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.

- 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.

Na programação imperativa nem sempre é fácil determinar a 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 $\mathtt{v}$ for uma das suas variáveis, é costume representar-se por $[\![\mathtt{v}]\!]_s$ o valor de $\mathtt{v}$ 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 $[\![\mathtt{x}]\!]_s = 3$ e $[\![\mathtt{y}]\!]_s = 4$ então

- $[\![\texttt{x + (y*x)}]\!]_s = [\![\mathtt{x}]\!]_s + ([\![\mathtt{y}]\!]_s * [\![\mathtt{x}]\!]_s) = 15$
- $[\![\texttt{x+1 == y}]\!]_s = \texttt{True}$

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 $[\![\mathrm{P}]\!]_S$.

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 estabelecem 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* escrever 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. Uma forma de lidar com este requisito consiste em, 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 (como pré-condição) 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** (ou `ghost variables` 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:	$\mathtt{x} == x_0 \wedge \mathtt{y} == y_0 \geq 0$
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 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:	$\mathtt{x} == x_0 > 0 \wedge \mathtt{y} == y_0 \geq 0$
Pós-condição:	$0 \leq m < y_0 \wedge \exists_{d \geq 0}\, d * y_0 + m == x_0$

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

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

Exemplo 7 (divmod) A especificação seguinte estabelece os requisitos de um programa que coloca em `r` a raiz quadrada de `x`.

Pré-condição:	$\mathtt{x} == x_0 \geq 0$
Pós-condição:	$\mathtt{r} * \mathtt{r} == x_0$

Exemplo 8 (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: $(\forall_{a\leq i\leq b}\,.\,\texttt{v[}i\texttt{]} == v_i) \wedge (\forall_{a\leq i<b}\,.\,v_i \leq v_{i+1})$
Pós-condição: $(\forall_{a\leq i\leq b}\,.\,\texttt{v[}i\texttt{]} == v_i) \wedge ((\exists_{a\leq i\leq b}\,.\,v_i == \texttt{x}) \Rightarrow \texttt{v[p]} = \texttt{x})$

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

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

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

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

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. $\{\, \mathtt{i} \mathrel{!=} \mathtt{j} \,\}\; \mathit{if}\ (i > j)\ \mathit{then}\ m := i - j\ \mathit{else}\ m := j - i \;\{\, \mathtt{m} > 0 \,\}$
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. $\{\, \mathtt{True} \,\}\; \mathit{if}\ (i < j)\ \mathit{then}\ \mathit{min} := i\ \mathit{else}\ \mathit{min} := j \;\{\, \mathtt{min} \leq \mathtt{i} \wedge \mathtt{min} \leq \mathtt{j} \,\}$
6. $\{\, \mathtt{i} > 0 \wedge \mathtt{j} > 0 \,\}\; \mathit{if}\ (i < j)\ \mathit{then}\ \mathit{min} := i\ \mathit{else}\ \mathit{min} := j \;\{\, \mathtt{min} > 0 \,\}$

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.

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

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

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

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

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

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` é 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

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

Quando escrevemos $P[x \setminus E]$ significamos *substituir todas as ocorrências (livres) da variável* x *pela expressão* E. 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 primeiro programa é executado a partir do estado inicial.
- o estado final é atingido após a execução do segundo programa
- o segundo programa deve ter como entrada (i.e., pré-condição) a saída (i.e., pós-condição) do primeiro.

Sequência

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

Exemplo 10 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:

$\{\,\mathsf{x} == \mathsf{x}_0 \wedge \mathsf{y} == \mathsf{y}_0\,\}$
`x := x + y ;`
$\{\,\mathsf{R}_2\,\}$
`y := x - y ;`
$\{\,\mathsf{R}_1\,\}$
`x := x - y ;`
$\{\,\mathsf{x} == \mathsf{y}_0 \wedge \mathsf{y} == \mathsf{x}_0\,\}$

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

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

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

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

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

Comecemos por simplificar o consequente desta implicação.

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

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

Exemplo 11 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:

$\{ \texttt{x} == \mathsf{x}_0 \wedge \texttt{y} == \mathsf{y}_0 \}$
`z := x`
$\{ \mathsf{R}_2 \}$
`x := y`
$\{ \mathsf{R}_1 \}$
`y := z`
$\{ \texttt{x} == \mathsf{y}_0 \wedge \texttt{y} == \mathsf{x}_0 \}$

Donde vem:

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

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

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

Simplifiquemos o consequente desta implicação.

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

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.

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

4.4 Condicionais

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

Condicional

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

Que traduz o significado intuitivo da construção $\texttt{if}\;\; c\; \{S_1\}\; \texttt{else}\;\; \{S_2\}$: 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 12 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:	$\texttt{x} == x_0 \wedge \texttt{y} == y_0$
Pós-condição:	$\texttt{M} == \texttt{max}\;\;(x_0, y_0)$

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

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

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

1. $$\begin{aligned} (\texttt{x} > \texttt{y} \wedge \texttt{x} == x_0 \wedge \texttt{y} == y_0) &\Rightarrow (\texttt{M} == \texttt{max}\;\;(x_0, y_0))[\texttt{M} \setminus \texttt{x}] \\ &\Rightarrow \texttt{x} == \texttt{max}\;\;(x_0, y_0) \end{aligned}$$
2. $$\begin{aligned} (\texttt{x} \leq \texttt{y} \wedge \texttt{x} == x_0 \wedge \texttt{y} == y_0) &\Rightarrow (\texttt{M} == \texttt{max}\;\;(x_0, y_0))[\texttt{M} \setminus \texttt{y}] \\ &\Rightarrow \texttt{y} == \texttt{max}\;\;(x_0, y_0) \end{aligned}$$

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

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

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

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 13 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 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 $\mathtt{x}$ e de $\mathtt{y}$ permanecem inalterados.
- O valor de $\mathtt{m}$ cresce proporcionalmente ao decréscimo de $\mathtt{d}$.

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

Ao tentarmos mostrar a validade desta última implicação 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.
É claro que, acrescentando esta conjunção ao invariante, teremos que recalcular as três implicações.

5 Correcção Total

Relembremos a especificação apresentada no Exemplo 7 do cálculo da raiz quadrada de um número positivo.

Pré-condição: $\mathtt{x} == x_0 > 0$
Pós-condição: $\mathtt{r} * \mathtt{r} == x_0$

Consideremos o seguinte programa:

```
r=0;
while (r>=0)
   r = r+1;
```

Finalmente, seja I o seguinte invariante $I \doteq \mathtt{r} \geq 0$.
Para *mostarmos* a correcção deste programa face à especificação, devemos provar as seguintes condições:

1. $$\begin{aligned}\mathtt{x} == x_0 > 0 \quad &\Rightarrow (\mathtt{r} \geq 0)[\mathtt{r} \setminus 0] \\ &\Rightarrow (0 \geq 0)\end{aligned}$$
2. $$\begin{aligned}(\mathtt{r} \geq 0 \wedge \mathtt{r} \geq 0) \quad &\Rightarrow (\mathtt{r} \geq 0)[\mathtt{r} \setminus \mathtt{r} + 1] \\ &\Rightarrow (\mathtt{r} + 1 \geq 0) \\ \mathtt{r} \geq 0 \quad &\Rightarrow \mathtt{r} \geq -1\end{aligned}$$

3. $\texttt{r} \geq 0 \wedge \neg(\texttt{r} \geq 0) \quad \Rightarrow \texttt{r} * \texttt{r} == x_0$
   $\texttt{False} \quad \Rightarrow \texttt{r} * \texttt{r} == x_0$

Que são trivialmente verdadeiras.
No entanto a intuição diz-nos que o programa em causa não calcula a raíz quadrada. Para entendermos esta discrepância, vejamos novamente as permissas da regra de correcção de um ciclo.

- As primeiras duas ($P \Rightarrow I$ e $\{ I \wedge c \}\ S\ \{ I \}$) garantem-nos que o invariante é válido antes de cada iteração do ciclo.
- A terceira ($I \wedge \neg c$) indica que, **quando o ciclo terminar**, a pós-condição é válida.

Ora é exactamente isso que se passa – o ciclo usado não termina!
Este exemplo motiva a introdução de uma definição mais restritiva de correcção – **correcção total** – e de nos referirmos à definição apresentada como **correcção parcial**.
Assim, dada uma especificação (P, Q) dizemos que um programa S está **totalmente correcto** face a essa especificação e escrevemos $[\,P\,]\ S\ [\,Q\,]$ sse

1. $\{ P \}\ S\ \{ Q \}$, ou seja que o programa está parcialmente correcto
2. Partindo de qualquer estado em que P é válido, o programa S termina.

As regras de inferência da correcção total são em tudo idênticas às da correcção parcial, excepto para a regra do ciclo. Esta é a única construção em que a termionação de um programa não depende exclusivamente da correcção das suas partes.
Assim teremos:

1. Atribuição
$$\frac{P \Rightarrow (Q[x \setminus E])}{[\,P\,]\ x := E\ [\,Q\,]} \quad \text{(Atrib)}$$
2. Sequência
$$\frac{[\,P\,]\ S_1\ [\,R\,] \quad [\,R\,]\ S_2\ [\,Q\,]}{[\,P\,]\ S_1\ S_2\ [\,Q\,]} \quad \text{(Seq)}$$
3. Condicional
$$\frac{[\,P \wedge c\,]\ S_1\ [\,Q\,] \quad [\,P \wedge \neg c\,]\ S_2\ [\,Q\,]}{[\,P\,]\ \texttt{if}\ \ c\ \{S_1\}\ \texttt{else}\ \ \{S_2\}\ [\,Q\,]} \quad \text{(ifThenElse)}$$

Para provarmos a correcção total de um ciclo vamos usar um conceito novo – **variante** – que consiste numa expressão **inteira V** que decresce (estritamente) em cada iteração do ciclo sem nunca ultrapassar um dado valor (tipicamente 0). A regra de correcção total de um ciclo será então.

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

Vejamos com mais pormenor as permissas que são diferentes das da regra da correcção parcial de um ciclo.

- $I \wedge c \Rightarrow V \geq 0$ significa que sempre que se pode fazer uma iteração do ciclo, o variante é positivo
- $[\, I \wedge c \wedge V == v_0 \,]\; S\; [\, I \wedge V < v_0 \,]$
  - A conjunção $V == v_0$ na pré-condição fixa o valor do variante antes de se efectuar uma iteração do ciclo.
  - A conjunção $V < v_0$ na pós-condição impõe que o valor do variante após a execução de uma iteração do ciclo decresce (estritamente).

No exemplo acima, da multiplicação inteira (pag. 13), o variante é fácil de encontrar: $V \doteq$ `d` uma vez que o valor desta variável (inteira) decresce em cada iteração do ciclo e tem um valor mínimo. Noutros casos a determinação do variante não é tão fácil.

6 Exemplos

Nesta secção vamos apresentar alguns exemplos de análise de correcção.
Nestes exemplos, e de forma a motivar o uso de ferramentas de apoio a esta análise, vamos apresentar os programas anotados com as várias condições (pré, pós, invariantes e definição de variantes).
Também com o mesmo propósito, vamos obedecer a algumas restrições nos programas apresentados, nomeadamente a de não alterar os valores dos argumentos das funções.
Desta forma, o exemplo apresentado nas secções anteriores, de cálculo do produto de dois números será escrito da seguinte forma.

```
int mult (int x, int y){
  // pre: y >= 0
  int m = 0, d = y;
  while (d>0) {
    // inv: d>=0 && m == x * (y-d);
    // var: d
    m = m+x;
    d = d-1;
  }
  // pos: m == x*y
  return m;
}
```

6.1 Multiplicação (binária)

A estratégia usada acima para calcular o produto de `x` por `y` consiste em calcular um somatório com `y` parcelas, todas iguais a `x`. Uma forma mais eficiente de calcular este produto consiste em diminuir o número de parcelas aumentando o valor de cada parcela. Por exemplo,

- para `y` um número par (i.e., `y = z+z`),

$$x * y = \underbrace{x + x + \cdots + x}_{\text{y parcelas}} = \underbrace{(x + x) + (x + x) + \cdots + (x + x)}_{\text{z parcelas}}$$

- para y um número ímpar (i.e., `y = 1+z+z`)

$$x * y = \underbrace{x + x + \cdots + x}_{\text{y parcelas}} = x + \underbrace{(x + x) + (x + x) + \cdots + (x + x)}_{\text{z parcelas}}$$

Daqui resulta a seguinte solução:

```
int multBin (int x, int y){
  // pre: y >= 0
  int r = 0, a = x, b = y;
  while (b>0) {
    // inv: ???
    // var: ???
    if (b%2 == 1) r = r + a;
    a = a * 2;  // equiv a:  a = a<<1
    b = b / 2; // equiv a: b = b>>1
  }
  // pos: r == x*y
  return m;
}
```

De forma a definir o invariante deste ciclo devemos ter em atenção que, à medida que vamos mudando o número de parcelas (`b`) e o valor de cada parcela (`a`), em `r` vamos acumulando o que não pode ser factorizado dessa forma (ver o exemplo acima para quando `y` é ímpar). Dito isto, a propriedade (invariante) que nos permitirá provar a correcção (parcial) desta função é

$$I \doteq (r + a * b = x * y) \wedge b \geq 0$$

A última conjunção será usada para mostrarmos que quando o ciclo termina o valor da variável `b` é 0.
Para provarmos a terminação do ciclo podemos usar como variante o valor da variável `b`.
As condições que daqui resultam são:

1. ($\{ P \}$ `r = 0; a = x; b = y` $\{ I \}$)

$$\begin{aligned} y \geq 0 \quad & \Rightarrow ((r + a * b = x * y) \wedge b \geq 0)[b \setminus y][a \setminus x][r \setminus 0] \\ & \Rightarrow ((0 + x * y = x * y) \wedge y \geq 0) \end{aligned}$$

2. ($I \wedge c \Rightarrow V \geq 0$)

$$(r + a * b = x * y) \wedge b \geq 0 \Rightarrow b \geq 0$$

3. ($\{ I \wedge c \wedge V = v_0 \}\ S\ \{ I \wedge V < v_0 \}$)
   Como o corpo do ciclo começa com um `if`, podemos desde já partir nos dois casos:

(a) $(\{\, I \wedge b > 0 \wedge b\%2 = 1 \wedge v_0 = b \,\}\ r = r + a; b = b/2; a = a * 2\ \{\, I \wedge b < v_0 \,\})$

$$\begin{array}{l}((r + a * b = x * y) \wedge b \geq 0 \wedge b > 0 \wedge b\%2 = 1 \wedge v_0 = b) \\ \Rightarrow ((r + a * b = x * y) \wedge b \geq 0 \wedge b < v_0)[a \setminus a * 2][b \setminus b/2][r \setminus r + a] \\ \Rightarrow (((r + a + (a * 2) * (b/2) = x * y) \wedge b \geq 0 \wedge b/2 < v_0)\end{array}$$

Note-se que, nas condições do antecedente ($b\%2 = 1$) a expressão $b/2$ é equivalente a $(b-1)/2$ e, por isso, $b/2*(a*2) = (b-1)/2*(a*2) = (b-1)*a = b*a-a$.

(b) $(\{\, I \wedge b > 0 \wedge b\%2 \neq 1 \wedge v_0 = b \,\}\ b = b/2; a = a * 2\ \{\, I \wedge b/2 < v_0 \,\})$

$$\begin{array}{l}((r + a * b = x * y) \wedge b \geq 0 \wedge b > 0 \wedge b\%2 \neq 1 \wedge v_0 = b) \\ \Rightarrow ((r + a * b = x * y) \wedge b \geq 0 \wedge b < v_0)[a \setminus a * 2][b \setminus b/2] \\ \Rightarrow (((r + a + (a * 2) * (b/2) = x * y) \wedge b \geq 0 \wedge b/2 < v_0)\end{array}$$

Note-se que, nas condições do antecedente ($b\%2 \neq 1$) a expressão $b/2*(a*2) = b * a$.

4. $(I \wedge \neg c \Rightarrow Q)$

$$(((r + a * b = x * y) \wedge b \geq 0 \wedge b \leq 0) \Rightarrow (r = x * y)$$

Das condições do antecedente ($b \geq 0 \wedge b \leq 0$) pode-se concluir que $b = 0$.

6.2 Valor de um polinómio num ponto

Pretende-se definir uma função que, dado um polinómio (por exemplo $3.x^5 - 4.x^2 + 3$) e um ponto (por exemplo $x = 10$) calcule o valor do polinómio nesse ponto ($3 * 10^5 - 4 * 10^2 + 3 = 299603$). Assumindo que os polinómios são representados por um array de coeficientes, em que no índice i se encontra o coeficiente correspondente a x^i, uma possível implementação dessa função seria:

```
int valor (float x, float p[], int N){
  // pre: N >= 0
  float r = 0; int i = 0;
  while (i<N) {
    // inv: ???
    // var: ???
    r = r + pow(x,i) * p[i];
    i = i+1;
  }
  // pos: r = sum_{0<=k<=N-1} p[k] * (x^k)
  return m;
}
```

De forma a determinarmos o invariante necessário, vamos começar por apresentar um exemplo de execução para um polinómio de grau 4 (`N=5`). O que queremos mostrar é que no final do ciclo, na variável `r` esteja o valor

$$\begin{array}{ll} r & = \sum_{k=0}^{4} p[k] * x^k \\ & = p[0] * x^0 + p[1] * x^1 + p[2] * x^2 + p[3] * x^3 + p[4] * x^4 \\ & = p[0] + p[1] * x + p[2] * x^2 + p[3] * x^3 + p[4] * x^4 \end{array}$$

i	r
0	0
1	$x^0 * p[0]$
2	$x^0 * p[0] + x^1 * p[1]$
3	$x^0 * p[0] + x^1 * p[1] + x^2 * p[2]$
4	$x^0 * p[0] + x^1 * p[1] + x^2 * p[2] + x^3 * p[3]$
5	$x^0 * p[0] + x^1 * p[1] + x^2 * p[2] + x^3 * p[3] + x^4 * p[4]$

Este exemplo evidencia que em cada iteração estamos a calcular mais uma das parcelas do somatório pretendido, ou seja,

$$r = \sum_{k=0}^{i-1} p[k] * x^k$$

Esta propriedade, juntamente com outra que nos garanta que no final do ciclo `i` é igual a `N`, é suficiente para mostrar a correcção parcial.
Quanto à terminação, o valor da variável `i` aumenta em cada iteração sem nunca ultrapassar `N`, pelo que a expressão `N-i` é um variante adequado.
Resumindo,

$$\begin{array}{rcl} I & \doteq & r = \sum_{k=0}^{i-1} p[k] * x^k \wedge i \leq N \\ V & \doteq & N - i \end{array}$$

Exercício 5 Tal como fizemos acima para a multiplicação binária, apresente as condições de verificação correspondentes à prova da correção (**total**) desta função.

Uma pequena optimização que podemos fazer na função apresentada, e de forma a estar sempre a calcular potências de `x`, consiste em usar uma variável onde essas potências vão sendo calculadas.

```
int valor (float x, float p[], int N){
  // pre: N >= 0
  float r = 0, pot = 1; int i = 0;
  while (i<N) {
    // inv: ???
    // var: ???
    r = r + pot * p[i];
    pot = pot * x;
    i = i+1;
  }
  // pos: r = sum_{0<=k<=N-1} p[k] * (x^k)
  return r;
}
```

Esta pequena mudança equivale também a uma pequena mudança no invariante. Devemos incluir o *significado* da variável `pot`. Assim teremos:

$$\begin{array}{rcl} I & \doteq & (r = \sum_{k=0}^{i-1} p[k] * x^k) \wedge (i \leq N) \wedge (\mathtt{pot} = x^i) \\ V & \doteq & N - i \end{array}$$

Uma outra optimização (conhecida como *método de Horner*) que se pode fazer a esta função baseia-se na factorização do somatório que se pretende calcular.

$$
\begin{aligned}
\textstyle\sum_{k=0}^{N-1} p[k] * x^k \quad &= p[0] * x^0 + p[1] * x^1 + p[2] * x^2 + \cdots + p[N-1] * x^{N-1} \\
&= p[0] + p[1] * x + p[2] * x^2 + \cdots + p[N-1] * x^{N-1} \\
&= p[0] + x * (p[1] + p[2] * x + \cdots + p[N-1] * x^{N-2}) \\
&= p[0] + x * (p[1] + x * (p[2] + \cdots + p[N-1] * x^{N-3})) \\
&= \cdots \\
&= p[0] + x * (p[1] + x * (p[2] + \cdots + x * (p[N-1] + 0) \cdots)
\end{aligned}
$$

Esta factorização sugere que se percorra o array da direita para a esquerda (i.e., o índice `i` varia entre `N` e `0`), acumulando em `r` os resultados parciais explicitados acima.
Para o exemplo que vimos acima ($N = 5$), a evolução do valor das várias variáveis será:

`i`	`r`
5	0
4	$\underbrace{p[4]}_{}$
3	$\underbrace{p[3] + x * \overbrace{p[4]}^{}}_{}$
2	$\underbrace{p[2] + x * (\overbrace{p[3] + x * p[4]}^{})}_{}$
1	$\underbrace{p[1] + x * (\overbrace{p[2] + x * (p[3] + x * p[4])}^{})}_{}$
0	$p[0] + x * (\overbrace{p[1] + x * (p[2] + x * (p[3] + x * p[4]))}^{})$

De forma a escrevermos o correspondente invariante, vamos apresentar a tabela acima com os valores de `r` *desfactorizado*

`i`	`r`
5	0
4	$p[4]$
3	$p[3] + x * p[4]$
2	$p[2] + x * p[3] + x^2 * p[4]$
1	$p[1] + x * p[2] + x^2 * p[3] + x^3 * p[4]$
0	$p[0] + x * p[1] + x^2 * p[2] + x^3 * p[3] + x^4 * p[4]$

Donde podemos escrever

$$I \quad \doteq \quad (r = \textstyle\sum_{k=i}^{N-1} p[k] * x^{k-i}) \wedge (i \geq 0)$$

Daqui resulta a seguinte definição:

```
int valor (float x, float p[], int N){
  // pre: N >= 0
  float r = 0; int i = N;
  while (i>0) {
    // inv: r == sum_{i<=k<=N-1} p[k] * x^{k-i}) && (i >= 0)
    // var: i
    i = i-1;
    r = p[i] + x * r;
  }
  // pos: r = sum_{0<=k<=N-1} p[k] * (x^k)
  return r;
}
```

Exercício 6 Apresente as condições de verificação correspondentes à prova da correção (**total**) desta função.

6.3 Procura num array

Uma função que procura um valor (inteiro) num array (de inteiros) pode ser especificada como:

Pré-condição: $N \geq 0$
Pós-condição: $(r = -1 \wedge \forall_{0 \leq k < N}\, v[k] \neq x) \vee (0 \leq r < N \wedge v[r] = x)$

Note-se a disjunção na pós-condição:

- o valor de `r` deve ser `-1` se o elemento não existir no array
- no caso de existir, o valor de `r` é o índice onde ele se encontra

Como não existe qualquer restrição sobre a ordenação do array, a procura tem que ser feita de forma exaustiva, consultando todos os elementos até encontrarmos ou percorrendo todo o array.

```
int procura (int x, int v[], int N){
  // pre: N >= 0
  int r = -1, i = 0;
  while (r == -1 && i < N){
    // inv:
    // var:
    if (v[i] == x) r = i;
    i = i+1;
  }
  // pos: (r == -1 && forall_{0 <= k < N} v[k] != x))
  //      (0 <= r < N && v[r] == x)
  return r;
}
```

A variável `r` vai ter o valor `-1` enquanto o elemento não for encontrado. Por outro lado, o índice `i` marca a primeira posição do array que ainda não foi consultada. Assim, o invariante deste ciclo pode ser definido como uma variante da pós-condição, onde nos pronunciamos apenas sobre a parte do array já consultada.
Quanto à terminação, a expressão `N-i` satisfaz os requisitos para um variante.

$$\begin{array}{rcl} I & \doteq & (i \leq N) \wedge ((r = -1 \wedge \forall_{0 \leq k < i}\, v[k] \neq x) \vee (0 \leq r < i \wedge v[r] = x)) \\ V & \doteq & N - i \end{array}$$

As condições que daqui resultam são:

1. $(\{\,P\,\}$ `r = -1; i = 0` $\{\,I\,\})$

$$\begin{array}{rcl} N \geq 0 & \Rightarrow & ((i \leq N) \wedge ((r = -1 \wedge \forall_{0 \leq k < i}\, v[k] \neq x) \vee (0 \leq r < i \wedge v[r] = x)))[i \setminus 0][r \setminus -1] \\ & \Rightarrow & (0 \leq N) \wedge ((-1 = -1 \wedge \forall_{0 \leq k < 0}\, v[k] \neq x) \vee (0 \leq -1 < 0 \wedge v[r] = x)) \end{array}$$

2. $(I \wedge c \Rightarrow V \geq 0)$

$$\begin{array}{rcl} (i \leq N) & \wedge & ((r = -1 \wedge \forall_{0 \leq k < i}\, v[k] \neq x) \vee (0 \leq r < i \wedge v[r] = x)) \\ & \wedge & r = -1 \wedge i < N) \\ & \Rightarrow & N - i \geq 0 \end{array}$$

3. $(\{\,I \wedge c \wedge V = v_0\,\}\ S\ \{\,I \wedge V < v_0\,\})$
   Como o corpo do ciclo começa com um `if`, podemos desde já partir nos dois casos:

   (a) $(\{\,I \wedge c \wedge v[i] = x \wedge v_0 = N - i\,\}\ r = i; i = i + 1\ \{\,I \wedge N - i < v_0\,\})$

$$\begin{array}{rclcl} (i \leq N) & \wedge & \multicolumn{3}{l}{((r = -1 \wedge \forall_{0 \leq k < i}\, v[k] \neq x) \vee (0 \leq r < i \wedge v[r] = x))} \\ & \wedge & \multicolumn{3}{l}{r = -1 \wedge i < N \wedge v[i] = x \wedge v_0 = N - i} \\ & \Rightarrow & ((i \leq N) & \wedge & ((r = -1 \wedge \forall_{0 \leq k < i}\, v[k] \neq x) \vee (0 \leq r < i \wedge v[r] = x)) \\ & & & \wedge & N - i < v_0)[i \setminus i + 1][r \setminus i] \\ & \Rightarrow & ((i + 1 \leq N) & \wedge & ((i = -1 \wedge \forall_{0 \leq k < i+1}\, v[k] \neq x) \vee (0 \leq i < i \wedge v[i] = x)) \\ & & & \wedge & N - i - 1 < v_0) \end{array}$$

   (b) $(\{\,I \wedge c \wedge (v[i] \neq x) \wedge v_0 = N - i\,\}\ i = i + 1\ \{\,I \wedge N - i < v_0\,\})$

$$\begin{array}{rclcl} (i \leq N) & \wedge & \multicolumn{3}{l}{((r = -1 \wedge \forall_{0 \leq k < i}\, v[k] \neq x) \vee (0 \leq r < i \wedge v[r] = x))} \\ & \wedge & \multicolumn{3}{l}{r = -1 \wedge i < N \wedge v[i] \neq x \wedge v_0 = N - i} \\ & \Rightarrow & ((i \leq N) & \wedge & ((r = -1 \wedge \forall_{0 \leq k < i}\, v[k] \neq x) \vee (0 \leq r < i \wedge v[r] = x)) \\ & & & \wedge & N - i < v_0)[i \setminus i + 1] \\ & \Rightarrow & ((i + 1 \leq N) & \wedge & ((r = -1 \wedge \forall_{0 \leq k < i+1}\, v[k] \neq x) \vee (0 \leq r < i + 1 \wedge v[r] = x)) \\ & & & \wedge & N - i < v_0) \end{array}$$

4. $(I \wedge \neg c \Rightarrow Q)$

$$\begin{array}{rcl} (i \leq N) & \wedge & ((r = -1 \wedge \forall_{0 \leq k < i}\, v[k] \neq x) \vee (0 \leq r < i \wedge v[r] = x)) \\ & \wedge & ((r \neq -1 \vee i \geq N)) \\ & \Rightarrow & (r = -1 \wedge \forall_{0 \leq k < N}\, v[k] \neq x) \vee (0 \leq r < N \wedge v[r] = x) \end{array}$$

Se o array estiver ordenado podemos definir uma versão ligeiramente mais eficiente desta função.

```
int procura (int x, int v[], int N){
  // pre: N >= 0 && forall_{0<k<N} v[k-1]<=v[k]
  int r = -1, i = 0;
  while (r == -1 && i < N && v[i] <= x){
    // inv:
    // var:
    if (v[i] == x) r = i;
    i = i+1;
  }
  // pos: (r == -1 && forall_{0 <= k < N} v[k] != x))
  //      (0 <= r < N && v[r] == x)
  return r;
}
```

Para o leitor menos atento, vejamos as diferenças relativamente à versão anterior:

- Na pré-condição da função foi acrescentada a restrição do array estar ordenado.
- Na condição do ciclo foi acrescentada uma conjunção extra: `v[i] <= x`

É por isso natural que o mesmo invariante satisfaça duas das propriedades necessárias: inicialização e preservação. Contudo, para provarmos a utilidade, i.e., que quando o ciclo termnina a pós-condição é atingida, precisamos de acrescentar uma propriedade extra ao invariante: que o array está ordenado. De facto, quando o ciclo termina por causa da nova condição, i.e., quando `v[i] > x`, precisamos de saber que o array está ordenado para podermos concluir que o elemento não se encontra na parte não consultada do array.
Resumindo,

$$
\begin{array}{rcl}
I & \doteq & (i \leq N) \wedge \\
 & & ((r = -1 \wedge \forall_{0 \leq k < i}\, v[k] \neq x) \vee (0 \leq r < i \wedge v[r] = x)) \wedge \\
 & & \forall_{0 < k < N} v[k-1] \leq v[k] \\
V & \doteq & N - i
\end{array}
$$

Quando o array a pesquisar está ordenado (por ordem crescente) podemos fazer a procura de uma forma substancialmente mais eficiente, eliminando da pesquisa vários elementos do array por cada iteração.

```
int procura (int x, int v[], int N){
  // pre: N >= 0 && forall_{0<k<N} v[k-1]<=v[k]
  int r = -1, i = 0, s = N-1;
  while (r == -1 && i <= s){
    // inv:
    // var:
    m = i + (s-i) / 2;
    if (v[m] == x) r = m;
    else if (v[m] > x) s = m-1;
    else i = m+1;
  }
  // pos: (r == -1 && forall_{0 <= k < N} v[k] != x))
  //      (0 <= r < N && v[r] == x)
  return r;
}
```

Tanto o invariante como o variante deste ciclo podem ser definidos baseados nos que foram definidos atrás.
Para a procura sequencial:

- O invariante descrevia a porção do array em que o elemento não tinha sido encontrado. Vimos ainda que temos que incluir como invariante a propriedade do array estar ordenado.
- O variante media a porção do array que ainda não havia sido consultada.

Adptando estas considerações para a procura binária, temos que:

- A parte do array que sabemos não conter x corresponde aos índices $[0..i[\cup]s..N-1]$
- O tamanho da porção ainda não consultada corresponde a $s - i + 1$

$$
\begin{array}{rcl}
I & \doteq & \quad (i \leq N) \\
& \wedge & ((r = -1 \wedge \forall_{0 \leq k < i}\, v[k] \neq x \wedge \forall_{s < k < N}\, v[k] \neq x) \vee (0 \leq r < i \wedge v[r] = x)) \\
& \wedge & \forall_{0 < k < N}\, v[k-1] \leq v[k] \\
V & \doteq & s - i
\end{array}
$$

6.4 Raíz quadrada inteira

Uma forma de especificar o cálculo da raíz quadrada de um número positivo x consiste em determinar (entre 0 e x) um número r tal que r*r == x.
No caso de se tratar de números inteiros, esta função só estará definida para *quadrados perfeitos* pelo que podemos enfraquecer a pós-condição de forma a calcular o **maior** número r para o qual r * r <= x.

```
int int_sqrt (int x){
  // pre: x >= 0
  ...
  // pos: r * r <= x && forall_{s>r} s*s > x
  return r;
}
```

Uma alternativa de expressar esta pós-condição (evitando o uso de quantificadores) seria

```
int int_sqrt (int x){
  // pre: x >= 0
  ...
  // pos: r * r <= x && (r+1)*(r+1) > x
  return r;
}
```

Uma estratégia para definir esta função (talvez não a mais eficiente) consiste em percorrer todos os quadrados perfeitos até ultrapassarmos o valor de `x`. O valor procurado será então o anterior quadrado perfeito testado.
Esta estratégia corresponde à seguinte definição:

```
int int_sqrt (int x){
  // pre: x >= 0
  int r = 1;
  while (r*r <= x)
    // inv: ???
    // var: ???
    r+=1;
  r-=1;
  // pos: r * r <= x && (r+1)*(r+1) > x
  return r;
}
```

Exercício 7 Determine um invariate e um variante que lhe permitam provar a correcção total da função anterior.
Note que a pós-condição do ciclo não coincide com a pós-condição da função. Para calcular a pós-condição do ciclo devemos começar por *desfazer* a última instrução da função (`r=r-1`).

Podemos apresentar uma versão ligeiramente mais eficiente desta função que, em vez de calcular a potência `r*r` em todas as iterações do ciclo, a vai construindo em cada iteração. E isso pode ser feito de uma forma eficiente sabendo a diferença entre dois quadrados perfeitos consecutivos:

$$\begin{aligned} (r+1)^2 - r^2 &= (r^2 + 2.r + 1) - r^2 \\ &= 2.r + 1 \end{aligned}$$

Daqui resulta a seguinte deinição alternativa.

```
int int_sqrt (int x){
  // pre: x >= 0
  int r=0, p=1, i=1;
  while (p<=x){
    // inv:
    // var:
    r+=1; i+=2; p+=i;
  // pos: r * r <= x && (r+1)*(r+1) > x
  return r;
}
```

De forma a definir o invariante apropriado, vejamos um exemplo de execução desta função, para `x==40`.

`x`	`r`	`p`	`i`
40	0	1	1
40	1	4	3
40	2	9	5
40	3	16	7
40	4	25	9
40	5	36	11
40	6	49	13

A análise deste exemplo aponta para duas propriedades importantes:

- Uma das condições ($r^2 \leq x$) que queremos garantir que é verdadeira no final do ciclo é de facto verdadeira durante toda a execução do ciclo.
- Os valores da variável `p` são os sucessivos quadrados perfeitos. Mais precisamente, $p = (r+1)^2$.

Quanto à terminação, i.e., quanto á determinação do variante deste ciclo, podemos ver que o valor de todas as variáveis (`r`, `p` e `i`) aumenta em cada iteração. Qualquer uma delas pode ser usada para definir o variante, subtraindo-a ao seu maior valor.
Vamos tentar provar a correcção desta função com as seguintes definições:

$$\begin{array}{rcl} I & \doteq & r^2 \leq x \wedge p = (r+1)^2 \\ V & \doteq & x - p \end{array}$$

Vejamos então as condições que teremos que provar.

1. $(\{\, P \,\}$ `r=0; p=1; i=1` $\{\, I \,\})$

$$\begin{array}{rcl} x \geq 0 & \Rightarrow & (r^2 \leq x \wedge p = (r+1)^2)[i \setminus 1][p \setminus 1][r \setminus 0] \\ & \Rightarrow & 0^2 \leq x \wedge 1 = (0+1)^2 \\ & \Rightarrow & 0 \leq x \wedge 1 = 1 \end{array}$$

2. $(I \wedge c \Rightarrow V \geq 0)$

$$\begin{array}{rcl} (r^2 \leq x \wedge p = (r+1)^2 \wedge p \leq x) & \Rightarrow & x - p \geq 0 \\ & \Rightarrow & p \leq x \end{array}$$

3. $(\{\, I \wedge c \wedge V = v_0 \,\}\ S\ \{\, I \wedge V < v_0 \,\})$

$$\begin{array}{rcl} ((r^2 \leq x) & \wedge & (p = (r+1)^2) \wedge (p \leq x) \wedge (v_0 = x - p)) \\ & \Rightarrow & ((r^2 \leq x) \wedge (p = (r+1)^2) \wedge (x - p < v_0))[p \setminus p + i][i \setminus i + 2][r \setminus r + 1] \\ & \Rightarrow & ((r^2 \leq x) \wedge (p + i = (r+1)^2) \wedge (x - (p+i) < v_0))[i \setminus i + 2][r \setminus r + 1] \\ & \Rightarrow & ((r^2 \leq x) \wedge (p + i + 2 = (r+1)^2) \wedge (x - (p+i+2) < v_0))[r \setminus r + 1] \\ & \Rightarrow & (((r+1)^2 \leq x) \wedge (p + i + 2 = (r+2)^2) \wedge (x - p - i - 2 < v_0)) \end{array}$$

4. $(I \wedge \neg c \Rightarrow Q)$

$$\begin{array}{rcl} ((r^2 \leq x) \wedge (p = (r+1)^2) \wedge \neg(p \leq x)) & \Rightarrow & r^2 \leq x \wedge (r+1)^2 > x \\ ((r^2 \leq x) \wedge (p = (r+1)^2) \wedge (p > x)) & \Rightarrow & r^2 \leq x \wedge (r+1)^2 > x \end{array}$$

Com excepção da 3ª condição, todas as implicações são fáceis de provar.
Vejamos então as várias partes do consequente desta 3ª condição:

- $(r+1)^2 \leq x$ é válido pois no antecedente temos que $(r+1)^2 = p$ e que $p \leq x$
- $x - p - i - 2 < v_0$, uma vez que $v_0 = x - p$, vem que

$$\begin{array}{rl} & x - p - i - 2 < x - p \\ \Leftrightarrow & i + 2 > 0 \\ \Leftrightarrow & i > -2 \end{array}$$

- Quanto ao predicado $p + i + 2 = (r+2)^2$, usando o predicado $p = (r+1)^2$ do antecedente, vem que

$$\begin{array}{rl} & (r+1)^2 + i + 2 = (r+2)^2 \\ \Leftrightarrow & r^2 + 2.r + 2 + i + 2 = r^2 + 4.r + 4 \\ \Leftrightarrow & i = 2.r + 1 \end{array}$$

A tabela acima indica-nos que as propriedades extra que precisamos são válidas ao longo das várias iterações do ciclo. A forma de as podermos usar na prova acima é inclui-las no invariante.
Desta forma,

$$\begin{array}{rcl} I & \doteq & (r^2 \leq x) \wedge (p = (r+1)^2) \wedge (i = 2 * r + 1) \wedge (i > 0) \\ V & \doteq & x - p \end{array}$$

Exercício 8 Recalcule as condições de verificação com esta definição do invariante.

6.5 Divisão e resto da divisão inteira

A divisão inteira de dois números pode ser especificada da seguinte forma:

Pré-condição: $x = x_0 \geq 0 \ \wedge \ y = y_0 > 0$
Pós-condição: $0 \leq r < y_0 \ \wedge \ q * y_0 + r = x_0$

A função seguinte calcula estes dois valores de uma forma pouco eficiente:

```
void divmod (int x, int y, int *rem){
  // pre: x >= 0 && y > 0
  int q = 0, r = x;
  while (r>=y) {
      // inv: ???
      // var: ???
      r = r-y; q = q+1;
  }
  // pos: 0 <= r < y && q * y + r == x
  *rem = r; return q;
}
```

Comecemos por analisar um exemplo da execução desta função, em que x=45 e y=7.

x	y	q	r
45	7	0	45
45	7	1	38
45	7	2	31
45	7	3	24
45	7	4	17
45	7	5	10
45	7	6	3

Das propriedades que queremos ver estabelecidas na pós-condição podemos constatar que quase todas são válidas ao longo das várias iterações do ciclo. A única que não se verifica sempre (só se verifica quando o ciclo termina) corresponde à negação da condição do ciclo.
No que diz respeito à terminação podemos ver que o valor da variável r diminui sempre (é contudo necessário incluir no invariante que a quantidade a subtrair a r em cada iteração (i.e., y é positiva). pelo que podemos provar a correcção total desta função com as seguintes definições.

$$\begin{aligned} I &\doteq 0 \leq r \wedge q * y + r = x \wedge y > 0 \\ V &\doteq r \end{aligned}$$

Exercício 9 Calcule as condições de verificação com esta definição de invariante e variante do ciclo.

Um problema mais simples consiste em calcular apenas o resto da divisão inteira. Esta maior simplicidade não se traduz numa maior simplicidade na sua especificação. De facto, a especificação dessa função resulta da anterior, *escondendo* a variável q.

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

Pós-condição: $0 \leq r < y_0 \wedge \exists q\, q * y_0 + r = x_0$

A função anterior pode ser *simplificada* para calcular apenas o resto da divisão inteira.

```
void div (int x, int y){
  // pre: x >= 0 && y > 0
  int r = x;
  while (r>=y) {
      // inv: ???
      // var: ???
      r = r-y;
  }
  // pos: 0 <= r < y && exists_{q} q * y + r == x
  return r;
}
```

As considerações feitas atrás para a função `divmod` continuam válidas pelo que vamos provar a correcção parcial desta função com as seguintes definições:

$$\begin{array}{rcl} I & \doteq & 0 \leq r \wedge \exists_q\, q * y + r = x \wedge y > 0 \\ V & \doteq & r \end{array}$$

As condições de verificação que daqui resultam são:

1. $\{\, P \,\}$ `r` $= x \,\{\, I \,\}$

$$\begin{array}{rcl} x \geq 0 \wedge y > 0 & \Rightarrow & (0 \leq r \wedge \exists_q\, q * y + r = x \wedge y > 0)[r \setminus x] \\ & \Rightarrow & (0 \leq x \wedge \exists_q\, q * y + x = x \wedge y > 0) \end{array}$$

   Para provarmos que $\exists_q\, q * y + x = x$ basta-nos fornecer como *testemunha* para q o valor 0.

2. $I \wedge c \Rightarrow V \geq 0$

$$(0 \leq r \wedge \exists_q\, q * y + r = x \wedge y > 0 \wedge r \geq y \quad \Rightarrow \quad r \geq 0$$

3. $\{\, I \wedge c \wedge V = v_0 \,\}$ `r = r-y` $\{\, I \wedge V < v_0 \,\}$

$$\begin{array}{rl} (0 \leq r & \wedge \quad \exists_q\, q * y + r = x \wedge y > 0 \wedge r \geq y \wedge r = v_0) \\ & \Rightarrow \quad (0 \leq r \wedge \exists_q\, q * y + r = x \wedge y > 0 \wedge r < v_0)[r \setminus r - y] \\ & \Rightarrow \quad (0 \leq r - y \wedge \exists_q\, q * y + (r - y) = x \wedge y > 0 \wedge r - y < v_0) \\ & \Rightarrow \quad (y \leq r \wedge \exists_q\, (q - 1) * y + r = x \wedge y > 0 \wedge r - y < v_0) \end{array}$$

   Vejamos as várias partes do consequente desta implicação.

   - $y \leq r$ e $y > 0$ são válidos uma vez que aparecem no antecedente.
   - $r - y < v_0$ uma vez que $r = v_0$, reduz-se a $y > 0$.
   - Para mostrarmos que $\exists_q\, (q-1) * y + r = x$ devemos fornecer uma *testemunha* q_1 tal que $(q_1 - 1) * y + r = x$. Ora do antecedente, podemos concluir a existência de uma *testemunha* q_0 que satisfaz $q_0 * y + r = x$. Seja $q_1 \doteq q_0 + 1$. Então,

$$\begin{array}{rl} & (q_1 - 1) * y + r = x \\ \Leftrightarrow & (q_0 + 1 - 1) * y + r = x \\ \Leftrightarrow & q_0 * y + r = x \end{array}$$

4. $I \wedge \neg c \Rightarrow Q$

$$\begin{array}{rcl} (0 \leq r \wedge \exists_q\, q * y + r = x \wedge y > 0 \wedge \neg(r \geq y)) & \Rightarrow & 0 \leq r < y \wedge \exists_q\, q * y + r = x \\ (0 \leq r \wedge \exists_q\, q * y + r = x \wedge y > 0 \wedge (r < y)) & \Rightarrow & 0 \leq r < y \wedge \exists_q\, q * y + r = x \end{array}$$