Notas Sobre Correcção de Algoritmos

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.
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 às linguagens de programação imperativa nos quais seja possível expressar essa ligação bem como provar que um dado programa satisfaz uma dada especificação.
Nestas notas apresenta-se, de uma forma muito introdutória, um desses mecanismos – *triplos de Hoare*. Veremos como estes podem ser usados para provar 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 derivar um algoritmo a partir de uma dada especificação.

2 Especificações

Para especificar um algoritmo vamos usar dois predicados:

- 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.

Assim sendo, dados

- Um programa S
- Dois predicados P e Q

escrevemos

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

com o seguinte significado:

1. Desde que P seja válido, o programa S termina
2. Se P for válido, Q é válido após a execução de S.

Se se verificar apenas a segunda das condições dizemos que o programa S está **parcialmente correcto** em relação à especificação.

Vejamos em primeiro lugar 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.

Pré-condição: $True$

Pós-condição: $\mathtt{x} = \mathtt{y} \wedge \mathtt{y} = \mathtt{x}$

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:

Pré-condição: $\mathtt{x} = x_0 \wedge \mathtt{y} = y_0$

Pós-condição: $\mathtt{x} = y_0 \wedge \mathtt{y} = x_0$

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

Pré-condição: $\mathtt{x} = x_0 \wedge \mathtt{y} = y_0 \geq 0$

Pós-condição: $\mathtt{m} = x_0 * y_0$

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: $\mathtt{x} = x_0 > 0 \wedge \mathtt{y} = y_0 \geq 0$

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: $\mathtt{x} = x_0 > 0 \wedge \mathtt{y} = y_0 \geq 0$

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: $\mathtt{x} = x_0 > 0 \wedge \mathtt{y} = y_0 \geq 0$

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: $(\forall_{a \leq i \leq b}\, .\, \mathtt{v}[i] = v_i) \wedge (\forall_{a \leq i < b}\, .\, v_i \leq v_{i+1})$

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

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.

3 Programas

Nesta secção vamos apresentar uma linguagem de programação muito simples mas que tem muitos dos ingredientes fundamentais das linguagens de programação imperativas: sequenciação, atribuição, condicionais e ciclos.
Para cada um destes construtores vamos apresentar regras de prova da correcção de programas que envolvam essas construções.

3.1 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 $P \Rightarrow Q$ 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 $\{\, P \,\}\ S\ \{\, Q \,\}$ queremos dizer que se P for válido 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.

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

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

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

3.2 Atribuição

A operação fundamental de qualquer linguagem de programação imperativa é a atribuição de um valor a uma variável. A seguinte regra de correcção traduz o significado da execução de uma atribuição.

Atribuição–1

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

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

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

3.3 Sequenciação

Uma outra construção fundamental de programas é a de sequenciação: executar um programa após outro. A regra de correcção associada a esta construção 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 7 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 2.
Usando a correcção da sequenciação, temos de encontrar predicados R_1 e R_2 tais que:

$$\begin{array}{l} \{\, \mathtt{x} = \mathtt{x}_0 \wedge \mathtt{y} = \mathtt{y}_0 \,\} \\ \mathtt{x = x + y} \\ \{\, \mathtt{R}_2 \,\} \\ \mathtt{y = x - y} \\ \{\, \mathtt{R}_1 \,\} \\ \mathtt{x = x - y} \\ \{\, \mathtt{x} = \mathtt{y}_0 \wedge \mathtt{y} = \mathtt{x}_0 \,\} \end{array}$$

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

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

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

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

Comecemos por simplificar o consequente desta implicação.

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

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

Exemplo 8 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:

$$\begin{array}{l} \{\, \mathtt{x} = \mathtt{x}_0 \wedge \mathtt{y} = \mathtt{y}_0 \,\} \\ \mathtt{z = x} \\ \{\, R_2 \,\} \\ \mathtt{x = y} \\ \{\, R_1 \,\} \\ \mathtt{y = z} \\ \{\, \mathtt{x} = \mathtt{y}_0 \wedge \mathtt{y} = \mathtt{x}_0 \,\} \end{array}$$

Donde vem:

- $\begin{array}{rcl} R_1 = & (\mathtt{x} = y_0 \wedge \mathtt{y} = x_0)[\mathtt{y} \setminus \mathtt{z}] \\ = & \mathtt{x} = y_0 \wedge \mathtt{z} = x_0 \end{array}$
- $\begin{array}{rcl} R_2 = & R_1[\mathtt{x} \setminus \mathtt{y}] \\ = & (\mathtt{x} = y_0 \wedge \mathtt{z} = x_0)[\mathtt{x} \setminus \mathtt{y}] \\ = & \mathtt{y} = y_0 \wedge \mathtt{z} = x_0 \end{array}$

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

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

Comecemos por simplificar o consequente desta implicação.

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

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

3.4 Condicionais

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

Condicional

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

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 9 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: $\mathtt{x} = x_0 \wedge \mathtt{y} = y_0$

Pós-condição: $\mathtt{M} = \mathtt{max}\ (x_0, y_0)$

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

$$
\begin{array}{l}
\{\, \mathtt{x} = \mathtt{x}_0 \wedge \mathtt{y} = \mathtt{y}_0 \,\} \\
\mathtt{if\ (x > y)} \\
\quad 1 \left[\begin{array}{l} \{\, \mathtt{x} > \mathtt{y} \wedge \mathtt{x} = \mathtt{x}_0 \wedge \mathtt{y} = \mathtt{y}_0 \,\} \\ \mathtt{M = x} \\ \{\, \mathtt{M} = \mathtt{max}\ (\mathtt{x}_0, \mathtt{y}_0) \,\} \end{array} \right. \\
\quad \mathtt{else} \\
\quad 2 \left[\begin{array}{l} \{\, \mathtt{x} \leq \mathtt{y} \wedge \mathtt{x} = \mathtt{x}_0 \wedge \mathtt{y} = \mathtt{y}_0 \,\} \\ \mathtt{M = y} \\ \{\, \mathtt{M} = \mathtt{max}\ (\mathtt{x}_0, \mathtt{y}_0) \,\} \end{array} \right.
\end{array}
$$

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

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

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

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

3.5 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 do programa da forma

```
while b S
```

vamos precisar de encontrar duas expressões:

Invariante do ciclo que é um predicado que vai traduzir o processo usado na obtenção do resultado, i.e., é um predicado que teremos de provar que é sempre verdadeiro e que no final do ciclo nos garante que a pós-condição é alcançada.

Variante do ciclo que é uma expressão inteira e não negativa e que decresce em cada iteração do ciclo. Daí que nos permita mostrar que o ciclo termina.

A regra de correcção fundamental para os ciclos é:

Ciclo-1

$$\frac{I \wedge c \Rightarrow V \geq 0 \quad \{\, I \wedge c \wedge V = v_0 \,\}\ S\ \{\, I \wedge V < v_0 \,\}}{\{\, I \,\}\ \texttt{while}\ c\ S\ \{\, I \wedge \neg c \,\}} \text{ (while-1)}$$

A premissa desta regra, por razões práticas é normalmente dividida em duas correspondentes à correcção parcial e à terminação de um ciclo.

Ciclo-2

$$\frac{\{\, I \wedge c \,\}\ S\ \{\, I \,\} \quad I \wedge c \Rightarrow V \geq 0 \quad \{\, I \wedge c \wedge V = v_0 \,\}\ S\ \{\, V < v_0 \,\}}{\{\, I \,\}\ \texttt{while}\ c\ S\ \{\, I \wedge \neg c \,\}} \text{ (while-2)}$$

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

Ciclo-3

$$\frac{\begin{array}{llll} P \Rightarrow I & I \wedge c \Rightarrow (V \geq 0) & \{\, I \wedge c \,\} & (I \wedge \neg c) \Rightarrow Q \quad \{\, I \wedge c \wedge V = v_0 \,\} \\ & & \texttt{S} & \texttt{S} \\ & & \{\, I \,\} & \{\, V < v_0 \,\} \end{array}}{\{\, P \,\}\ \texttt{while}\ c\ S\ \{\, Q \,\}} \text{ (while-3)}$$

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. $\mathbf{I} \wedge \mathbf{c} \Rightarrow (\mathbf{V} \geq \mathbf{0})$: antes de cada iteração o variante é não negativo.
3. $\{\, \mathbf{I} \wedge \mathbf{c} \,\}\ \mathbf{S}\ \{\, \mathbf{I} \,\}$: Assumindo que o invariante é válido antes de uma iteração do ciclo, ele continua válido depois dessa iteração.

4. $(\mathbf{I} \wedge \neg \mathbf{c}) \Rightarrow \mathbf{Q}$: Quando o ciclo termina a pós-condição é estabelecida.

5. $\{\, \mathbf{I} \wedge \mathbf{c} \wedge \mathbf{V} = \mathbf{v_0} \,\}\ \mathbf{S}\ \{\, \mathbf{V} < \mathbf{v_0} \,\}$: Por cada iteração, o valor do variante decresce.

Na prova de correcção de um ciclo vamos usar as seguintes anotações:

$\{\, \mathsf{P} \,\}$
$\{\, \mathsf{I} \,\}$
`while (c) {`
    $\{\, \mathsf{I} \wedge \mathsf{c} \wedge 0 \leq \mathsf{V} = \mathsf{v}_0 \,\}$
    S
    $\{\, \mathsf{I} \wedge 0 \leq \mathsf{V} < \mathsf{v}_0 \,\}$
    `}`
$\{\, \mathsf{I} \wedge \neg \mathsf{c} \,\}$
$\{\, \mathsf{Q} \,\}$

Exemplo 10 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: $\mathtt{x} = x_0 \wedge \mathtt{y} = y_0 \geq 0$

Pós-condição: $\mathtt{m} = x_0 * y_0$

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

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 d decresce sempre sem nunca se tornar negativo.
- O valor de m cresce proporcionalmente ao decréscimo de d.

Ajudados por estas observações, podemos formular o seguinte

$$\begin{array}{rcl} I & \doteq & x_0 * \mathtt{d} + m = x_0 * y_0 \\ V & \doteq & \mathtt{d} \end{array}$$

Veremos que o predicado I acima não é suficiente para provar a correcção; mas vamos usá-lo como primeira aproximação.
Usando as regras apresentadas, podemos anotar o algoritmo acima com os seguintes predicados.

$$\begin{array}{l} 1\left[\begin{array}{l}\{\mathtt{x} = x_0 \wedge \mathtt{y} = y_0 \geq 0\} \\ \mathtt{m = 0; \ d = y} \\ \{\mathsf{I}\}\end{array}\right. \\ \mathtt{while\ (d>0)\ \{} \\ \quad 2\left[\begin{array}{l}\{\mathsf{I} \wedge \mathtt{d} > 0 \wedge 0 \leq \mathsf{V} = v_0\} \\ \mathtt{m = m + x;d = d-1} \\ \{\mathsf{I} \wedge 0 \leq \mathsf{V} < v_0\}\end{array}\right. \\ \quad \mathtt{\}} \\ 3\left[\begin{array}{l}\{\mathsf{I} \wedge \neg(\mathtt{d} > 0)\} \\ \{\mathtt{m} = x_0 * y_0\}\end{array}\right. \end{array}$$

Ao qual correspondem as seguintes provas

1. $(\mathtt{x} = x_0 \wedge \mathtt{y} = y_0 \geq 0) \Rightarrow (I[\mathtt{m} \setminus 0, \mathtt{d} \setminus \mathtt{y}])$
2. $(I \wedge d > 0 \wedge 0 \leq V = v_0) \Rightarrow ((I \wedge 0 \leq V < v_0)[\mathtt{m} \setminus \mathtt{m} + \mathtt{x}, \mathtt{d} \setminus \mathtt{d} - 1])$
3. $(I \wedge \neg(d > 0)) \Rightarrow (\mathtt{m} = x_0 * y_0)$

.....

Exemplo 11 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: $\mathtt{a} = a_0 > 0 \wedge \mathtt{b} = b_0 > 0$

Pós-condição: $\mathtt{a} = \mathtt{mdc}\ (a_0, b_0)$

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)

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

Como variante vamos usar a expressão $V \doteq \mid a - b \mid$
Usando as regras apresentadas, podemos anotar o algoritmo acima com os seguintes predicados.

$$
\begin{array}{l}
1\left[\begin{array}{l}\{\,\mathsf{a} = \mathsf{a}_0 > 0 \wedge \mathsf{b} = \mathsf{b}_0 > 0\,\}\\ \{\,\mathsf{mdc(a,b)} = \mathsf{mdc}(\mathsf{a}_0,\mathsf{b}_0) \wedge \mathsf{a} > 0 \wedge \mathsf{b} > 0 \wedge 0 \leq \mid a-b \mid = \mathsf{v}_0\,\}\end{array}\right.\\
\texttt{while (a != b)}\\
\quad \{\,\mathsf{a} \neq \mathsf{b} \wedge \mathsf{mdc(a,b)} = \mathsf{mdc}(\mathsf{a}_0,\mathsf{b}_0) \wedge \mathsf{a} > 0 \wedge \mathsf{b} > 0 \wedge 0 \leq \mid a-b \mid = \mathsf{v}_0\,\}\\
\quad \texttt{if (a < b)}\\
\quad 2\left[\begin{array}{l}\{\,\mathsf{a} \neq \mathsf{b} \wedge \mathsf{mdc(a,b)} = \mathsf{mdc}(\mathsf{a}_0,\mathsf{b}_0) \wedge \mathsf{a} > 0 \wedge \mathsf{b} > 0 \wedge 0 \leq \mid a-b \mid = \mathsf{v}_0 \wedge \mathsf{a} < \mathsf{b}\,\}\\ \texttt{b = b-a;}\\ \{\,\mathsf{mdc(a,b)} = \mathsf{mdc}(\mathsf{a}_0,\mathsf{b}_0) \wedge \mathsf{a} > 0 \wedge \mathsf{b} > 0 \wedge 0 \leq \mid a-b \mid < \mathsf{v}_0\,\}\end{array}\right.\\
\quad \texttt{else}\\
\quad 3\left[\begin{array}{l}\{\,\mathsf{a} \neq \mathsf{b} \wedge \mathsf{mdc(a,b)} = \mathsf{mdc}(\mathsf{a}_0,\mathsf{b}_0) \wedge \mathsf{a} > 0 \wedge \mathsf{b} > 0 \wedge 0 \leq \mid a-b \mid = \mathsf{v}_0 \wedge \mathsf{a} > \mathsf{b}\,\}\\ \texttt{a = a-b;}\\ \{\,\mathsf{mdc(a,b)} = \mathsf{mdc}(\mathsf{a}_0,\mathsf{b}_0) \wedge \mathsf{a} > 0 \wedge \mathsf{b} > 0 \wedge 0 \leq \mid a-b \mid \leq \mathsf{v}_0\,\}\end{array}\right.\\
4\left[\begin{array}{l}\{\,\mathsf{mdc(a,b)} = \mathsf{mdc}(\mathsf{a}_0,\mathsf{b}_0) \wedge \mathsf{a} > 0 \wedge \mathsf{b} > 0 \wedge \mathsf{a} = \mathsf{b}\,\}\\ \{\,\mathsf{a} = \mathsf{mdc}(\mathsf{a}_0,\mathsf{b}_0)\,\}\end{array}\right.
\end{array}
$$

Vejamos então quais as implicações que devemos provar:

1. $(a = a_0 > 0 \wedge b = b_0 > 0)$
   $\Rightarrow (mdc(a,b) = mdc(a_0,b_0) \wedge a > 0 \wedge b > 0 \wedge 0 \leq \mid a-b \mid = v_0)$
2. $(a \neq b \wedge mdc(a,b) = mdc(a_0,b_0) \wedge a > 0 \wedge b > 0 \wedge 0 \leq \mid a-b \mid = v_0 \wedge a < b)$
   $\Rightarrow (mdc(a,b) = mdc(a_0,b_0) \wedge a > 0 \wedge b > 0 \wedge 0 \leq \mid a-b \mid < v_0)[b \setminus b-a]$
3. $(a \neq b \wedge mdc(a,b) = mdc(a_0,b_0) \wedge a > 0 \wedge b > 0 \wedge 0 \leq \mid a-b \mid = v_0 \wedge a > b)$
   $\Rightarrow (mdc(a,b) = mdc(a_0,b_0) \wedge a > 0 \wedge b > 0 \wedge 0 \leq \mid a-b \mid \leq v_0)[a \setminus a-b]$
4. $(mdc(a,b) = mdc(a_0,b_0) \wedge a > 0 \wedge b > 0)$
   $\Rightarrow (a = mdc(a_0,b_0))$

4 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.

Exemplo 12 (Multiplicação inteira)
Relembremos o programa analisado no Exemplo 10. 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 de forma a que ele satisfaça a especificação apresentada nesse exemplo.

```
m = 0; d = y;
while (d>0) {
   ........;
   d = d>>1;
}
```

.....

Exemplo 13 (Divisão inteira)
Um algoritmo naïve de cálculo da divisão inteira de dois números positivos consiste em sucessivamente subtrair o divisor ao dividendo. Assim, o seguinte programa 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 informal feita acima pode ser formalizada por:

Pré-condição: $\mathtt{x} = x_0 \geq 0 \wedge \mathtt{y} = y_0 > 0$

Pós-condição: $0 \leq \mathtt{r} < y_0 \wedge \mathtt{q} * y_0 + \mathtt{r} = x_0$

A prova da correcção pode ser feita usando os seguintes:

$$\begin{array}{rcl} I & \doteq & \mathtt{q} * y_0 + \mathtt{r} = x_0 \wedge 0 \leq \mathtt{y} = y_0 \wedge 0 \leq \mathtt{r} \\ V & \doteq & \mathtt{r} \end{array}$$

.....
E como é que podemos chegar deste à versão eficiente?

```
b = 1; p = y;
while (p <= x) {
   p = p<<1; b = b<<1;
   }
q = 0 ; r = x;
while (r>=y) {
   if (r >= p) { q = q+b; r = r-p;}
   p = p>>1 ; b = b>>1;
   }
```

Exemplo 14 Analisemos agora o problema de procurar um elemento num vector ordenado (cf. Exemplo 6) e cuja especificação apresentada foi

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

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

Uma forma de atingir esta pós-condição é através de um processo iterativo que, começando com o intervalo completo dos índices, o vai restringindo garantindo que o valor a procurar, a existir, se encontra dentro desse intervalo. Dito por outras palavras, vamos usar duas variáveis l e u para representar os limites inferior e superior desse intervalo, e vamos garantir a preservação do seguinte invariante:

$$I \doteq (\forall_{a \leq i \leq b} \,.\, \mathtt{v}[i] = v_i) \wedge ((\exists_{a \leq i \leq b} \,.\, v_i = \mathtt{x}) \Rightarrow (\exists_{\mathtt{l} \leq i \leq u} \,.\, v_i = \mathtt{x}))$$

Uma forma de estabelecer o invariante, i.e., de garantir que o invariante é válido antes da execução da primeira iteração, é inicializar as variáveis l e u com os limites iniciais dos índices. Teremos por isso

$$
\begin{array}{l}
1\left[\begin{array}{l}
\{\,(\forall_{\mathtt{a}\le \mathtt{i}\le \mathtt{b}}\,.\,\mathtt{v[i]} = \mathtt{v_i}) \wedge (\forall_{\mathtt{a}\le \mathtt{i}<\mathtt{b}}\,.\,\mathtt{v_i} \le \mathtt{v_{i+1}})\,\} \\
\mathtt{l = a;\ u = b;} \\
\{\,(\forall_{\mathtt{a}\le \mathtt{i}\le \mathtt{b}}\,.\,\mathtt{v[i]} = \mathtt{v_i}) \wedge ((\exists_{\mathtt{a}\le \mathtt{i}\le \mathtt{b}}\,.\,\mathtt{v_i} = \mathtt{x}) \Rightarrow (\exists_{\mathtt{l}\le \mathtt{i}\le \mathtt{u}}\,.\,\mathtt{v_i} = \mathtt{x}))\,\}
\end{array}\right. \\
\mathtt{while\ (c)\ \{} \\
\quad 2\left[\begin{array}{l}
\{\,(\forall_{\mathtt{a}\le \mathtt{i}\le \mathtt{b}}\,.\,\mathtt{v[i]} = \mathtt{v_i}) \wedge ((\exists_{\mathtt{a}\le \mathtt{i}\le \mathtt{b}}\,.\,\mathtt{v_i} = \mathtt{x}) \Rightarrow (\exists_{\mathtt{l}\le \mathtt{i}\le \mathtt{u}}\,.\,\mathtt{v_i} = \mathtt{x})) \wedge \mathtt{c}\,\} \\
\mathtt{?} \\
\{\,(\forall_{\mathtt{a}\le \mathtt{i}\le \mathtt{b}}\,.\,\mathtt{v[i]} = \mathtt{v_i}) \wedge ((\exists_{\mathtt{a}\le \mathtt{i}\le \mathtt{b}}\,.\,\mathtt{v_i} = \mathtt{x}) \Rightarrow (\exists_{\mathtt{l}\le \mathtt{i}\le \mathtt{u}}\,.\,\mathtt{v_i} = \mathtt{x}))\wedge\,\}
\end{array}\right. \\
\mathtt{\}} \\
3\left[\begin{array}{l}
\{\,(\forall_{\mathtt{a}\le \mathtt{i}\le \mathtt{b}}\,.\,\mathtt{v[i]} = \mathtt{v_i}) \wedge ((\exists_{\mathtt{a}\le \mathtt{i}\le \mathtt{b}}\,.\,\mathtt{v_i} = \mathtt{x}) \Rightarrow (\exists_{\mathtt{l}\le \mathtt{i}\le \mathtt{u}}\,.\,\mathtt{v_i} = \mathtt{x})) \wedge \neg(\mathtt{c})\,\} \\
\{\,(\forall_{\mathtt{a}\le \mathtt{i}\le \mathtt{b}}\,.\,\mathtt{v[i]} = \mathtt{v_i}) \wedge ((\exists_{\mathtt{a}\le \mathtt{i}\le \mathtt{b}}\,.\,\mathtt{v_i} = \mathtt{x}) \Rightarrow \mathtt{v[p]} = \mathtt{x})\,\}
\end{array}\right.
\end{array}
$$

.....

```
p = a; u = b;
while (p<u) {
  if (v[p] < x)
     p = p+1;
  else if (v [p] == x)
          u = p;
       else u = p-1;
  }
```

.....

```
p = a; u = b;
while (p<u) {
  m = (p+u)/2;
  if (v[m] < x)
     p = m+1;
  else if (v [m] == x) {
          u = m;
          p = m;
       }
       else u = m-1;
  }
```

Exemplo 15 (Raíz quadrada)
Um algoritmo para calcular a raiz quadrada de um número consiste em

1. Determinar um intervalo onde garantidamente essa raiz exista
2. Calcular o ponto médio do intervalo e verificar se é maior ou menor do que a raiz pretendida
3. Usar o valor obtido no ponto anterior para actualizar os limites do intervalo
4. Repetir os passos anteriores até que a amplitude do intervalo seja suficientemente pequena para atingir a precisão pretendida.

Comecemos então por especificar o problema em causa:

Pré-condição: $\mathtt{x} = x_0 \ge 0$

Pós-condição: $\mid \sqrt{x} - \mathtt{s} \mid \le \epsilon$

.....

```
l = 0; u = x; d = x; s = (l + u)/2;
while (d > epsilon) {
    if (s*s < x) l = m;
    else u = m;
    d = d/2;
    s = (l + u)/2;
    }
```

5 Exercícios

Exercício 1 Considere a seguinte definição em C:

```
void facts (int f [], int n) {
  int i, j, aux;

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

Mostre que este procedimento satisfaz a seguinte especificação:

Pré-condição: $n = n_0$

Pós-condição: $\forall_{0 \leq i < n_0} . \mathtt{f}[i] = i!$

Exercício 2 Repita a prova acima para a seguinte definição alternativa.

```
void facts (int f [], int n) {
  int i;

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

Exercício 3 Considere a seguinte definição em C:

```
fn = fn1 = 1;
i = 1;
while (i<n) {
  x = fn1 + fn ;
  fn1 = fn ;
  fn = x ;
  i = i+1;
  }
```

Mostre que este procedimento satisfaz a seguinte especificação:

Pré-condição: $n = n_0$

Pós-condição: `fn` $=$ Fibonacci(n_0)

Exercício 4 Repita o exercício anterior para a função que resulta da anterior substituindo o corpo do ciclo `while` por

```
  fn = fn1 + fn ;
  fn1 = fn - fn1 ;
  i = i+1;
```

Exercício 5 Considere a seguinte definição em C:

```
p = 1.0;
i = 0;

while (i<n) {
   p = p * b ;
   i++ ;
   } ;
return (p) ;
}
```

Mostre que este procedimento satisfaz a seguinte especificação:

Pré-condição: $b = b_0 \wedge e = e_0 \wedge e \geq 0$

Pós-condição: $p = c_0^{e_0}$

Resolução O invariante que vamos usar diz que após a execução de cada iteração do ciclo, a variável p tem o valor de uma potência de b. Precisamos ainda de mostrar que o valor da variável b permanece inalterado. Formalmente, $I \doteq p = b_0^i \wedge i \leq n \wedge$ b $= b_0$ Temos por isso de provar:

1. $\{b = b_0 \wedge e = e_0 \wedge e \geq 0\}$p= 1.0; i=0$\{I\}$
2. $\{I \wedge i < n\}$p = p * b ; i = i+1 $\{I\}$
3. $(I \wedge i \geq 0) \Rightarrow p = b_0^{e_0}$

.....

Exercício 6 Split
Seja $P(a[i..j]) \doteq P(a[i]) \wedge \ldots \wedge P(a[j]) = \forall k \in \{i, \ldots, j\}, P(a[k])$.

Prove que a função `split` satisfaz a seguinte especificação.

`Pre` $\doteq 0 \leq l \leq r$
1: `int split(int a[], int l, int r) {` 2: `  int i=l, k=r;` 3: `  while(i<k) {` 4: `  if (a[i]<a[r]) i++;` 5: `  else if (a[k-1]>=a[r] k--;` 6: `  else { swap(a,i,k-1); k=k-1; i=i+1 }` 7: `  }` 8: `}`
`Post` $\doteq a[l..i-1] < a[r] \wedge a[i..r] \geq a[r]$

Note que $P(a[i..j]) = P(a[i]) \wedge P(a[i+1..j] = P(a[i..j-1]) \wedge P(a[j])$

Resolução Vamos utilizar a regra **while-2** para demonstrar a correcção do algoritmo.

1. Definição do invariante de ciclo.

$$I \doteq a[l..i-1] < a[r] \wedge a[k..r] \geq a[r] \wedge i \leq k$$

2. Inicialização: a pré-condição e a execução das atribuições da linha 2 devem implicar a validade do invariante.

$$P \Rightarrow I[k \backslash r, i \backslash l] \Leftrightarrow 0 \leq l \leq r \Rightarrow a[l..l-1] < a[r] \wedge a[r..r] \geq a[r] \wedge l \leq r$$

   Esta implicação é verdadeira uma vez que o *array* $a[l..l-1]$ é vazio.

3. Terminação: no final da execução do ciclo, o invariante e a não verificação da condição de terminação do ciclo devem implicar a pós condição.

$$\begin{array}{l} I \wedge \neg c \Rightarrow \\ a[l..i-1] < a[r] \wedge a[i..r] \geq a[r] \Leftrightarrow \end{array}$$

$$\begin{array}{l} a[l..i-1] < a[r] \wedge a[k..r] \geq a[r] \wedge i \leq k \wedge i \geq k \Rightarrow \\ a[l..i-1] < a[r] \wedge a[i..r] \geq a[r] \end{array}$$

   Esta implicação é verdadeira porque $i \leq k \wedge i \geq k \Rightarrow i = k$.

4. Preservação do invariante de ciclo.
   (a) Vamos tomar o valor $V \doteq k - i + 1$. Sendo S o conjunto de instruções no interior do ciclo, temos que demonstrar que:

$$\{\, I \wedge c \wedge V = v_0 \geq 0 \,\}\ S\ \{\, I \wedge 0 \leq V < v_0 \,\}$$

   (b) Aplicando a regra **if** verificamos que, para demonstrar a preservação do invariante, é necessário considerar os três casos correspondentes aos três ramos da construção `if` no interior do ciclo.

(c) Primeiro caso: $a[i] < a[r]$.

$$
\begin{array}{l}
\{I \wedge c \wedge V = v_0 \geq 0 \wedge a[i] < a[r]\} \\
i = i + 1 \\
\{I \wedge 0 \leq V < v_0\} \\
\Leftrightarrow \\
I \wedge c \wedge V = v_0 \geq 0 \wedge a[i] < a[r] \Rightarrow \\
(I \wedge 0 \leq V < v_0)[i \backslash i + 1] \\
\Leftrightarrow \\
a[l..i-1] < a[r] \wedge a[k..r] \geq a[r] \wedge \\
\wedge i \leq k \wedge i < k \wedge 0 \leq k - i + 1 = v_0 \wedge a[i] < a[r] \Rightarrow \\
a[l..i] < a[r] \wedge a[k..r] \geq a[r] \wedge i + 1 \leq k \wedge 0 \leq k - i < v_0
\end{array}
$$

Esta implicação é verdadeira porque a condição associada ao ramo em questão permite adicionar $a[i]$ à sequência $a[l..i-1]$. Adicionalmente, as relações $i < k$ e $k - i + 1 = v_0$ permitem inferir a parte da implicação relacionada com os índices.

(d) Segundo caso: $a[i] \geq a[r] \wedge a[k-1] \geq a[r]$.

$$
\begin{array}{l}
\{I \wedge c \wedge V = v_0 \geq 0 \wedge a[k-1] \geq a[r]\} \\
k = k - 1 \\
\{I \wedge 0 \leq V < v_0\} \\
\Leftrightarrow \\
I \wedge c \wedge V = v_0 \geq 0 \wedge a[k-1] \geq a[r] \Rightarrow \\
(I \wedge 0 \leq V < v_0)[k \backslash k - 1] \\
\Leftrightarrow \\
a[l..i-1] < a[r] \wedge a[k..r] \geq a[r] \wedge \\
\wedge i \leq k \wedge i < k \wedge 0 \leq k - i + 1 = v_0 \wedge a[k-1] \geq a[r] \Rightarrow \\
a[l..i-1] < a[r] \wedge a[k-1..r] \geq a[r] \wedge i \leq k - 1 \wedge 0 \leq k - i < v_0
\end{array}
$$

Esta implicação é verdadeira porque a condição associada ao ramo em questão permite adicionar $a[k-1]$ à sequência $a[k..r]$. Adicionalmente, as relações $i < k$ e $k - i + 1 = v_0$ permitem inferir a parte da implicação relacionada com os índices.

(e) Terceiro caso: $a[i] \geq a[r] \wedge a[k-1] < a[r]$.

$$
\begin{array}{l}
\{I \wedge c \wedge ind = v_0 \geq 0 \wedge a[i] \geq a[r] \wedge a[k-1] < a[r]\} \\
\texttt{swap}(a, i, k-1); k = k - 1; i = i + 1; \\
\{I \wedge 0 \leq V < v_0\} \\
\Leftrightarrow \\
I \wedge c \wedge V = v_0 \geq 0 \wedge a[i] \geq a[r] \wedge a[k-1] \geq a[r] \Rightarrow \\
(I \wedge 0 \leq V < v_0)[i \backslash i + 1, k \backslash k + 1, \texttt{swap}(a, i, k-1)] \\
\Leftrightarrow \\
a[l..i-1] < a[r] \wedge a[k..r] \geq a[r] \wedge i \leq k \wedge i < k \wedge \\
\wedge 0 \leq k - i + 1 = v_0 \wedge a[i] \geq a[r] \wedge a[k-1] < a[r] \Rightarrow \\
(a[l..i] < a[r] \wedge a[k-1..r] \geq a[r] \wedge i + 1 \leq k - 1 \wedge \\
\wedge 0 \leq k - i - 1 < v_0)[\texttt{swap}(a, i, k-1)] \\
\Leftrightarrow \\
a[l..i-1] < a[r] \wedge a[k..r] \geq a[r] \wedge i \leq k \wedge i < k \wedge \\
\wedge 0 \leq k - i + 1 = v_0 \wedge a[i] \geq a[r] \wedge a[k-1] < a[r] \Rightarrow \\
(a[l..i-1] < a[r] \wedge a[k-1] < a[r] \wedge a[k..r] \geq a[r] \wedge \\
\wedge a[i] \geq a[r] \wedge i \leq k - 2 \wedge 0 \leq k - i - 1 < v_0)
\end{array}
$$

Em que a última equivalência é obtida retirando os indíces $a[i]$ e $a[k-1]$ das sequências em que se encontravam inseridos, adicionando as conjunções correspondentes, e efectuando a troca associada à operação *swap*.

O único aspecto menos claro da demonstração desta implicação está relacionado com o termo $i \leq k-2$. No entanto, esta desigualdade é garantida pelas condições que $i < k$ e que $i \neq k-1$ (que se pode inferir pela condição de entrada neste ramo da construção `if`).

(f) A preservação do invariante de ciclo fica assim demonstrada.

Exercício 7 Bubble Sort

Seja $\texttt{Sorted}(\texttt{a}[\texttt{i..j}]) \doteq \forall k \in \{i, \ldots, j-1\}, a[k] \leq a[k+1]$.

Prove que a função `bubble` satisfaz a seguinte especificação.

$\texttt{Pre} \doteq n = n_0 \wedge 0 \leq n_0$

`int bubble(int a[], int n) {`
`  int i,j;`
`  for(i=0;i<n-1;i++) {`
`  for (j=n-1;j>i;j--) {`
`  if (a[j]<a[j-1])`
`  swap(a,j,j-1);`
`  }`
`  }`
`}`

$\texttt{Post} \doteq \texttt{Sorted}(a[0..n_0])$

Note que $\texttt{Sorted}(a[i..j]) = \texttt{Sorted}(a[i+1..j]) \wedge a[i] \leq a[i+1] = \texttt{Sorted}(a[i..j-1]) \wedge a[j-1] \leq a[j]$.

A Regras de correcção

Fortalecimento das especificações

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

Atribuição

$$\frac{}{\{\, P[x \setminus E] \,\}\ x = E\ \{\, P \,\}}\ \text{(Atrib1)} \qquad \frac{P \Rightarrow (Q[x \setminus E])}{\{\, P \,\}\ x = E\ \{\, Q \,\}}\ \text{(Atrib2)}$$

Sequência

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

Condicionais

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

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

Ciclos

$$\frac{I \wedge c \Rightarrow V \geq 0 \quad \{\, I \wedge c \wedge V = v_0 \,\}\ S\ \{\, I \wedge V < v_0 \,\}}{\{\, I \,\}\ \texttt{while}\ \ c\ S\ \{\, I \wedge \neg c \,\}}\ \text{(while-1)}$$

$$\frac{\{\, I \wedge c \,\}\ S\ \{\, I \,\} \quad I \wedge c \Rightarrow V \geq 0 \quad \{\, I \wedge c \wedge V = v_0 \,\}\ S\ \{\, V < v_0 \,\}}{\{\, I \,\}\ \texttt{while}\ \ c\ S\ \{\, I \wedge \neg c \,\}}\ \text{(while-2)}$$

$$\frac{\begin{array}{llll} P \Rightarrow I & I \wedge c \Rightarrow (V \geq 0) & \{\, I \wedge c \,\} & (I \wedge \neg c) \Rightarrow Q \quad \{\, I \wedge c \wedge V = v_0 \,\} \\ & & \texttt{S} & \texttt{S} \\ & & \{\, I \,\} & \{\, V < v_0 \,\} \end{array}}{\{\, P \,\}\ \texttt{while}\ \ c\ S\ \{\, Q \,\}}\ \text{(while-3)}$$