C1. Especificação e Correcção de Algoritmos

Correcção 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 correcção de um algoritmo pretende determinar se ele é correcto, e em que condições.

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

```
int soma(int a, int b) {
  int sum = a+b;
  return sum;
}
```

Em alguns casos a correcção 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 correcção *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.

```
int factorial(int n) {
  int f;
  if (n<1) f = 1;
  else f = n*factorial(n-1);
  return f;
}
```

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:

```
int min (int u[], int N) {
   // pre: N > 0
   C
   // pos: 0<=m<N && forall_{0<=i<N} a[m] <= a[i]
  return m;
}
```

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?

Correcção de Triplos de Hoare

Consideremos o triplo seguinte:

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

Intuitivamente percebe-se que é válido (i.e. o programa que consiste na instrução $x = x + 1$ é correcto face à especificação que consiste na pré-condição $\{x = y\}$ e na pós-condição $\{x = y + 1\}$.

É no entanto desejável ter uma justificação mais rigorosa desta validade. Para isso esquecemos temporariamente a pré-condição e perguntamo-nos:

> *Por forma a que a pós-condição seja verdadeira depois da execução da instrução, que condição tem necessariamente que ser verdadeira **antes** dessa execução?*

Para responder a esta questão basta propagarmos a pós-condição para trás, utilizando a instrução de atribuição como uma **substituição**. Assim, substituindo x por $x + 1$ na condição $\{x = y + 1\}$ obtemos $\{x + 1 = y + 1\}$, condição que terá que ser verdadeira antes da execução.

Ora, efectivamente, se a pré-condição for verdadeira, também a condição que calculámos o será, uma vez que $x = y \Rightarrow x + 1 = y + 1$. Isto permite-nos provar a validade do triplo de Hoare.

Tony Hoare formalizou esta forma de raciocínio através de um sistema dedutivo, hoje conhecido por **Lógica de Hoare**.

Este princípio de prova aplica-se também na presença de *sequências* de instruções, bem como de *condicionais*.

Exercício

Investigue como poderá ser provada a validade dos seguintes triplos de Hoare:

1. $\{x = y\}\ \ x = x + 1\ ;\ x = x + 1\ \{x = y + 2\}$

2. $\{\ x = y\ \}$
   $\textbf{if}\ (z > 0)\ \ x = x + 1\ \ \textbf{else}\ \ y = y + 1$
   $\{\ (z > 0 \wedge x = y + 1) \vee (z \leq 0 \wedge y = x + 1)\ \}$