

# C1-2. Especificação e Correção de Algoritmos

## Correção de um algoritmo

Um algoritmo diz-se **correcto** se para todos os valores dos inputs (variáveis de entrada) ele pára com os valores esperados (i.e. correctos...) dos outputs (variáveis de saída). Neste caso diz-se que ele **resolve** o problema computacional em questão.

Nem sempre a incorrecção é um motivo para a inutilidade de um algoritmo:

- Em certas aplicações basta que um algoritmo funcione correctamente para *alguns dos seus inputs*.
- Em problemas muito difíceis, poderá ser suficiente obter *soluções aproximadas* para o problema.

A análise da correção de um algoritmo pretende determinar se ele é correcto, e em que condições.

A demonstração da correção de um algoritmo cuja estrutura não apresente fluxo de controlo pode ser efectuada por simples inspecção. Exemplo:

```
1  int soma(int a, int b) {  
2      int sum = a+b;  
3      return sum;  
4  }
```

Em alguns casos a correção advém da própria especificação. Considere-se por exemplo uma implementação recursiva da noção de *factorial* de um número. A implementação segue de perto a definição, pelo que a sua correção *algorítmica* é imediata — uma vez que a própria definição é algorítmica, trata-se apenas de verificar se a sua codificação na linguagem de programação escolhida é correcta.

```
1  int factorial(int n) {  
2      int f;  
3      if (n<1) f = 1;  
4      else f = n*factorial(n-1);
```

```
5   return f;  
6 }
```

No entanto, no caso geral esta análise poderá apresentar uma dificuldade muito elevada e deve por isso ser efectuada com algum grau de formalismo, possivelmente recorrendo a uma **lógica de programas**.

## Especificação

### Pré-condições e pós-condições

A análise de correcção dos algoritmos baseia-se na utilização de **asserções**: proposições lógicas sobre o estado actual do programa (o conjunto das suas variáveis). Por exemplo,

- $x > 0$
- $a[i] < a[j]$
- $\forall i. 0 \leq i < n \Rightarrow a[i] < 1000$

Note que em  $a[i] < a[j]$ ,  $i$  é uma variável do programa, e em cada ponto do programa esta fórmula poderá ser verdadeira ou falsa, dependendo dos valores do array  $a$  nos índices  $i$  e  $j$ .

Já na fórmula  $\forall i. 0 \leq i < n \Rightarrow a[i] < 1000$ ,  $i$  é uma variável ligada pelo quantificador, que não corresponde a nenhuma variável do programa (e pode mesmo ser trocada por outra, desde que o seja em todas as ocorrências dentro da fórmula quantificada).

#### Pré-condição:

*É uma propriedade que se assume como verdadeira no estado inicial de execução do programa, i.e., só interessa considerar as execuções do programa que satisfaçam esta condição.*

#### Pós-condição:

*É uma propriedade que se deseja provar verdadeira no estado final de execução do programa.*

## Triplos de Hoare

Um **triplo de Hoare** escreve-se como  $\{P\} C \{Q\}$ , em que

- $C$  é um bloco de instruções cuja correcção se considera
- $P$  é uma *pré-condição* e  $Q$  é uma *pós-condição*

O triplo  $\{P\} C \{Q\}$  é **válido** quando todas as execuções de  $C$  partindo de estados iniciais que satisfaçam  $P$ , caso terminem, resultem num estado final do programa que satisfaz  $Q$ .

Veja-se por exemplo como poderíamos especificar o comportamento de uma função, com corpo

**C**, que calcula o índice de um mínimo de um *array* de comprimento  $N$ :

```
1  int min (int u[], int N) {  
2      // pre: N > 0  
3      C  
4      // pos: 0 ≤ m < N && forall_{0 ≤ i < N} a[m] ≤ a[i]  
5      return m;  
6  }
```

**Observe que:**

- a pós-condição anotada no código é uma versão em ASCII da seguinte fórmula da lógica de primeira ordem:

$$0 \leq m \wedge m < N \wedge \forall i. 0 \leq i \wedge i < N \rightarrow a[m] \leq a[i]$$

- Provar a correcção desta função corresponde a provar a validade do triplo

$$\{N > 0\} \ C \ \{0 \leq m \wedge m < N \wedge \forall i. 0 \leq i \wedge i < N \rightarrow a[m] \leq a[i]\}$$

- a especificação pode ser vista como um *caderno de encargos*: pode ser fornecida a um programador, que terá de escrever um programa que esteja de acordo com ela
- considere a pós-condição alternativa seguinte:

$$0 \leq m \wedge m < N \wedge \forall i. 0 \leq i \wedge i < N \rightarrow a[m] < a[i]$$

que consequências teria a sua utilização em vez da anterior?

## Invariantes de Ciclo

Estabelecer a correcção de algoritmos que incluam ciclos implica considerar qualquer número de iterações; no entanto, não é viável proceder a esta análise por casos, de forma exaustiva.

Recorremos por isso à noção de *invariante de ciclo* — uma propriedade (fórmula de primeira ordem) que se mantém verdadeira em todas as iterações, e que reflecte as transformações de estado efectuadas durante a execução do ciclo.

A ideia é que se o invariante se mantém verdadeiro ao longo da execução, ele será ainda verdadeiro à saída do ciclo, e deverá ser suficientemente forte para permitir provar a pós-condição

desejada para o ciclo.

Raciocinar com um invariante de ciclo corresponde à ideia de prova indutiva no número de iterações de uma execução (que termina) do ciclo. Assim, para provar que uma fórmula  $I$  é um invariante, devemos mostrar:

1. **Inicialização**: que  $I$  é verdade à entrada do ciclo (i.e. antes de se iniciar a primeira iteração) — *caso de base*
2. **Preservação**: assumindo que  $I$  é verdade no início de uma iteração arbitrária (i.e. assumindo que a condição do ciclo é satisfeita), então será satisfeito no final dessa iteração — *caso indutivo*

Sendo  $I$  de facto um invariante, é ainda preciso mostrar que é útil:

3. **Utilidade**: o invariante  $I$ , juntamente com a negação da condição do ciclo (uma vez que terminou a sua execução), implica a verdade da pós-condição

Estas 3 propriedades podem por sua vez ser expressas por triplos de Hoare, envolvendo:

1. para a inicialização, o código que antecede o ciclo
2. para a preservação, o código que constitui o corpo do ciclo
3. para a utilidade, o código que sucede ao ciclo

## Exemplo: Divisão Inteira

Especificação de um programa que calcula a divisão de  $x$  por  $y$ , colocando o quociente em  $q$  e o resto em  $r$ :

```
1  int divide (int x, int y) {  
2      // Pre: x >= 0 && y > 0  
3  
4      // Pos: 0 <= r < y && q*y+r == x  
5      return q  
6  }
```

Resolver este problema consiste em escrever um bloco de código **C** que satisfaça o seguinte

## triplo de Hoare

$$\{x \geq 0 \wedge y > 0\} \text{ C } \{0 \leq r < y \wedge q * y + r = x\}$$

Note que  $x$  e  $y$ , referidas na pré-condição, são **variáveis de entrada** (*inputs*), e  $q$  e  $r$ , referidas apenas na pós-condição, são **variáveis de saída** (*outputs*).

Por exemplo, contando o número de vezes que  $y$  cabe em  $x$  através de um ciclo:

```
1  r = x;  
2  q = 0;  
3  while (y <= r) {  
4      r = r-y;  
5      q = q+1;  
6  }
```

Simulemos uma execução deste programa para  $x = 14$  e  $y = 3$ , escrevendo o valor das variáveis  $r$  e  $q$ , alteradas pelo programa, à **entrada de cada iteração** do ciclo:

$r$	$q$	$q * y + r$
14	0	14
11	1	14
8	2	14
5	3	14
2	4	14

Constata-se que a expressão  $q * y + r$  mantém o seu valor ao longo da execução, à entrada de cada iteração, e que este valor é igual a ao valor de  $x$ .

Por outro lado, o valor de  $r$  é não-negativo ao longo de toda a execução, pelo que escrevemos o seguinte invariante de ciclo:

$$I \equiv 0 \leq r \wedge q * y + r = x$$

Este exemplo encaixa num cenário que é o mais simples possível para o caso de programas que terminam com um ciclo:

| A pós-condição é equivalente à conjunção do invariante e da negação da condição do ciclo.

Vejamos como estabelecer a correcção do programa com este invariante:

- *Inicialização do invariante.* Corresponde ao triplo de Hoare:  
 $\{x \geq 0 \wedge y > 0\} \text{ r} = \mathbf{x}; \text{ q} = 0; \{I\}$

É imediato constatar que é válido, uma vez que  $0 \leq x \wedge 0 * y + x = x$

- *Preservação do invariante.* Corresponde ao triplo de Hoare:  
 $\{I \wedge y \leq r\} \text{ r} = \mathbf{r} - \mathbf{y}; \text{ q} = \mathbf{q} + 1; \{I\}$

Intuitivamente, admitimos que  $I \equiv 0 \leq r \wedge q * y + r = x$  é verdade à entrada de uma iteração qualquer. Temos também  $y \leq r$ , caso contrário não seria executada essa iteração.

Queremos mostrar que  $I$  voltará a ser verdade depois da execução da iteração (arbitrária) em causa.

Ora, os valores de  $r$  e  $q$  no fim da iteração serão respectivamente dados por  $r - y$  e  $q + 1$ . Sendo  $I \equiv 0 \leq r \wedge q * y + r = x$ , queremos então mostrar que a seguinte condição é verdadeira:

$$0 \leq (r - y) \wedge (q + 1) * y + (r - y) = x$$

$$\equiv y \leq r \wedge q * y + r = x$$

E efectivamente, a pré-condição do triplo de Hoare garante isto:

$$(0 \leq r \wedge q * y + r = x) \wedge y \leq r \rightarrow y \leq r \wedge q * y + r = x$$

Ou seja, o invariante é preservado por qualquer iteração do ciclo.

- *Utilidade do invariante.* Corresponde ao triplo de Hoare:

$$\{I \wedge \neg(y \leq r)\} \{ \} \{0 \leq r < y \wedge q * y + r = x\}$$

A utilidade é neste caso trivial, uma vez que o programa termina com o ciclo, não sendo executadas quaisquer instruções subsequentes. Ora, sendo  $I \equiv 0 \leq r \wedge q * y + r = x$ ,  $I \wedge \neg(y \leq r)$  implica (neste caso é mesmo equivalente, mas podia não ser se houvesse código depois do ciclo!) a pós-condição  $0 \leq r < y \wedge q * y + r = x$ .

## Exemplo: Factorial

A função seguinte calcula de forma iterativa (sem recursividade) o factorial de um número natural. A função inclui uma especificação.

```
1  int fact (int n) {
2      // Pre: n >= 0
3      f = 1;
4      i = 1;
5      while (i<=n) {
6          f = f*i;
7          i = i+1;
8      }
9      // Pos: f = n!
10     return f;
```

Mostrar que a função é correcta corresponde a provar a validade do seguinte triplo de Hoare, em que  $\mathbf{F}$  é o corpo da função:

$$\{n \geq 0\} \quad \mathbf{F} \quad \{f = n!\}$$

Para isso temos que descobrir um invariante de ciclo  $I$  apropriado, tal que os triplos de Hoare seguintes sejam válidos:

1. **Inicialização:**  $\{n \geq 0\} \quad \mathbf{f} = 1; \mathbf{i} = 1} \quad \{I\}$
2. **Preservação:**  $\{I \wedge i \leq n\} \quad \mathbf{f} = \mathbf{f} * \mathbf{i}; \mathbf{i} = \mathbf{i} + 1} \quad \{I\}$
3. **Utilidade:**  $\{I \wedge \neg(i \leq n)\} \quad \{ \} \quad \{f = n!\}$

Para caracterizar o invariante apropriado, simulemos uma execução deste ciclo.

```

1  while (i<=n) {
2      f = f*i;
3      i = i+1;
4  }
```

O par de variáveis  $(i, f)$  toma sucessivamente os seguintes valores:

$(1, 1), (2, 1), (3, 2), (4, 6), (5, 24), \dots$

É fácil observar que o valor de  $f$  à entrada de uma iteração corresponde ao factorial de  $i - 1$ . Por outro lado o valor de  $i$  varia entre 0 e  $n + 1$ . Propomos então o invariante:

$$I \equiv f = (i - 1)! \wedge i \leq n + 1$$

É imediato ver que:

- este invariante é bem inicializado, uma vez que  $0! = 1$
- ele é também preservado, uma vez que o corpo do ciclo multiplica  $f$  por  $i$  e incrementa esta variável. Temos que mostrar que

$$\begin{aligned}
 f * i &= (i + 1 - 1)! \wedge i + 1 \leq n + 1 \\
 &\equiv \\
 f * i &= i! \wedge i \leq n
 \end{aligned}$$

Ora, esta condição é implicada por  $I \wedge i \leq n$ :

- Se  $f = (i - 1)!$  então  $f * i = i!$
- Se  $i \leq n + 1$  e  $i \leq n$ , então  $i \leq n$

- Finalmente, falta considerar a utilidade do invariante: à saída do ciclo, sendo a condição  $i \leq n$  falsa, teremos que o valor final de  $i$  será  $i = n + 1$ , uma vez que

$$i \leq n + 1 \wedge \neg(i \leq n) \equiv i = n + 1$$

e o invariante implicará que  $f = n!$  como desejado para satisfazer a pós-condição.

### Exercício

Considere-se o problema de somar todos os elementos de um *array* com  $N$  posições, e a seguinte função que o resolve:

```

1  int sum (int vector[], int n) {
2      // n >= 0
3      result = 0;
4      i = 0;
5      while (i < n) {
6          result = vector[i] + result;
7          i = i+1;
8      }
9      // result == SOMA_{k=0..n-1} vector[k]
10     return result;
11 }
```

Mostrar a correcção desta função corresponde a mostrar a validade do seguinte triplo de Hoare, sendo **Sum** o corpo da função:

$$\{0 \leq n\} \text{ Sum } \{result = \sum_{k=0}^{n-1} vector[k]\}$$

1. Apresente um invariante adequado para o ciclo
2. Escreva os triplos de Hoare correspondentes à inicialização, preservação, e utilidade do invariante, e argumente informalmente que são válidos

Repita depois o exercício para as seguintes versões alternativas do programa (ambas correctas):

```

1  result := 0;
2  i := -1;
3  while (i < n-1) {
4      i := i+1;
5      result := vector[i] + result;
```



```
6 }
```

```
1 result := 0;  
2 i := n;  
3 while (i > 0) {  
4     i := i-1;  
5     result := vector[i] + result;  
6 }
```

## Terminação de Programas: Variantes de Ciclo

A noção de correcção apresentada antes é conhecida por *correcção parcial*, uma vez que não implica provar que o programa termina sempre: basta mostrar que se *terminar* então a pós-condição é satisfeita.

A noção de *correcção total* implica provar adicionalmente que o programa termina sempre. Para isso utilizamos, para cada ciclo do programa, uma técnica baseada numa medida da distância entre o estado actual e o estado de terminação. Chamamos a esta medida um *variante* do ciclo.

Um variante é uma expressão inteira, construída com as variáveis do programa, e deve satisfazer duas condições:

1. Quando a execução entra numa iteração (a condição é verdadeira), o variante é **positivo**
2. As iterações fazem decrescer (estritamente,  $<$  e não  $\leq$ ) o valor do variante

No exemplo no algoritmo da divisão, é fácil ver que a diferença entre  $y$  e  $r$  vai diminuindo em todas as iterações porque o valor de  $y$  não é alterado e o de  $r$  decresce sempre, logo a expressão  $r-y$  é um candidato a variante.

No entanto, não é verdade que  $r - y > 0$  sempre que é executada uma iteração, uma vez que a condição  $y \leq r$  só garante  $r-y \geq 0$ .

Escolhemos então o variante  $r-y+1$ , esse sim sempre positivo à entrada das iterações.

```

1  r = x;
2  q = 0;
3  while (y <= r) {
4      r = r-y;
5      q = q+1;
6  }

```

Relembremos agora o ciclo do factorial:

```

1  while (i<=n) {
2      f = f*i;
3      i = i+1;
4  }

```

é em tudo semelhante: a expressão  $n-i+1$  é um variante deste ciclo, uma vez que:

- $n$  mantém-se constante e  $i$  é incrementado em todas as iterações, o que significa que o valor de  $n-i+1$  decresce em todas as iterações
- é garantido pelo condição  $i \leq n$  que  $n - i + 1 > 0$  à entrada de todas as iterações

Já para o ciclo da função `sum` :

```

1  while (i < n) {
2      result = vector[i] + result;
3      i = i+1;
4  }

```

podemos escolher como variante  $n-i$ .

Note-se que estes exemplos são muito simples uma vez que os ciclos são controlados por variáveis cujos valores são conhecidos estaticamente. Nem sempre é assim, por exemplo o ciclo seguinte terminará quando a fila de espera ficar vazia:

```

1  queue = ... ;    // inicialização com um conjunto de elementos
2  while (!is_empty(queue)) {
3      ...
4      x = dequeue(&q);
5      ...
6  }

```

Um variante apropriado poderá ser neste caso o *comprimento da fila* `queue`: sempre positivo quando é executada uma iteração, e diminui em todas as iterações.

Um outro exemplo será o seguinte, em que a terminação depende da forma como o bloco C alterar o valor de `flag`.

```

1  flag = true;
2  while (flag) {
3      C
4  }
```

Em casos como este pode ser muito desafiante obter um variante: o padrão de iteração pode ser complexo, e a garantia de terminação pode resultar de propriedades não evidentes dos dados do programa.

## Soluções

$$I: i \leq n \wedge result = \sum_{k=0}^{i-1} vector[k]$$

- Inic:  $\{0 \leq n\} \text{ result} = 0; i = 0 \{I\}$
- Preserv:  $\{I \wedge i < n\} \text{ result} = vector[i] + result; i = i+1 \{I\}$
- Util:  $I \wedge i \geq n \rightarrow result = \sum_{k=0}^{n-1} vector[k]$

$$I: i \leq n - 1 \wedge result = \sum_{k=0}^i vector[k]$$

- Inic:  $\{0 \leq n\} \text{ result} := 0; i := -1 \{I\}$
- Preserv:  $\{I \wedge i < n - 1\} i := i+1; result := vector[i] + result \{I\}$
- Util:  $I \wedge i \geq n - 1 \rightarrow result = \sum_{k=0}^{n-1} vector[k]$

$$I: i \geq 0 \wedge result = \sum_{k=i}^{n-1} vector[k]$$

- Inic:  $\{0 \leq n\}$  result := 0; i:=n  $\{I\}$
- Preserv:  $\{I \wedge i > 0\}$  i := i-1; result := vector[i] + result  $\{I\}$
- Util:  $I \wedge i \leq 0 \rightarrow result = \sum_{k=0}^{n-1} vector[k]$