Notas das Aulas Teóricas
Correcção de Algoritmos

Métodos de Programação II
(5304O4 e 7004N6)

Ano Lectivo de 2004–2005

Uma forma de escrever uma especificação de um programa é através de 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.

Dizemos então que, um dado programa S satisfaz uma especificação (P, Q) (em que P é a pré-condição e Q a pós-condição) quando, assumindo que P é verdadeiro, podemos garantir que

- o programa S termina e que,
- após a conclusão de S, Q é verdadeiro.

É costume escrever $\{\, P \,\}\ S\ \{\, Q \,\}$ para dizer que S satisfaz a especificação (P, Q).

Desta definição de satisfação resultam imediatamente duas regras

Fortalecimento da pré-condição

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

Enfraquecimento da pós-condição

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

Note-se que dizer que $A \Rightarrow B$ corresponde a dizer que A é mais restritivo (ou mais forte) do que B. Daí que estas regras se possam ler como:

- Se um programa funciona em determinadas condições iniciais, ele continuará a funcionar se em condições mais restritas.
- Se um programa garante que alguma propriedade é válida, garantirá que qualquer condição menos restritiva também é válida.

A instrução fundamental de qualquer linguagem imperativa é a **atribuição**. Em C, escreve-se `x = E` para significar que atribuimos à variável `x` o valor da expressão `E`. Este significado é traduzido pela seguinte regra.

Atribuição–1

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

Em que $E[x \setminus F]$ corresponde a, na expressão E, substituir todas as ocorrências de x pela expressão E.
À expressão $P[x \setminus E]$ é costume chamar a *pré-condição mais fraca* (**weakest pre-condition**) que garante que P é válido após a execução da atribuição $x = E$

A conjunção desta regra com a do fortalecimento da pré-condição permite-nos descrever o significado de uma atribuição de outra forma:

Atribuição–2

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

Esta regra é operacionalmente mais útil do que a anterior pois dá-nos um meio directo de provar que uma dada atribuição satisfaz uma particular especificação: para provar que $\{\, P \,\}\ x = E\ \{\, Q \,\}$ basta mostrar que $P \Rightarrow (Q[x \setminus E])$.

A correcção da composição sequencial de dois programas é feita através da descoberta de um predicado intermédio que actua como pós-condição do primeiro e pré-condição do segundo. Sob a forma de uma regra podemos escrever

Sequência

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

Exemplo 1

Para provarmos que o programa `x = x + y ; y = x - y ; x = x - y` faz a troca dos valores das variáveis `x` e `y`, vamos usar os seguintes predicados

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

A aplicação da regra da sequenciação conduz-nos a descobrir predicados A_1 e A_2 tais que são válidos os seguintes:

- $\{ x = x_0 \ \wedge y = y_0 \}$ `x = x + y` $\{ A_1 \}$
- $\{ A_1 \}$ `y = x - y` $\{ A_2 \}$
- $\{ A_2 \}$ `x = x - y` $\{ x = y_0 \ \wedge y = x_0 \}$

Uma estratégia para descobrir estes predicados é usar a regra da atribuição (Atrib1) para em primeiro lugar determinar A_2 e depois A_1.

1. Basta escolher A_2 como $(x = y_0 \ \wedge y = x_0)[x \setminus x - y]$ o que corresponde a

$$A_2 \doteq x - y = y_0 \wedge y = x_0$$

2. Basta escolher A_1 como $(x - y = y_0 \wedge y = x_0)[y \setminus x - y]$ o que corresponde a

$$A_1 \doteq x - (x - y) = y_0 \wedge x - y = x_0$$

3. Falta-nos apenas provar que

$$(x = x_0 \ \wedge y = y_0) \Rightarrow A_1[x \setminus x + y]$$

   Note-se que o consequente desta implicação pode ser simplificado da seguinte forma:

$$\begin{array}{ll} & A_1[x \setminus x + y] \\ \Leftrightarrow & (x - (x - y) = y_0 \wedge x - y = x_0)[x \setminus x + y] \\ \Leftrightarrow & (y = y_0 \wedge x - y = x_0)[x \setminus x + y] \\ \Leftrightarrow & y = y_0 \wedge (x + y) - y = x_0 \\ \Leftrightarrow & y = y_0 \wedge x = x_0 \end{array}$$

   que não é mais do que o seu antecendente.

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

Condicional

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

Exemplo 2

Para provarmos que o programa `if (x > y) a = x else a = y` coloca em `a` o máximo dos valores de `x` e `y`, vamos usar os seguintes predicados

- **pré-condição:** $x = x_0 \ \wedge y = y_0$
- **pós-condição:** $a = max(x_0, y_0)$

A prova da correcção faz-se em dois passos:

1. Provando que $\{\, x = x_0 \wedge y = y_0 \wedge x > y \,\}$ `a = x` $\{\, a = max(x_0, y_0) \,\}$, ou seja que

$$\begin{array}{rl} & (x = x_0 \wedge y = y_0 \wedge x > y) \Rightarrow (a = max(x_0, y_0))[a \setminus x] \\ \Leftrightarrow & (x = x_0 \wedge y = y_0 \wedge x > y) \Rightarrow (x = max(x_0, y_0)) \end{array}$$

2. Provando que $\{\, x = x_0 \wedge y = y_0 \wedge x \leq y \,\}$ `a = y` $\{\, a = max(x_0, y_0) \,\}$, ou seja que

$$\begin{array}{rl} & (x = x_0 \wedge y = y_0 \wedge x \leq y) \Rightarrow (a = max(x_0, y_0))[a \setminus y] \\ \Leftrightarrow & (x = x_0 \wedge y = y_0 \wedge x \leq y) \Rightarrow (y = max(x_0, y_0)) \end{array}$$

que são consequência da definição de max.

A correcção de programas que envolvam ciclos é descrita pela seguinte regra.

Ciclo-1

$$\frac{\{\, I \wedge c \wedge ind = i_0 \geq 0 \,\}\ S\ \{\, I \wedge 0 \leq ind < i_0 \,\}}{\{\, I \,\}\ \texttt{while}\ c\ S\ \{\, I \wedge \neg c \,\}} \quad \text{(while-1)}$$

Ao predicado I que é mencionado nesta regra é habitual chamar **invariante do ciclo**. Aquilo que esta regra diz é que:

- partindo do princípio que o invariante é válido antes da execução do ciclo,
- sempre que o invariante do ciclo é válido antes de cada iteração, também o é depois dessa iteração,
- por cada iteração do ciclo há uma expressão inteira e não negativa (ind) que decresce,

então, o ciclo termina e quando termina podemos garantir que o invariante se verifica e que a condição do ciclo é falsa.

Esta regra, conjugada com as de enfraquecimento (da pós-condição) e de fortalecimento (da pré-condição), dá origem à seguinte regra, mais operacional.

Ciclo-2

$$\frac{P \Rightarrow I \quad \{\, I \wedge c \wedge ind = i_0 \geq 0 \,\}\ S\ \{\, I \wedge 0 \leq ind < i_0 \,\} \quad (I \wedge \neg c) \Rightarrow Q}{\{\, P \,\}\ \texttt{while}\ c\ S\ \{\, Q \,\}} \quad \text{(while-2)}$$

Assim, para mostrar que um ciclo satisfaz uma especificação (P, Q) precisamos de encontrar um invariante I e uma medida de terminação ind (I é um predicado, enquanto que ind é uma expressão inteira e não negativa) tais que:

1. A pré-condição garante que o invariante começa por ser verdadeiro ($P \Rightarrow I$)
2. Cada iteração do ciclo preserva a validade do invariante, além de diminuir o valor da medida de terminação;
$$\{\, I \wedge c \wedge ind = i_0 \geq 0 \,\}\ S\ \{\, I \wedge 0 \leq ind < i_0 \,\}$$
3. Quando o ciclo termina (e neste caso a condição de controlo do ciclo é falsa e o invariante continua válido), a pós-condição é assegurada
$$(I \wedge \neg c) \Rightarrow Q$$

Exemplo 3

O programa seguinte calcula o máximo divisor comum entre dois números inteiros positivos (mdc).

$$\{\, a = a_0 > 0 \wedge b = b_0 > 0 \,\}$$

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

$$\{\, a = mdc(a_0, b_0) \,\}$$

A demonstração da correcção baseia-se no seguinte teorema (de Euclides)

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

Vamos usar como invariante o seguinte predicado

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

Apesar de mudarmos o valor de a e b estamos a mudá-los mantendo o valor do máximo divisor comum.
Como medida de terminação vamos usar (o valor absoluto) da diferença entre a e b

$$ind \doteq |a - b|$$

Usando a regra `(while2)` temos que mostrar que:

1. $(a = a_0 > 0 \wedge b = b_0 > 0) \Rightarrow ((mdc(a, b) = mdc(a_0, b_0)) \wedge a > 0 \wedge b > 0)$

   O que é trivialmente verdadeiro.

2. $\{\, (mdc(a, b) = mdc(a_0, b_0)) \wedge a > 0 \wedge b > 0 \wedge a \neq b \wedge |a - b| = i_0 > 0 \,\}$

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

   $\{\, (mdc(a, b) = mdc(a_0, b_0)) \wedge a > 0 \wedge b > 0 \wedge 0 \leq |a - b| < i_0 \,\}$

   Podemos então usar a regra `(if)` e temos

   (a) $\{\, (mdc(a, b) = mdc(a_0, b_0)) \wedge a > 0 \wedge b > 0 \wedge a \neq b \wedge |a - b| = i_0 > 0 \wedge a < b \,\}$

   ```
   b = b-a
   ```

   $\{\, (mdc(a, b) = mdc(a_0, b_0)) \wedge a > 0 \wedge b > 0 \wedge 0 \leq |a - b| < i_0 \,\}$

   (b) $\{\, (mdc(a, b) = mdc(a_0, b_0)) \wedge a > 0 \wedge b > 0 \wedge a \neq b \wedge |a - b| = i_0 > 0 \wedge a \geq b \,\}$

   ```
   a = a-b
   ```

   $\{\, (mdc(a, b) = mdc(a_0, b_0)) \wedge a > 0 \wedge b > 0 \wedge 0 \leq |a - b| < i_0 \,\}$

   Ambas as implicações resultantes destas expressões são consequência do teorema de Euclides.

3. Finalmente devemos mostrar que

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

   O que é verdadeiro, sabendo ainda que $mdc(x, x) = x$.

Da mesma forma que usámos as regras do ciclo, do enfraquecimento e fortalecimento para obter esta última regra, podemos agora conjugá-la com a regra de sequenciação para definirmos a seguinte.

Ciclo-3

$$\frac{\begin{array}{lll}\{\,P\,\} & \{\,I \wedge c \wedge ind = i_0 \geq 0\,\} & (I \wedge \neg c) \Rightarrow Q \\ S_1 & S_2 & \\ \{\,I\,\} & \{\,I \wedge 0 \leq ind < i_0\,\} & \end{array}}{\{\,P\,\}\ S_1\,;\ \texttt{while}\ c\ S_2\ \{\,Q\,\}} \quad \text{(while-3)}$$

Esta regra é particularmente útil se quisermos provar a correcção de um ciclo `for` em C. Relembre-se que em C, o programa `for (X;Y;Z) W;` é equivalente a `X ; while Y { W ; Z }`.

For

$$\frac{\begin{array}{lll}\{\,P\,\} & \{\,I \wedge Y \wedge ind = i_0 \geq 0\,\} & (I \wedge \neg Y) \Rightarrow Q \\ X & W; Z & \\ \{\,I\,\} & \{\,I \wedge 0 \leq ind < i_0\,\} & \end{array}}{\{\,P\,\}\ \texttt{for}\ (X;Y;Z)\ W\ \{\,Q\,\}} \quad \text{(for)}$$

Exemplo 4

Considere-se o seguinte programa que calcula a potência (inteira) de um dado número.

```
r = 1; i = 0;
while (i < e) {
   r = r * b ;
   i = i + 1 ;
   }
```

Vamos provar que este programa satisfaz a seguinte especificação:

- **pré-condição:** $b = b_0 \wedge e = e_0 \geq 0$
- **pós-condição:** $r = b_0^{e_0}$

Para isso, e como temos um ciclo, vamos ter que definir:

- um **invariante** do ciclo. $I \doteq b = b_0 \wedge e = e_0 \wedge r = b_0^i \wedge i \leq e$
- uma **medida de terminação**. $ind \doteq e - i$

Usando a regra (`while-3`) acima, temos que mostrar que:

1. $\{\, b = b_0 \wedge e = e_0 \geq 0\,\}\ r = 1; i = 0\ \{\, b = b_0 \wedge e = e_0 \wedge r = b_0^i \wedge i \leq e\,\}$

   Isto corresponde a mostrar que

   $$\begin{array}{ll} & (b = b_0 \wedge e = e_0 \geq 0) \Rightarrow (b = b_0 \wedge e = e_0 \wedge r = b_0^i \wedge i \leq e)[r \setminus 1, i \setminus 0] \\ \Leftrightarrow & (b = b_0 \wedge e = e_0 \geq 0) \Rightarrow (b = b_0 \wedge e = e_0 \wedge 1 = b_0^0 \wedge 0 \leq e) \end{array}$$

2. $\{\, b = b_0 \wedge e = e_0 \wedge r = b_0^i \wedge i \leq e \wedge (i < e) \wedge (e - i) = i_0 \geq 0\ \}$

   `r = r*b; i = i+1`

   $\{\, b = b_0 \wedge e = e_0 \wedge r = b_0^i \wedge i \leq e \wedge 0 \leq e - i < i_0\,\}$

   que é equivalente a mostrar que

   $$\begin{array}{ll} & b = b_0 \wedge e = e_0 \wedge r = b_0^i \wedge i \leq e \wedge (i < e) \wedge (e - i) = i_0 \geq 0 \\ & \quad \Rightarrow (b = b_0 \wedge e = e_0 \wedge r = b_0^i \wedge i \leq e \wedge 0 \leq e - i < i_0)[r \setminus r * b, i \setminus i + 1] \\ \Leftrightarrow & b = b_0 \wedge e = e_0 \wedge r = b_0^i \wedge i \leq e \wedge (i < e) \wedge (e - i) = i_0 \geq 0 \\ & \quad \Rightarrow (b = b_0 \wedge e = e_0 \wedge r * b = b_0^{i+1} \wedge (i + 1) \leq e \wedge 0 \leq e - i - 1 < i_0 \end{array}$$

3. $(b = b_0 \wedge e = e_0 \wedge r = b_0^i \wedge i \leq e \wedge i \geq e) \Rightarrow r = b_0^{e_0}$

Exemplo 5

Mostre que os seguintes programas satisfazem a especificação.

- **pré-condição:** $n = n_0 \geq 0$
- **pós-condição:** $n2 = n_0^2$

1. ```
   n2 = 0 ; i = 0 ;
   while (i < n) {
     n2 = n2 + (2 * i) + 1 ;
     i = i+1;
     }
   ```

2. ```
   n2 = 0 ;
   while (n > 0) {
     n2 = n2 + (2 * n) + 1;
     n = n-1;
     }
   ```