Algoritmos e Complexidade

Introdução à Análise de Correcção de Algoritmos

José Bernardo Barros
Departamento de Informática
Universidade do Minho

Contents

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 em C:

```
int fact (int n) {
  if (n==0) return 1;
  else return (n*fact(n-1));
}
```

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

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

- Na programação **declarativa** 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.

- Na programação **imperativa** 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/~mjcg/`)

2 Estado e Especificações

Uma das características mais importantes das linguagens imperativas é a existência de **estado**. O estado de um programa define-se 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 $[\![\mathtt{v}]\!]_s$ 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 $[\![\mathrm{x}]\!]_s = 3$ e $[\![\mathrm{y}]\!]_s = 4$ então

- $[\![\mathtt{x + (y*x)}]\!]_s = [\![\mathrm{x}]\!]_s + ([\![\mathrm{y}]\!]_s * [\![\mathrm{x}]\!]_s) = 15$

- $[\![\mathtt{x+1 == y}]\!]_s = \mathtt{True}$

A função de cálculo do valor de uma expressão num estado pode ser usada para calcular o valor de um predicado num dado estado, e consequentemente caracterizar os estados de um programa imperativo.
Dado um estado S e um predicado P cujas variáveis livres pertencem às variáveis do estado S, dizemos que um esse predicado é válido no estado S sse é válido o predicado $[\![\mathtt{P}]\!]_S$.

Exemplo 1 Seja S o estado em que as variáveis x, y e z têm os valores 10, 2 e 12, respectivamente. Nesse estado **são válidos** os seguintes predicados.

- `x+z>y && x<z`
- `x+y == z`

Por outro lado, **não é válido** o predicado `x > y*z`

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 establecem 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 analizar alguns exemplos de especificações de problemas simples e bem conhecidos.

Exemplo 2 (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: $\mathtt{x} == y_0 \wedge \mathtt{y} == x_0$

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 3 (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: $\texttt{x} == x_0 \wedge \texttt{y} == y_0 \geq 0$
Pós-condição: $\texttt{m} == x_0 * y_0$

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 4 (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: $\texttt{x} == x_0 > 0 \wedge \texttt{y} == y_0 \geq 0$
Pós-condição: $0 \leq m < y_0 \wedge \exists_{d \geq 0}\, d * y_0 + m == x_0$

Exemplo 5 (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: $\texttt{x} == x_0 > 0 \wedge \texttt{y} == y_0 \geq 0$
Pós-condição: $d \geq 0 \wedge \exists_{0 \leq m < y_0}\, d * y_0 + m == x_0$

Exemplo 6 (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: $\texttt{x} == x_0 > 0 \wedge \texttt{y} == y_0 \geq 0$
Pós-condição: $0 \leq m < y_0 \wedge d \geq 0 \wedge d * y_0 + m == x_0$

Exemplo 7 (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: $(\forall_{a \leq i \leq b} \,.\, \texttt{v}[i] == v_i) \wedge (\forall_{a \leq i < b} \,.\, v_i \leq v_{i+1})$
Pós-condição: $(\forall_{a \leq i \leq b} \,.\, \texttt{v}[i] == v_i) \wedge ((\exists_{a \leq i \leq b} \,.\, v_i == \texttt{x}) \Rightarrow \texttt{v[p]} = \texttt{x})$

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: $\mid r * r - x_0 \mid < e_0$

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

3 Programas

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> }
```

4 Correcção 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 8 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 $[\![\mathtt{x}]\!]_A = 3$ e $[\![\mathtt{y}]\!]_A = -5$.
Note-se que neste estado a pré-condição é válida:

$$[\![\mathtt{x\ >\ 0}]\!]_A \Leftrightarrow (3 > 0) \Leftrightarrow True$$

Partindo desse estado, atingimos um estado B em que $[\![\mathtt{x}]\!]_B = -2$ e $[\![\mathtt{y}]\!]_B = -5$. Ora neste estado a pós-condição não é válida:

$$[\![\mathtt{x\ >\ 1}]\!]_B \Leftrightarrow (-2 > 1) \Leftrightarrow False$$

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 establecer um conjunto de regras de prova que nos permitam atingir tal objectivo
Para cada um dos construtores de programas vistos na secção 3 vamos apresentar regras de prova da correcção de programas que envolvam essas construções.

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

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

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.

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

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

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

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

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

4.2 Atribuição

A operação fundamental de qualquer linguagem de programação imperativa é a atribuição do valor de uma expressão 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 pelos seguintes passos.

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. Ou seja, que qualquer propriedade sobre o valor final de `x` também deve ser válida sobre o valor da expressão `e` no estado inicial.

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

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

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

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

Exemplo 9 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 2 da página 3.
Usando a correcção da sequenciação, temos de encontrar predicados R_1 e R_2 tais que:

$$\begin{array}{l}
\{\, \mathsf{x} == \mathsf{x}_0 \wedge \mathsf{y} == \mathsf{y}_0 \,\} \\
\texttt{x := x + y ;} \\
\{\, \mathsf{R}_2 \,\} \\
\texttt{y := x - y ;} \\
\{\, \mathsf{R}_1 \,\} \\
\texttt{x := x - y ;} \\
\{\, \mathsf{x} == \mathsf{y}_0 \wedge \mathsf{y} == \mathsf{x}_0 \,\}
\end{array}$$

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

- $\begin{array}{rl} R_1 = & (\mathrm{x} == y_0 \wedge \mathrm{y} == x_0)[\mathrm{x} \setminus \texttt{x - y}] \\ = & \texttt{x - y} == y_0 \wedge \mathrm{y} == x_0 \end{array}$

- $\begin{array}{rl} R_2 = & R_1[\mathrm{y} \setminus \texttt{x - y}] \\ = & (\texttt{x - y} == y_0 \wedge \mathrm{y} == x_0)[\mathrm{y} \setminus \texttt{x - y}] \\ = & \texttt{x - (x - y)} == y_0 \wedge \texttt{x - y} == x_0 \\ = & \mathrm{y} == y_0 \wedge \texttt{x - y} == x_0 \end{array}$

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

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

Comecemos por simplificar o consequente desta implicação.

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

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

Exemplo 10 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:

$$\begin{array}{l}
\{\, \mathsf{x} == \mathsf{x}_0 \wedge \mathsf{y} == \mathsf{y}_0 \,\} \\
\texttt{z := x} \\
\{\, \mathsf{R}_2 \,\} \\
\texttt{x := y} \\
\{\, \mathsf{R}_1 \,\} \\
\texttt{y := z} \\
\{\, \mathsf{x} == \mathsf{y}_0 \wedge \mathsf{y} == \mathsf{x}_0 \,\}
\end{array}$$

Donde vem:

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

Mias uma vez, para completarmos a prova vamos usar a segunda das regras apresentadas para a atribuição. Temos então de provar que:

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

Simplifiquemos o consequente desta implicação.

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

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.

$$\frac{P \Rightarrow ((Q[\mathtt{x}_n \setminus E_n])[\mathtt{x}_{n-1} \setminus E_{n-1}]) \cdots)[\mathtt{x}_1 \setminus E_1]}{\{\, R \,\}\ \mathtt{x}_1 = E_1; \cdots ; \mathtt{x}_n = E_n \ \{\, Q \,\}} \quad \text{(SeqAtr)}$$

4.4 Condicionais

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

Condicional

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

Que traduz o significado intuitivo da construção $\mathtt{if}\ \ c\ \{S_1\}\ \mathtt{else}\ \ \{S_2\}$: 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 11 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.

$$
\begin{array}{l}
\{\, \mathsf{x} = \mathsf{x_0} \wedge \mathsf{y} = \mathsf{y_0} \,\} \\
\texttt{if (x > y)} \\
\quad 1 \left[\begin{array}{l} \{\, \mathsf{x} > \mathsf{y} \wedge \mathsf{x} == \mathsf{x_0} \wedge \mathsf{y} == \mathsf{y_0} \,\} \\ \{\ \texttt{M := x;}\ \} \\ \{\, \mathtt{M} == \mathtt{max}\ (\mathsf{x_0}, \mathsf{y_0}) \,\} \end{array} \right. \\
\quad \texttt{else} \\
\quad 2 \left[\begin{array}{l} \{\, \mathsf{x} \leq \mathsf{y} \wedge \mathsf{x} == \mathsf{x_0} \wedge \mathsf{y} == \mathsf{y_0} \,\} \\ \texttt{M := y;}\ \} \\ \{\, \mathtt{M} == \mathtt{max}\ (\mathsf{x_0}, \mathsf{y_0}) \,\} \end{array} \right.
\end{array}
$$

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

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

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

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

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

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

Ciclo-3

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

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 12 Consideremos o seguinte programa que multiplica dois números inteiros por somas sucessivas:

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

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 *experimentar* 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

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

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

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, establecendo então a pós-condição em causa.