0) \Rightarrow (mdc(a, b) = mdc(a_0, b_0) \wedge a > 0 \wedge b > 0)$
2.  $((mdc(a, b) = mdc(a_0, b_0) \wedge a > 0 \wedge b > 0) \wedge \neg(a \neq b)) \Rightarrow (a = mdc(a_0, b_0))$
3. as condições de verificação associadas ao programa anotado

```
{ (mdc(a, b) = mdc(a_0, b_0) ∧ a > 0 ∧ b > 0) ∧ (a != b) }
if (a < b)
  b = b-a;
else
  a = a-b;
{ ((mdc(a, b) = mdc(a_0, b_0) ∧ a > 0 ∧ b > 0)) }
```

que correspondem às condições de verificação de

- (a)  $\{ (mdc(a, b) = mdc(a_0, b_0) \wedge a > 0 \wedge b > 0) \wedge (a \neq b) \wedge (a < b) \}$   
 $b = b - a;$   
 $\{ ((mdc(a, b) = mdc(a_0, b_0) \wedge a > 0 \wedge b > 0)) \}$   
 ou seja, que  
 $((mdc(a, b) = mdc(a_0, b_0) \wedge a > 0 \wedge b > 0) \wedge (a \neq b) \wedge (a < b))$   
 $\Rightarrow mdc(a, (b - a)) = mdc(a_0, b_0) \wedge a > 0 \wedge (b - a) > 0$
- (b)  $\{ (mdc(a, b) = mdc(a_0, b_0) \wedge a > 0 \wedge b > 0) \wedge (a \neq b) \wedge (a \geq b) \}$   
 $a = a - b;$   
 $\{ ((mdc(a, b) = mdc(a_0, b_0) \wedge a > 0 \wedge b > 0)) \}$   
 ou seja, que  
 $((mdc(a, b) = mdc(a_0, b_0) \wedge a > 0 \wedge b > 0) \wedge (a \neq b) \wedge (a \geq b))$   
 $\Rightarrow (((mdc((a - b), b) = mdc(a_0, b_0) \wedge (a - b) > 0 \wedge b > 0)))$

## 5.5 Multiplicação por incrementos

O seguinte algoritmo calcula o produto de dois números inteiros (não negativos) por sucessivos incrementos.

```
m = 0;
i = 0;
while (i < x)
{ j = 0;
  while (j < y)
    { j = j + 1; m = m + 1 };
  i = i + 1}
```

A especificação deste algoritmo já foi apresentada acima:

**Pré-condição:**  $x = x_0 \geq 0 \wedge y = y_0 \geq 0$

**Pós-condição:**  $m = x_0 * y_0$

Para anotarmos convenientemente este programa precisamos definir os invariantes dos dois ciclos:

- O ciclo mais interior faz tantos incrementos quanto o valor da variável  $y$ , acabando por adicionar à variável  $m$  o valor de  $y$ .
- O ciclo mais exterior é executado tantas vezes quanto o valor da variável  $x$ , acabando por repetir a adição de  $y$  a  $m$ ,  $x$  vezes.

Importante será garantir que o valor das variáveis  $x$  e  $y$  não é alterado. Vamos então usar o seguinte invariante:

1. Para o ciclo exterior

$$I_1 \doteq x = x_0 \wedge y = y_0 \wedge i \leq x \wedge m = i * y$$

## 2. Para o ciclo interior

$$I_2 \doteq x = x_0 \wedge y = y_0 \wedge i \leq x \wedge j \leq y \wedge m = i * y + j$$

Vejamos então como resulta anotarmos o algoritmo acima com os predicados acima. Começemos por apresentar os sítios onde as anotações serão feitas.

```

{ }
  m = 0;
  i = 0;
  { }
  while (i < x)
    { }
    j = 0;
    { }
    while (j < y)
      { }
      j = j + 1; m = m + 1
      i = i + 1
    { }
  { }

```

Podemos então colocar a pré e pós condições no início e fim, e os invariantes junto às condições dos ciclos.

```

{ x = x0 ≥ 0 ∧ y = y0 ≥ 0 }
  m = 0;
  i = 0;
  { }
  while (i < x)
    { x = x0 ∧ y = y0 ∧ i ≤ x ∧ m = i * y }
    j = 0;
    { }
    while (j < y)
      { x = x0 ∧ y = y0 ∧ i ≤ x ∧ j ≤ y ∧ m = i * y + j }
      j = j + 1; m = m + 1
      i = i + 1
    { m = x0 * y0 }

```

As restantes (duas) anotações devem reflectir o efeito das atribuições que as precedem.

O programa completamente anotado é:

```

{ x = x0 ≥ 0 ∧ y = y0 ≥ 0 }
m = 0;
i = 0;
{ x = x0 ≥ 0 ∧ y = y0 ≥ 0 ∧ m = 0 ∧ i = 0 }
while (i < x)
  { x = x0 ∧ y = y0 ∧ i ≤ x ∧ m = i * y }
  j=0;
  { x = x0 ∧ y = y0 ∧ i ≤ x ∧ m = i * y ∧ j = 0 ∧ i < x }
  while (j < y)
    { x = x0 ∧ y = y0 ∧ i ≤ x ∧ j ≤ y ∧ m = i * y + j }
    j=j+1; m=m+1
  i=i+1
{ m = x0 * y0 }

```

Vejamos então as condições de verificação geradas por este programa anotado.

1.  $(x = x_0 \geq 0 \wedge y = y_0 \geq 0)$   
 $\Rightarrow (x = x_0 \geq 0 \wedge y = y_0 \geq 0 \wedge 0 = 0 \wedge 0 = 0)$
2.  $(x = x_0 \geq 0 \wedge y = y_0 \geq 0 \wedge m = 0 \wedge i = 0)$   
 $\Rightarrow (x = x_0 \wedge y = y_0 \wedge i \leq x \wedge m = i * y)$
3.  $(x = x_0 \wedge y = y_0 \wedge i \leq x \wedge m = i * y \wedge i \geq x)$   
 $\Rightarrow (m = x_0 * y_0)$
4. juntamente com as condições de verificação associadas ao seguinte programa anotado

```

{ x = x0 ∧ y = y0 ∧ i ≤ x ∧ m = i * y ∧ i < x }
j=0;
{ x = x0 ∧ y = y0 ∧ i ≤ x ∧ m = i * y ∧ i < x ∧ x - i = i0 ∧ j = 0 }
while (j < y)
  { x = x0 ∧ y = y0 ∧ i ≤ x ∧ j ≤ y ∧ m = i * y + j }
  j=j+1; m=m+1
i=i+1
{ x = x0 ∧ y = y0 ∧ i ≤ x ∧ m = i * y }

```

Que correspondem a

- (a)  $(x = x_0 \wedge y = y_0 \wedge i \leq x \wedge m = i * y \wedge i < x)$   
 $\Rightarrow (x = x_0 \wedge y = y_0 \wedge i \leq x \wedge m = i * y \wedge i < x \wedge 0 = 0)$
- (b)  $(x = x_0 \wedge y = y_0 \wedge i \leq x \wedge m = i * y \wedge i < x \wedge j = 0)$   
 $\Rightarrow (x = x_0 \wedge y = y_0 \wedge i \leq x \wedge j \leq y \wedge m = i * y + j)$
- (c)  $(x = x_0 \wedge y = y_0 \wedge i \leq x \wedge j \leq y \wedge m = i * y + j \wedge j \geq y)$   
 $\Rightarrow (x = x_0 \wedge (i + 1) \leq x \wedge m = (i + 1) * y)$

(d) E ainda às condições de verificação do programa anotado

$$\begin{aligned} & \{x = x_0 \wedge y = y_0 \wedge i \leq x \wedge j \leq y \wedge m = i * y + j \wedge j < y\} \\ & \quad j=j+1; \quad m=m+1 \\ & \{x = x_0 \wedge y = y_0 \wedge i \leq x \wedge j \leq y \wedge m = i * y + j\} \end{aligned}$$

que é apenas

$$\begin{aligned} \text{i. } & (x = x_0 \wedge y = y_0 \wedge i \leq x \wedge j \leq y \wedge m = i * y + j \wedge j < y) \\ & \Rightarrow (x = x_0 \wedge y = y_0 \wedge i \leq x \wedge j + 1 \leq y \wedge m + 1 = i * y + j + 1) \end{aligned}$$

**Exercício 5** Considere o seguinte código que calcula o menor valor de um dado vector.

```
m = A[0];
i = 1;
while (i < n) {
    if (m > A[i]) {
        m = A[i]; i = i+1;
    }
    else i = i+1;
```

De forma a provar a correcção face à seguinte especificação

**Pré-condição:**  $n > 0$

**Pós-condição:**  $\forall 0 \leq p < n. m \leq A[p]$

1. Anote o programa convenientemente.
2. Determine as condições de verificação associadas.
3. Mostre a validade dessas condições.

**Exercício 6** Considere o seguinte programa que calcula os  $n$  primeiros factoriais.

```
i = 0;
f[0] = 1;
while (i < n) {
    i = i+1; f[i] = i * f[i-1];
```

De forma a provar a correcção face à seguinte especificação

**Pré-condição:**  $n = n_0 \geq 0$

**Pós-condição:**  $\forall 0 \leq p \leq n_0. f[p] = p!$

1. Anote o programa convenientemente.
2. Determine as condições de verificação associadas.
3. Mostre a validade dessas condições.

## 6 Correção Total

O conceito de correção que temos vindo a usar não é suficiente para exprimir na totalidade a intuição que temos quanto à adequação de um programa face a uma especificação. Vejamos o seguinte exemplo extremo que evidencia tal desajuste.

Sejam  $P$  e  $Q$  dois quaisquer predicados e seja  $S$  o seguinte programa

```
while (True)
  Skip;
```

Para provarmos que este programa satisfaz a especificação em causa (i.e., qualquer especificação) usemos como invariante o predicado  $I \doteq \text{True}$ .

Vejamos então as condições de verificação que deveremos validar:

1.  $P \Rightarrow I$ , que no caso resulta em  $P \Rightarrow \text{True}$ , e que é trivialmente válido.
2.  $(I \wedge \neg c) \Rightarrow Q$ , que corresponde neste caso a  $(\text{True} \wedge \text{False}) \Rightarrow Q$ , ou seja  $\text{False} \Rightarrow Q$ , o que mais uma vez é trivialmente válido.
3.  $\{I \wedge c\} \text{Skip} \{I\}$ , que corresponde á validade do predicado  $(\text{True} \wedge \text{True}) \Rightarrow \text{True}$ , uma vez mais um predicado trivialmente válido.

Este exemplo evidencia um detalhe que sub-valorizámos quando apresentados a noção de validade de um triplo de Hoare. Recordemos que então definimos a validade de um triplo a partir da validade de um predicado após o programa terminar. Ora se o programa não terminar (como é o caso do exemplo apresentado), tal estado nunca será atingido.

Para evitar estes casos é necessária uma noção de validade mais forte que garanta, para além do que foi enunciado, que partindo de um estado onde a pré-condição é válida, o programa termina.

Tal noção é conhecida por **correção total** e escrevemos

$$[P] S [Q]$$

e lê-se *o programa  $S$  está (totalmente) correcto face à especificação  $(P, Q)$* , com o seguinte significado:

1. Desde que  $P$  seja válido, o programa  $S$  termina
2. Se, a partir de todos os estados em que  $P$  é válido, executarmos o programa  $S$ , depois dessa execução terminar, atingimos estados em que  $Q$  é válido.

Em termos de regras de inferência sobre correção total convem referir que, a menos da regra relativa aos ciclos, todas as regras apresentadas para a correção parcial permanecem válidas. E isto porque nesses casos a terminação de um programa depende única e exclusivamente da terminação dos seus vários constituintes.

A única regra que teremos que alterar é a regra do ciclo, uma vez que a terminação da execução do corpo do ciclo não é suficiente para estabelecer a terminação do ciclo.

Para fazermos isso vamos precisar de introduzir um conceito novo – **variante**, que será usado na prova da terminação dos ciclos.

Enquanto que o invariante de um ciclo é um predicado que mostramos ser sempre válido, o variante de um ciclo é uma expressão **inteira** que:

1. decresce (estritamente) em cada iteração do ciclo,
2. permanece sempre maior do que um determinado valor pré-determinado (tipicamente 0).

De forma a traduzir esta noção de correcção total vamos usar a seguinte regra de inferência.

#### Ciclo-T

$$\frac{P \Rightarrow I \quad I \wedge c \Rightarrow V \geq 0 \quad [I \wedge c \wedge V = v_0] S [I \wedge V < v_0] \quad (I \wedge \neg c) \Rightarrow Q}{[P] \text{ while } c S [Q]} \quad (\text{Ciclo-T})$$

Nos exemplos que fomos apresentando ao longo deste texto a definição do variante é normalmente fácil.

Vejamos por exemplo o código apresentado no Exemplo 11 (pag. 12). O código aí apresentado era:

```
m = 0; d = y;
while (d>0) {
  m = m + x; d = d-1;
}
```

É fácil ver que a expressão *d* satisfaz os requisitos enumerados para o variante:

1. Decresce (estritamente) em cada iteração do ciclo.
2. Nunca desce abaixo de um valor fixo (a condição do ciclo assim o determina neste caso).

Note-se que se alterarmos este programa para

```
m = 0; d = y;
while (d!=0) {
  m = m + x; d = d-1;
}
```

a prova de que essa mesma expressão (i.e. *x*) é um variante vai obrigar a usar o invariante para provar que essa expressão é sempre não negativa.

A determinação de um variante não é, no entanto, sempre tão trivial. Veja-se por exemplo o famoso procedimento **hotpo** (*half or triple plus one*):

```
// n>0
while (n!=1)
  if (n % 2 == 0) n = n/2;
  else n = 3*n + 1;
// n==1
```



**Exercício 7** Prove que o algoritmo de Euclides apresentado atrás termina. Para isso determine um variante do ciclo em causa e determine as condições necessárias à correcção total do dito algoritmo.

**Exercício 8** Considere a seguinte especificação de um programa que calcula o somatório dos elementos de um vector.

**Pré-condição:**  $\forall_{0 \leq i < N} A[i] = a_i$

**Pós-condição:**  $s = \sum_{i=0}^{N-1} a_i$

Encontre variantes e invariantes de ciclo que lhe permitam provar a correcção total dos seguintes programas (face a essa especificação).

```
1. s = 0; p = 0;
   while (p < N) {
       s = s + A[p]; p = p+1;
   }
```

```
2. s = 0; p = N;
   while (p > 0) {
       p = p-1; s = s + A[p];
   }
```

**Exercício 9** Considere a seguinte especificação de um programa que calcula o factorial de um número inteiro não negativo.

**Pré-condição:**  $x = x_0 \geq 0$

**Pós-condição:**  $f = x_0! = \prod_{i=1}^{x_0} i$

Encontre variantes e invariantes de ciclo que lhe permitam provar a correcção total dos seguintes programas (face a essa especificação):

```
1. f = 1; i = 0;
   while (i < x) {
       i = i+1;
       f = f * i;
   }
```

```
2. f = 1;
   while (x > 0) {
       f = f * x;
       x = x-1;
   }
```

**Exercício 10** Considere a seguinte especificação de um programa que coloca em  $p$  o índice do maior elemento de um array

**Pré-condição:**  $\forall 0 \leq i < N \ A[i] = a_i$

**Pós-condição:**  $\forall 0 \leq i < N (A[i] = a_i) \wedge \forall 0 \leq i < N (A[i] \leq A[p])$

Encontre o variante e invariante de ciclo que lhe permita provar a correcção total do seguinte programa (face a essa especificação):

```
p = 0; i = 1;
while (i < N) {
  if (A[i] > A[p]) p = i;
  i = i + 1;
}
```

## 6.1 Condições de verificação

Ao analisarmos a correcção total de um programa vamos ter que mudar a sintaxe das nossas anotações de forma a gerar também as condições que permitem provar que os ciclos terminam.

Assim, as anotações que aparecem depois da condição dos ciclos têm que ter duas componentes:

- Um predicado  $I$  que corresponde ao invariante do ciclo
- Uma expressão inteira  $V$  que corresponde ao variante do ciclo.

A geração de condições tem então uma nova forma para quando se trata de um bloco começado por um **while**.

Ao bloco constituído por um **while** seguido de uma sequência de atribuições

$$\begin{array}{l} \{ P \} \\ \text{while } c \\ \{ I \} [V] \\ S; \\ x_1 = E_1; \quad x_2 = E_2; \quad \dots \quad x_n = E_n \{ Q \} \end{array}$$

correspondem:

1. a condição  $P \Rightarrow I$
2. a condição  $P \wedge c \Rightarrow V \geq 0$
3. a condição  $(I \wedge \neg c) \Rightarrow (Q[x_n \setminus E_n] \dots [x_2 \setminus E_2][x_1 \setminus E_1])$
4. bem como as condições de verificação associadas ao programa anotado

$$\{ I \wedge c \wedge V = v_0 \} \quad S \quad \{ I \wedge V < v_0 \}$$

## 7 Construção

Nesta secção vamos usar os conceitos expostos atrás, não para provar a correcção de um dado programa, mas como guia na escrita de programas que satisfaçam uma dada especificação.

### 7.1 Multiplicação inteira

Relembremos o programa analisado no Exemplo 11.

**Pré-condição:**  $x = x_0 \wedge y = y_0 \geq 0$

**Pós-condição:**  $m = x_0 * y_0$

Para a mesma especificação, vamos tentar obter um programa mais eficiente, que por cada iteração do ciclo, em vez de decrescer  $y$  apenas uma unidade, o vai dividir por 2. Note-se que, tanto a divisão (inteira) por 2 como a multiplicação por 2 são operações muito eficientes e que correspondem a um deslocamento (*shift*) dos *bits* do número. Em **C**, isto pode ser feito com as operações  $>>1$  (a divisão inteira por 2) e  $<<1$  (a multiplicação por 2).

Pretendemos então completar o programa abaixo (i.e., determinar **s1**, **S2** e **S3**) de forma a que ele satisfaça a especificação apresentada nesse exemplo.

```
S1;
while (y>0)
  if (y % 2 == 1) {
    S2;
    y = y>>1; // o que é equivalente a y=(y-1)/2
  } else {
    S3;
    y = y>>1; // o que é equivalente a y = y/2
  }
```

Uma vez que este programa partilha com o que foi apresentado a especificação e a condição do ciclo, podemos tentar usar como variante e invariantes versões próximas das que foram usadas atrás. Usaremos então os seguintes

$$\begin{aligned} I &\doteq x * y + m = x_0 * y_0 \wedge y \geq 0 \\ V &\doteq y \end{aligned}$$

Anotemos então o programa com os predicados acima.

```

{ x = x0 ∧ y = y0 ≥ 0 }
S1;
{ x * y + m = x0 * y0 ∧ y ≥ 0 }
while (y > 0)
  { x * y + m = x0 * y0 ∧ y ≥ 0 }[y]
  if (y % 2 == 1) {
    S2;
    y = y >> 1; // o que é equivalente a y = (y-1)/2
  } else {
    S3;
    y = y >> 1; // o que é equivalente a y = y/2
  }
{ m = x0 * y0 }

```

As condições de verificação associadas a este programa anotado são:

1.  $(x * y + m = x_0 * y_0 \wedge y \geq 0 \wedge y > 0) \Rightarrow (y > 0)$
2.  $(x * y + m = x_0 * y_0 \wedge y \geq 0 \wedge y \leq 0) \Rightarrow (m = x_0 * y_0)$
3. as condições de verificação associadas ao programa

```

{ x = x0 ∧ y = y0 ≥ 0 }
S1;
{ x * y + m = x0 * y0 ∧ y ≥ 0 }

```

Uma forma expedita de garantir tal é atribuir 0 à variável m.

4. as condições de verificação associadas ao programa

```

{ x * y + m = x0 * y0 ∧ y ≥ 0 ∧ y > 0 ∧ y % 2 == 1 ∧ y = y0 }
S2
{ x * (y - 1) / 2 + m = x0 * y0 ∧ (y - 1) / 2 ≥ 0 ∧ (y - 1) / 2 < y0 }

```

Para descobrirmos o comando S2, podemos apresentar a primeira parte da pré condição de uma outra forma:

```

x * y + m = x0 * y0
⇔ x * (y - 1 + 1) + m = x0 * y0
⇔ x * (y - 1) + x + m = x0 * y0
⇔ (2 * x) * (y - 1) / 2 + (x + m) = x0 * y0

```

Justificando que o comando S2 corresponda a  $m = m + x; x = x << 1$ .

5. as condições de verificação associadas ao programa

```

{ x * y + m = x0 * y0 ∧ y ≥ 0 ∧ y > 0 ∧ y % 2 != 1 ∧ y = y0 }
S3
{ x * y / 2 + m = x0 * y0 ∧ y / 2 ≥ 0 ∧ y / 2 < y0 }

```

De igual forma, podemos apresentar a primeira componente da pré condição de forma a descobrirmos S3.

$$\begin{aligned} \mathbf{x} * \mathbf{y} + \mathbf{m} &= x_0 * y_0 \\ \Leftrightarrow (2 * \mathbf{x}) * \mathbf{y}/2 + \mathbf{m} &= x_0 * y_0 \end{aligned}$$

Justificando que o comando S3 corresponda a  $\mathbf{x} = \mathbf{x} \ll 1$ .

O programa resultante é então:

```
m = 0;
while (y>0)
  if (y % 2 == 1) {
    m = m + x;
    x = x<<1;
    y = y>>1; // o que é equivalente a y=(y-1)/2
  } else {
    x = x<<1;
    y = y>>1; // o que é equivalente a y = y/2
  }
```

## 7.2 Valor de um polinómio num ponto

Um polinómio pode ser representado por um vector em que na posição  $i$  se guarda o coeficiente de grau  $i$ .

Nesta representação, o cálculo do valor de um dado polinómio (de grau  $n$ ) num ponto pode ser especificado por:

<p><b>Pré-condição:</b> <math>(n = n_0 \geq 0) \wedge (\mathbf{x} = x_0) \wedge (\forall_{0 \leq j \leq n_0} p[j] = p_j)</math></p> <p><b>Pós-condição:</b> <math>(\forall_{0 \leq j \leq n_0} p[j] = p_j) \wedge (\mathbf{r} = \sum_{j=0}^{n_0} p_j * x_0^j)</math></p>
--

Uma forma de cumprir esta especificação é através de um ciclo que vai somando os valores dos vários monómios. Teremos então:

```
i=0; r=0;
while (i<=n) {
  r = r + p[i] * x^i;
  i = i+1
}
```

**Exercício 11** A prova da correcção deste programa face à especificação apresentada pode ser feita à custa dos seguintes:

$$\begin{aligned} I &\doteq (\forall_{0 \leq j \leq n_0} p[j] = p_j) \wedge (n = n_0) \wedge (\mathbf{r} = \sum_{j=0}^{i-1} p_j * x_0^j) \wedge (i \leq n + 1) \\ V &\doteq n - i + 1 \end{aligned}$$

Anote o programa acima convenientemente, e calcule as condições de verificação associadas.

O programa acima é pouco eficiente pois refaz muitos cálculos: em cada iteração é calculada uma potência de  $x$  que não será usada para calcular a próxima potência. Podemos contornar este problema apresentando a seguinte variação (em que cada iteração actualiza o valor da potência de  $x$  necessária).

```
i=0; r=0; px = 1
while (i<=n) {
  r = r + p[i] * px;
  px = px * x;
  i = i+1
}
```

**Exercício 12** Prove a correcção deste programa face à especificação apresentada. Para isso determine o invariante e o variante do ciclo, anote o programa convenientemente, e calcule as condições de verificação associadas.

Uma outra optimização que se pode fazer sobre o programa original resulta da seguinte manipulação da pós condição:

$$\begin{aligned}
r &= \sum_{j=0}^{n_0} p_j * x_0^j \\
&= p_0 + x_0 * p_1 + x_0^2 * p_2 + x_0^3 * p_3 + \dots + x_0^{n_0} * p_{n_0} \\
&= p_0 + x_0 * (p_1 + x_0 * p_2 + x_0^2 * p_3 + \dots + x_0^{n_0} * p_{n_0-1}) \\
&= p_0 + x_0 * (p_1 + x_0 * (p_2 + x_0 * p_3 + \dots + x_0^{n_0} * p_{n_0-2})) \\
&= p_0 + x_0 * (p_1 + x_0 * (p_2 + x_0 * (p_3 + \dots + x_0^{n_0} * p_{n_0-3}))) \\
&= \dots
\end{aligned}$$

e que motiva o seguinte invariante

$$I \doteq (\forall_{0 \leq j \leq n_0} p[j] = p_j) \wedge (n = n_0) \wedge (r = \sum_{j=i+1}^n p_j * x_0^{j-(i+1)}) \wedge (i \geq -1)$$

O programa em causa continuará a percorrer o vector (desta vez do fim para o início) e guardando em  $r$  o somatório intermédio. Teremos por isso o seguinte esqueleto.

```
{ (n = n0 ≥ 0) ∧ (x = x0) ∧ (∀0 ≤ j ≤ n0 p[j] = pj) }
S1
{ P }
while c
{ (∀0 ≤ j ≤ n0 p[j] = pj) ∧ (n = n0) ∧ (r = ∑_{j=i+1}^n pj * x0^{j-(i+1)}) ∧ (i ≥ -1) }[]
S2
i=i-1
{ (∀0 ≤ j ≤ n0 p[j] = pj) ∧ (r = ∑_{j=0}^{n0} pj * x0^j) }
```

O predicado  $P$ , a ser estabelecido após o comando **S1**, deve garantir que o invariante seja válido antes da primeira iteração do ciclo. Uma forma expedita de o fazer é transformar o domínio de quantificação do somatório em vazio, sendo por isso o valor de  $r$  0. A condição  $c$  deve permitir que todos os coeficientes do polinómio sejam usados (incluindo a posição 0).

Estas observações permitem-nos apresentar uma versão mais detalhada do programa:

```

{ (n = n0 ≥ 0) ∧ (x = x0) ∧ (∀0 ≤ j ≤ n0 p[j] = pj) }
  r=0; i=n;
  { (n = n0 ≥ 0) ∧ (x = x0) ∧ (∀0 ≤ j ≤ n0 p[j] = pj) ∧ (r = 0) ∧ (i = n) }
  while (i >= 0)
    { (∀0 ≤ j ≤ n0 p[j] = pj) ∧ (n = n0) ∧ (r = ∑j=i+1n pj * x0j-(i+1)) ∧ (i ≥ -1) }[i + 1]
    S2
    i=i-1
  { (∀0 ≤ j ≤ n0 p[j] = pj) ∧ (r = ∑j=0n0 pj * x0j) }

```

Das condições de verificação associadas a este programa anotado (cuja escrita se deixa como exercício), vejamos a parte que diz respeito ao comando S2:

```

{ (r = ∑j=i+1n pj * x0j-(i+1)) ∧ (i ≥ -1) ∧ (i ≥ 0) ∧ (v0 = i + 1) }
  S2
{ (r = ∑j=(i-1)+1n pj * x0j-((i-1)+1)) ∧ (i - 1 ≥ -1) ∧ (v0 > (i - 1) + 1) }

```

Note-se que (por razões de espaço) retirámos dos predicados a parte que dizia respeito à preservação dos valores do vector e por isso devemos manter essa preocupação.

A pós condição acima ainda pode ser simplificada:

```

{ (r = ∑j=i+1n pj * x0j-(i+1)) }
  S2
{ (r = ∑j=in pj * x0j-i) }

```

Vejamos então como rescrever esta pós-condição:

$$\begin{aligned}
r &= \sum_{j=i}^n p_j * x_0^{j-i} \\
&= p_i * x_0^{i-i} + \sum_{j=i+1}^n p_j * x_0^{j-i} \\
&= p_i * x_0^0 + x_0 * \sum_{j=i+1}^n p_j * x_0^{j-i-1} \\
&= p_i + x_0 * \sum_{j=i+1}^n p_j * x_0^{j-(i+1)}
\end{aligned}$$

E por isso o comando S2 em causa não é mais do que  $r = p[i] + x * r$ .

O programa completo é então:

```

r=0; i=n;
while (i >= 0) {
  r = p[i] + x*r;
  i= i-1;
}

```

### 7.3 Divisão inteira

Um algoritmo naïve de cálculo da divisão inteira de dois números positivos consiste em sucessivamente subtrair o divisor ao dividendo. O resultado da divisão é o número de vezes que esta subtracção pode ser feita.

A especificação de tal programa já foi vista nos exemplo 5 da página 4.

O programa seguinte coloca em  $q$  e  $r$  respectivamente, o quociente e o resto da divisão inteira dos valores iniciais de  $x$  por  $y$ .

```

q = 0; r = x;
while (r >= y) {
  q = q+1;
  r = r-y;
}

```

A especificação apresentada no exemplo 5 é:

**Pré-condição:**  $(x = x_0 \geq 0) \wedge (y = y_0 > 0)$

**Pós-condição:**  $(0 \leq r < y_0) \wedge (q * y_0 + r = x_0)$

A prova da correcção pode ser feita usando os seguintes:

$$\begin{aligned}
I &\doteq (q * y_0 + r = x_0) \wedge (0 < y = y_0) \wedge (0 \leq r) \\
V &\doteq r
\end{aligned}$$

Podemos anotar o programa usando os predicados acima.

```

{ (x = x0 ≥ 0) ∧ (y = y0 > 0) }
q = 0; r = x;
{ (x = x0 ≥ 0) ∧ (y = y0 > 0) ∧ (q = 0) ∧ (r = x) }
while (r >= y)
  { (q * y0 + r = x0) ∧ (0 < y = y0) ∧ (0 ≤ r) }[r]
  q = q+1; r = r-y
{ (0 ≤ r < y0) ∧ (q * y0 + r = x0) }

```

**Exercício 13** Determine as condições de verificação associadas a este programa anotado e mostre a sua validade.

O programa apresentado acima vai iterar tantas vezes quanto o resultado da divisão inteira (isto porque cada iteração do ciclo acrescenta uma unidade ao resultado  $q$ ). Uma forma de tornar este algoritmo mais eficiente será, em cada iteração do ciclo, acrescentar ao resultado  $q$  uma quantidade maior (e consequentemente, retirar de  $r$  uma quantidade maior). Note-se que, de acordo com o invariante usado, sempre que se adiciona  $i$  a  $q$  deve-se retirar  $i*y$  a  $r$ .

A optimização que vamos apresentar de seguida, tenta, em cada iteração retirar a  $r$  um *grande* múltiplo de  $y$ . Para isso vamos acrescentar uma primeira etapa em que calcularemos um múltiplo de  $y$  para subtrair a  $r$ .

```

{ (x = x0 ≥ 0) ∧ (y = y0 > 0) }
S1
{ (x = x0 ≥ 0) ∧ (y = y0 > 0) ∧ (m = i * y0) }
q = 0; r = x;
{ (x = x0 ≥ 0) ∧ (y = y0 > 0) ∧ (m = i * y0) ∧ (q = 0) ∧ (r = x) }
while (r >= y)
  { (q * y0 + r = x0) ∧ (0 ≤ y = y0) ∧ (0 ≤ r) ∧ (m = i * y0) }[r]
S2
{ (0 ≤ r < y0) ∧ (q * y0 + r = x0) }

```



Comecemos então por determinar o programa S1. Este deve determinar um múltiplo de y, sem modificar os valores das variáveis x e y. Se tentarmos obter este múltiplo por sucessivas adições de y, vamos incorporar neste primeiro passo, todo o cálculo necessário. Alternativamente podemos ir obtendo este múltiplo, multiplicando-o por alguma quantidade (relembremos que a multiplicação por 2 é uma operação tão ou mais simples do que a adição). Teremos então.

```

{ (x = x0 ≥ 0) ∧ (y = y0 > 0) }
S1.1
{ ??? ∧ (x = x0 ≥ 0) ∧ (y = y0 > 0) }
while (C1)
  { (m = y0 * i) ∧ (x = x0 ≥ 0) ∧ (y = y0 > 0) }[?]
  m=m << 1 // que e equivalente a m=m*2
  i=i << 1 // que e equivalente a i=i*2
{ (x = x0 ≥ 0) ∧ (y = y0 > 0) ∧ (m = y0 * i) }

```

De forma a conseguirmos estabelecer a validade do invariante temos que colocar em m um múltiplo de y (e em i a respectiva multiplicidade).

Quanto à condição C1, e como queremos um múltiplo de y que se possa subtrair a x deveremos garantir que esse múltiplo não exceda x.

Teremos então:

```

{ (x = x0 ≥ 0) ∧ (y = y0 > 0) }
m = y; i = 1;
{ (m = y) ∧ (i = 1) ∧ (x = x0 ≥ 0) ∧ (y = y0 > 0) }
while (m < x)
  { (m = y0 * i) ∧ (m > 0) ∧ (x = x0 ≥ 0) ∧ (y = y0 > 0) }[x - m]
  m=m >> 1 // que e equivalente a m=m*2
  i=i >> 1 // que e equivalente a i=i*2
{ (x = x0 ≥ 0) ∧ (y = y0 > 0) ∧ (m > 0) ∧ (m = y0 * i) }

```

As condições de verificações associadas a este programa anotado são:

1.  $((x = x_0 \geq 0) \wedge (y = y_0 > 0))$   
 $\Rightarrow ((y = y) \wedge (1 = 1) \wedge (x = x_0 \geq 0) \wedge (y = y_0 > 0))$
2.  $((m = y) \wedge (i = 1) \wedge (x = x_0 \geq 0) \wedge (y = y_0 > 0))$   
 $\Rightarrow ((m = y_0 * i) \wedge (x = x_0 \geq 0) \wedge (m > 0) \wedge (y = y_0 > 0))$
3.  $((m = y_0 * i) \wedge (x = x_0 \geq 0) \wedge (y = y_0 > 0) \wedge (m > 0) \wedge (m < x))$   
 $\Rightarrow (x - m > 0)$
4.  $((m = y_0 * i) \wedge (x = x_0 \geq 0) \wedge (y = y_0 > 0) \wedge (m > 0) \wedge (m \geq x))$   
 $\Rightarrow ((x = x_0 \geq 0) \wedge (y = y_0 > 0) \wedge (m = y_0 * i))$
5.  $((m = y_0 * i) \wedge (x = x_0 \geq 0) \wedge (y = y_0 > 0) \wedge (m < x) \wedge (m > 0) \wedge (x - m = v_0))$   
 $\Rightarrow ((m * 2 = y_0 * (i * 2)) \wedge (x = x_0 \geq 0) \wedge (y = y_0 > 0) \wedge (m > 0) \wedge (x - m * 2 < v_0))$

**Exercício 14** Relembre o algoritmo apresentado na secção 5.3 para calcular a potência positiva de um número. Tal como foi feito acima, use as anotações para obter uma versão mais eficiente, que em vez de subtrair um elemento a b em cada iteração, divide o seu valor por 2.

## A Exercícios de exames

1. (*Teste, 2007/2008*)

Determine as condições de verificação necessárias para provar a correcção parcial do seguinte programa (anotado em comentários) usado para calcular o máximo divisor comum entre dois números positivos.

```
// (a == a0 > 0) /\ (b == b0 > 0)
while (a != b)
    // mdc(a0,b0) = mdc(a,b)
    if (a > b) a = a - b;
    else b = b - a;
// a == mdc(a0,b0)
```

2. (*Exame de recurso, 2007/2008*)

Determine as condições de verificação necessárias para provar a correcção parcial da seguinte função anotada.

```
int bubble(int a[], int n) {
    int i, k;
    // n > 0
    i = n-1; k = 0;
    // n > 0 && i == n-1 && k == 0
    while (i > 0) {
        // i >= 0 && k < n
        if (a[i] < a[i-1]) {swap(a,i,i-1); k++;}
        i--;
    }
    // k < n
    return k;
}
```

3. (*Época especial, 2007/2008*)

Determine as condições de verificação necessárias para provar a correcção parcial da seguinte função anotada, para cálculo do resto da divisão inteira.

```
int mod (int x, int y) {
    // x >= 0 && y > 0
    int r = x;
    // x >= 0 && y > 0 && r == x
    while (r>=y) {
```

```

    // r >= 0 && (exists q >= 0 : q * y + r = x)
    r = r - y;
}
// 0 <= r < y && (exists q >= 0 : q * y + r = x)
return r;

```

4. (*Época especial, 2007/2008*)

Determine um variante apropriado para demonstrar que a função acima termina. Apresente as condições necessárias para provar que tal acontece (i.e., que a função termina).

5. (*Teste, 2009/2010*)

Calcule as condições de verificação necessárias à prova de correcção parcial do seguinte programa (anotado em comentário //)

```

// True
v = 0; i = 0;
// v == 0 && i == 0
while (i <= N) {
    // v == sum (k=0..i-1) b[k] * 2^(N-k) && i <= N+1
    v = v*2 + b[i];
    i = i+1
}
// v == sum (k=0..N) b[k] * 2^(N-k)

```

6. (*Época especial, 2009/2010*)

Calcule as condições de verificação necessárias à prova de correcção parcial do seguinte programa (anotado em comentário //)

```

// s == 0 && i == 0
while (i < N) {
    // s == sum (k=0..i-1) a[k] && i <= N
    s = s + a[i];
    i = i+1;
}
// s == sum (k=0..N-1) a[k]
if (s > 10) res = 1;
else res = 0;
// sum (k=0..N-1) a[k] > 10 <==> res == 1

```

7. (*Época especial, 2009/2010*)

Suponha que o ciclo while do programa anterior é alterado para

```

while (i < N) { a[i] = 10;
                s = s + a[i];
                i = i+1; }

```

Explique porque é que a especificação do programa (anotações da alínea anterior) permanece válida. Apresente uma especificação alternativa para o programa original, que exclua esta solução claramente contra-intuitiva.

8. (*Época especial, 2009/2010*)

Relembre a função de partição usado no algoritmo de *quicksort*.

```
int partition (int v[], int l, int u) {
    int i, j;
    i = j = l;
    while j < u {
        if v[j] < v[u] { swap (v, i, j); i++; }
        j++;
    }
    swap (v, i, u);
    return i;
}
```

- (a) Apresente um invariante e um variante adequados à prova da correcção total desta função, face à seguinte especificação:

**pré-condição**  $l \leq u$

**pós-condição**  $(\forall_k. l \leq k < i \Rightarrow v[k] \leq v[i]) \wedge (\forall_k. i < k \leq u \Rightarrow v[i] \leq v[k])$

- (b) Usando este invariante e variante, apresente uma versão anotada desta função de acordo com as regras de anotação estudadas.

9. (*Teste, 2010/2011*)

Considere o seguinte programa (anotado em comentário //)

```
// (exists (i in [0..n-1]) v[i] == x)
k = 0;
// (exists (i in [k..n-1]) v[i] == x)
while (v[k] != x)
    // (exists (i in [k..n-1]) v[i] == x)
    k=k+1
// v[k] == x
```

- (a) Apresente as condições de verificação necessárias à prova da correcção parcial deste programa.
- (b) Indique um variante do ciclo em causa, e justifique informalmente porque se trata efectivamente de um variante.

10. (*Exame, 2010/2011*)

Considere o seguinte programa (anotado em comentário)

Apresente as condições de verificação necessárias à prova da correcção parcial deste programa.

```

// x == x0 >= 0
a = x; b = 0;
// (x == x0 >= 0) /\ (b <= sqrt x0 <= a)
while ((a-b) > epsilon) {
    // (x == x0 >= 0) /\ (b <= sqrt x0 <= a)
    m = (a+b)/2;
    // (x == x0 >= 0) /\ (b <= sqrt x0 <= a) /\ (m = (a+b)/2)
    if (m^2 > x) a = m; else b = m;
}
// b <= sqrt x0 <= a

```

11. (*Exame, 2010/2011*)

Relativamente ao programa apresentado na alínea anterior, apresente um variante que lhe permita provar a **correção total** do programa.

Apresente ainda as condições adicionais (às referidas nessa alínea) necessárias para esta prova.

Note que todas as variáveis em questão são `float`.

12. (*Época especial, 2010/2011*)

Considere o seguinte programa (anotado em comentário)

Apresente as condições de verificação necessárias à prova da correção parcial deste programa.

```

// n == n0 > 0
x=1; y=0;
// n == n0 > 0 && x==1 && y == 0
while (n>1) {
    // x == fib (n0 - n + 1) && y == fib (n0 - n) && n >= 1
    x = x+y; y = x-y; n = n-1;
}
// x == fib (n0)

```

13. (*Época especial, 2010/2011*)

Aumente as anotações do programa apresentado na questão anterior de forma a poder provar a sua correção total. Apresente ainda as novas condições de verificação que resultam dessa nova anotação do programa.

14. (*Teste, 2011/2012*)

Suponha que se usam vectores de coeficientes para representar polinómios. O polinómio  $4.2x^5 - 3.2x^2 - 0.5$  será representado por um array em que as primeiras seis componentes são -0.5, 0, -3.2, 0, 0 e 4.2.

A seguinte função calcula o valor de um polinómio  $p$  (de grau  $n-1$ ) num ponto.

<pre>float valor(float p[],int n,float x){     int i; float r;     i = n; r = 0;     while (i&gt;0) {         i = i-1;         r = r * x + p[i];     }     return r; }</pre>	<p>Para provar que esta função satisfaz a especificação</p> <p><b>pré-condição:</b> <math>n \geq 0</math></p> <p><b>pós-condição:</b> <math>r = \sum_{k=0}^{n-1} p_k * x^k</math> (em que <math>p_k</math> corresponde ao valor de <math>p[k]</math>),</p>
--	--

vamos usar como invariante o predicado

**invariante:**  $(i \geq 0) \wedge (r = \sum_{k=i}^{n-1} p_k * x^k)$

Anote correctamente o programa e, a partir do programa anotado, gere as condições de verificação necessárias para a prova da correcção **parcial** desta função (face à especificação fornecida).

15. (*Teste, 2012/2013*)

Apresente as condições de verificação necessárias à prova da correcção parcial do seguinte programa anotado que calcula o número representado num array de (N) bits.

```
// N >= 0
n = 0; i = 0;
// N>=0 && n == 0 && i == 0
while (i<N) {
    // n = sum{k=0}{k=i-1} b[k] * (2^k)
    // && i<=N
    n = n*2 + b[i];
    i = i+1;
}
// n == sum{k=0}{k=N-1} b[k] * (2^k)
```

16. (*Teste, 2012/2013*)

Considere o seguinte programa, parcialmente anotado, para calcular a representação binária de um número inteiro positivo.

```
// n == n0 >= 0
i = 0;
// n == n0 >= 0 && i == 0
while (n > 0) {
    // ...?...
    b[i] = n%2;
    i = i+1; n = n/2;
}
// n0 == sum{k=0}{k=i-1} b[k] * (2^k)
```

Determine um invariante que lhe permita provar a correcção parcial deste excerto de código. Mostre que o invariante que apresentou é preservado (em cada iteração).

17. (*Exame de recurso, 2012/2013*)

Apresente as condições de verificação necessárias à prova da correcção parcial do

seguinte programa anotado para calcular a representação binária de um número inteiro positivo.

```
// n == n0 >= 0
i = 0;
// n == n0 >= 0 && i == 0
while (n > 0) {
    // n0 = (n*2^i) + sum{k=0}{k=i-1} b[k] * (2^k) && n >= 0
    if (n%2 == 0) { b[i] = 0; n = n/2;}
    else { b[i] = 1; n = (n-1)/2;}
    i = i+1;
}
// n0 = sum{k=0}{k=i-1} b[k] * (2^k)
```

18. (*Época especial, 2012/2013*)

Apresente as condições de verificação necessárias à prova da correcção parcial do seguinte programa anotado para calcular a raiz quadrada inteira de um número.

```
// n == n0 > 0
r = 1;
// n == n0 >= 0 && r == 1
while (r*r < n)
    // n == n0 >= 0 && (r-1)*(r-1) <= n
    r = r+1;
// n == n0 >= 0 && (r-1)*(r-1) <= n && r * r >= n
if (r * r > n) r = r-1;
// r*r <= n0 && (r+1)*(r+1) > n0
```

19. (*Época especial, 2012/2013*)

Considere a seguinte alternativa para o problema apresentado na alínea anterior.

```
// n == n0 > 0
r = 0 ; s = n;
// .....
while (r < s-1) {
    // .....
    m = (r+s)/2;
    // ....
    if (m*m > n) s = m;
    else r = m;
}
// r*r <= n0 && (r+1)*(r+1) > n0
```

Complete as anotações do código acima de forma a provar a sua correcção **parcial**.

20. (*Teste, 2013/2014*)

Considere o programa que se segue, anotado com uma pré-condição, um invariante de ciclo, e uma pós-condição. Gere as respectivas condições de verificação para provar a correcção parcial, começando por acrescentar ao código outras anotações necessárias.

```
// N >= 0
k = 0;
while ((k<N/2) && (v[k]==v[N-k-1])) {
    // (k <= N/2) && forall i; (0 <= i < k) ==> v[i] == v[N-i-1]
    k++;
}
if (k==N/2) r = 1;
else r = 0;
// (r == 1 && forall i; (0 <= i < N/2) ==> v[i] == v[N-i-1])
// (r == 0 && exists i; (0 <= i < N/2) && v[i] != v[N-i-1])
```

21. (*Teste, 2014/2015*)

Considere o seguinte programa parcialmente anotado:

```
// 0 <= a <= b && forall (a <= i <= b) v[i] == v_i
j=a; r=0;
while (j<=b) {
    r = r+v[j]; j=j+1;
}
// (forall (a <= i <= b) v[i] == v_i)
// && r == sum (a <= i <= b) v_i
```

- (a) Descreva informalmente (i.e., sucintamente e por palavras) a especificação deste programa.
- (b) Complete a anotação do programa (fornecendo o invariante do ciclo) e apresente as condições de verificação correspondentes à prova da **correcção parcial** do programa em causa.

22. (*Exame de recurso, 2014/2015*)

Considere o programa para calcular o quociente  $q$  entre inteiros  $b$  e  $a$



- Apresente as condições de verificação necessárias à prova da sua correcção parcial utilizando o símbolo I no lugar do invariante de ciclo.
  - Anote o programa com um invariante e um variante de ciclo que lhe permitam provar a correcção total do algoritmo.
- ```
// a == a0 > 0 && b == b0 >= 0
int q, m;
q=0; m=0;
while (b>=m+a) {
    // I
    m = m+a; q=q+1
}
// q*a0 + (b0-m) == b0
// && 0 <= m <= b0
```

23. (*Época especial, 2014/2015*)

Considere a função (anotada) que calcula o índice do maior elemento de um array de inteiros.

- Determine as condições de verificação resultantes da anotação apresentada.
- Apresente um invariante I que permita provar a correcção parcial da função acima, i.e., que faça com que as condições referidas na alínea anterior sejam válidas.

```
int maxInd (int v[], int n){
    int r, i;
    // n > 0
    r = 0; i = 1;
    // I
    while (i<n){
        // I
        if (v[i] > v[r]) r=i;
        i++;
    }
    // forall 0<=k<n v[k] <= v[r]
    return r;
}
```

24. Considere o seguinte excerto de uma função de procura binária num array ordenado de inteiros.

- Complete a anotação desta função apresentando um invariante I, que lhe permita provar (apenas) que o programa é correcto face à pré-condição e pós-condição anotadas no código.
- *Teste, 2015/2016*  
Apresente e prove as condições de verificação geradas a partir destas anotações.

```
// PRE: n > 0
l = -1; r = n;
while (l+1 != r) {
    m = (l+r)/2;
    // I && m == (l+r)/2
    if (a[m] <= x) l = m;
    else r = m;
}
// POS: -1 <= l < n
```

25. (*Exame de recurso, 2015/2016*)

Considere o seguinte programa (parcialmente) anotado.

```
// n == n0 > 1 && f == 2
while (n%f != 0)
    // I
    f = f+1;
// n0 % f == 0
// && forall(1<k<f) n0%k != 0
```

- (a) Descreva por palavras a especificação apresentada.
- (b) Apresente um variante e um invariante que lhe permitam provar a **correção total** do programa face à especificação apresentada.
- (c) Gere as condições de verificação correspondentes apenas à **correção parcial** do programa.

26. (*Época especial, 2015/2016*)

Considere a função (anotada) que calcula o índice do menor elemento de um array de inteiros.

- (a) Determine as condições de verificação resultantes da anotação apresentada.
- (b) Apresente um invariante I que permita provar a correção parcial da função acima, i.e., que faça com que as condições referidas na alínea anterior sejam válidas.

```
int minInd (int v[], int n){
    int r, i;
    // n > 0
    r = 0; i = 1;
    // I
    while (i<n){
        // I
        if (v[i] < v[r]) r=i;
        i++;
    }
    // forall 0<=k<n v[k] >= v[r]
    return r;
}
```

27. (*Teste, 2016/2017*)

Considere o seguinte excerto de uma função de partição para ser usada no algoritmo de *Quick-sort*.

- Complete a anotação desta função apresentando um invariante I, que lhe permita provar (apenas) que o programa é correcto face à pré-condição e pós-condição anotadas no código.
- Apresente as condições de verificação geradas a partir destas anotações.

```
// PRE: n > 0
i=0; j=n-2; k=0;
while (k<n-1) {
    // I
    if (a[k] <= a[n-1]) {
        b[i] = a[k]; i=i+1;
    } else {
        b[j] = a[k]; j=j-1;
    }
    k=k+1;
}
// POS: j == i-1
```

28. (*Exame de recurso, 2016/2017*)

Considere a função ao lado que coloca em r o índice do menor elemento do vector v.

Anote convenientemente o programa e calcule as condições de verificação resultantes.

```
// N > 0
r = 0; i = 1;
while (i < N) {
    if (v[i] < v[r]) r = i;
    i = i+1;
}
// forall (0<=k<N) v[k] >= v[r]
```

29. (*Época especial, 2016/2017*)

Considere as seguintes definições.

```
void insert (int v[], int N, int x){    void isort (int v[], int N){
    int i;                               int i;
    for (i=N-1; i>=0 && v[i]>x; i--)      for (i=1;i<N;i++)
        v[i+1]=v[i];                     insert (v,i,v[i]);
    v[i+1] = x;                           }
}
```

De forma a estabelecer que a função `isort` ordena o array que recebe como argumento:

- Apresente (por palavras, se preferir) uma especificação (i.e. pré e pós-condição) adequada para a função `insert`.
- Defina (por palavras, se preferir) um invariante do ciclo usado na função `isort`, que permita provar que no final o array está ordenado.