CORREÇÃO DE PROGRAMAS
DRAFT 1

RAUL H.C.LOPES

1. Introdução

Estas notas introduzem um problema importante da arte de programação de computadores: correção de programas no paradigma imperativo. O conjunto de exemplos usados aborda em geral o tratamento de seqüências: pesquisa, soma, particionamento. Os problemas de particionamento motivarão a próxima seqüência de notas que trata de ordenação de seqüências.

Referências fundamentais sobre esses problemas são:

- correção de programas: [5], o marco inicial na área, [4], coleção artigos abordando algoritmos e provas interessantesm [6], texto claro e fácil de ler.
- ordenação: [9], a bíblia.

Cada uma das próximas seções proporá um problema, cuja solução será delineada. A cada solução apresentada, notação da linguagem de comandos guardados será introduzida e exercícios serão propostos.

Notação 1. *O símbolo* $\stackrel{\triangle}{=}$ *é usado para introduzir* nomes *como abreviações de fórmulas, ou seja, introduzir* nomes/símbolos *via definiçãológica. Por exemplo,*

$$P \stackrel{\triangle}{=} 0 \leq k \leq m \wedge S = \sum i : 0 \leq i < k : v.i$$

introduz o símbolo P *como nome para a fórmula*

$$0 \leq k \leq m \wedge S = \sum i : 0 \leq i < k : v.i$$

Notação 2. *As seguintes regras, essencialmente retiradas de* [2] *e usadas tanto por Curry* [1] *quanto por Church, poderão ser usadas para reduzir o número de parêntesis usados em expressões aritméticas e lógicas.*

[1] Aliás, se você tem interesse sério em **Ciência da Computação**, cedo ou tarde deveria ler [2].

- *operadores aritméticos são mais fortes do que operadores relacionais. Assim*

$$1 + 2 = 3$$

obviamente é uma abreviação de

$$(1 + 2) = 3$$

- *dentre os operadores relacionais, negação é o mais forte, seguido de conjunção e disjunção e, por último, implicação e igualdade.*
- *ponto (.) e dois pontos (:) podem ser usados como* marcadores *para reduzir a número de parêntesis. Fim e início de expressão são marcadores com zero pontos. As regras básicas para seu uso neste texto são:*
  (1) *um marcador com mais pontos tem mais força do que um marcador com menos pontos.*

$$rev.x^\frown y :=: revy^\frown x$$

abrevia

$$(rev(x^\frown y)) = (rev(y^\frown x))$$

  (2) *um marcador à esquerda de um operador delimita uma expressão que se estende para a esquerda até um marcador de mais alta prioridade.*

$$x + f.y$$

abrevia

$$x + f(y)$$

  (3) *um marcador à direita de um operador delimita uma expressão que se estende para a direita até um marcador de mais alta prioridade.*
  (4) *a expressão delimitada por um marcador respeita os tipos da expressão envolvida.*

$$rev.\varepsilon = \varepsilon$$

abrevia

$$rev(\varepsilon) = \varepsilon$$

e não

$$rev(\varepsilon = \varepsilon)$$

que violaria o fato de rev mapeia seqüências para seqüências.

(5) *um ponto amarrado a um operador relacional tem mais força do que um ponto ligado operador funcional.*

$$rev.x .=. rev.y$$

abrevia

$$rev(x) = rev(y)$$

(6) *um marcador mais à esquerda tem mais prioridade do que um marcador mais á direita.*

$$p \Rightarrow . \; q \Rightarrow . \; r \Rightarrow s$$

abrevia

$$p \Rightarrow (q \Rightarrow (\Rightarrow s)$$

As provas listadas nestas notas e em outras presentadas ao longo do curso apresentam provas baseadas no formalismo da lógica equacional, usada também em [8]. Algumas dessas regras são apresentadas abaixo. Você pode usar os sistemas de prova, como Dedução Natural ou Sequent Calculus, ou sistema de Hilbert. O importante é que use um sistema de forma coerente e apresente referência para as fontes de suas regras para que não restem dúvidas sobre quaisquer passos de suas provas.

IMPORTANTE: Estas notas não substituem o conhecimento que você adquiriu na disciplina de **Noções de lógica**.

Definição 1. *A seguir são apresentados teoremas da lógica de primeira ordem que são usados em diversas provas ao longo deste documento. Note que:*

- *Equivalência lógica de duas proposições é representada pela igualdade da mesma, ou seja,*

$$p = q$$

é usado em lugar de

$$p \equiv q$$

- *O símbolo $\vdash$ define uma relação de conseqüência.*

$$p, q \vdash r$$

abrevia: " assumindo que p e q são verdades, então é possível provar r.

- *Regras de inferência são freqüentemente introduzidas usando a notação:*

$$\frac{\Gamma_0 \vdash \alpha_1, \quad \Gamma_1 \vdash \alpha_1}{\Lambda \vdash \beta}$$

que abrevia o raciocínio: se Γ_0 é um conjunto de hipóteses apartir do qual se pode provar α_0 e Γ_1 é outro conjunto de

hipóteses do qual se pode provar α_1, então do conjunto de hipóteses Λ é possível provar (deduzir) β.

Para exemplo mais concreto, observe a seguinte a regra, apresentada como Introdução da Conjunção *no sistema de Dedução Natural.*

$$\frac{\vdash p, \quad \vdash q}{\vdash p \wedge q}$$

que diz que

- *se p é um teorema da lógica e q é um teorema da lógica, então $(p \wedge q)$ é um teorema da lógica;*
- *ou, lendo de baixo para cima, para provar que $(p \wedge q)$ é verdadeiro (ou seja, $\vdash (p \wedge q)$), prove que p é verdadeiro (ou seja, $\vdash p$) e prove que q é verdadeiro (ou seja, $\vdash q$).*

A lista a seguir apresenta alguns dos teoremas da lógica clássica usados. Ela é obviamente incompleta: por exemplo, não estão listados comutatividade de $\wedge$ e $\vee$, associatividade, etc.

- **Introdução de** $\vee : \vdash p \Rightarrow (p \vee q)$
- **Eliminação de** $\vee : \vdash (p \Rightarrow r) \wedge (q \Rightarrow r) \; .\Rightarrow. \; (p \vee q) \Rightarrow r$
- **Eliminação de** $\wedge : \vdash p \wedge q \Rightarrow p$
- **Modus Ponens** $:\vdash \; p \wedge (p \Rightarrow q) \Rightarrow q$
- **Conseqüência** : $\vdash (q \Leftarrow p) = (p \Rightarrow q)$
- **Leibniz** : $(e = f) \vdash E[z : e] = E[z := f]$
- **de Morgan sobre** $\wedge : \vdash \neg(p \wedge q) = \neg p \vee \neg q$
- **de Morgan sobre** $\vee : \vdash \neg(p \vee q) = \neg p \wedge \neg q$
- **de Morgan sobre cand** $: \vdash \neg(p \textbf{ cand } q) = \neg p \textbf{ cor } \neg q$
- **de Morgan sobre cor** $: \vdash \neg(p \textbf{ cor } q) = \neg p \textbf{ cand } \neg q$
- **distributividade de** $\wedge$**sobre**$\vee : \vdash r \wedge (p \vee q) = (r \wedge p) \vee (r \wedge q)$

Técninas usuais de prova são codificadas pelas regras:

$$\textbf{Teorema da dedução}: \frac{P \vdash Q}{\vdash P \Rightarrow Q}$$

$$\textbf{Análise de caso}: \frac{\vdash E[z := true], \quad \vdash E[z := false]}{\vdash E[z := P]}$$

$$\textbf{Prova por contradição}: \frac{\vdash \neg P \Rightarrow false}{\vdash P}$$

$$\textbf{Contrapositiva}: \frac{\vdash \neg Q \Rightarrow \neg P}{\vdash P \Rightarrow Q}$$

$$\textbf{Implicação mútua}: \frac{\vdash P \Rightarrow Q, \quad \vdash Q \Rightarrow P}{\vdash P = Q}$$

$$\textbf{Introdução de } \wedge: \frac{\vdash P, \quad \vdash Q}{\vdash P \wedge Q}$$

2. Pesquisa em uma seqüência

O problema aqui consiste em determinar se um elemento x ocorre em uma seqüência.

Uma seqüência de elementos de um tipo A é definida através do seguinte conjunto de axiomas:

(1) seqüência vazia: $\varepsilon \in seq.A$

(2) construtor: $c \lhd x \in seq.A$

(3) diferença de vazio:$c \lhd x \neq \varepsilon$

(4) Igualdade: $b \lhd x = c \lhd y \ .=. \ b = c \wedge x = y$

Além disso, o seguinte priincípio da indução é usado para provar propriedades sobre seqüências.

(5)
$\forall x : x \in seq.A : P.x \ .=. \ P.\varepsilon \wedge \forall c, y : c \in A \wedge y \in seq.A : P.y \Rightarrow P.c \lhd y$

Uma seqüência pode ser representada por uma lista ou por um vetor. Seus elemtos são todos de um mesmo tipo A. Neste texto, em princípio, todas as seqüências contêm inteiros, tipo Int. Serão abordadas soluções funcionais usando listas em Haskell e soluções imperativas, usando vetores e matrizes em uma linguagem de comandos guardados[5, 6].

2.1. **Pesquisa Linear.** Dada uma seqüência S de m inteiros e um elementos x, também inteiro, determine se $x \in S$.

2.1.1. *Uma solução funcional.* Uma solução em Haskell para o problema, assumindo que S é representada por uma lista é bastante trivial.

Motivação

$$x \in S = (x = S.0 \vee x = S.1 \vee \ldots \vee x = S.m - 1)$$

A solução

```
linsearch x xs = foldl (||) False (map (x==) xs)
```

Note que essa solução tem a desvantagem de passar pela seqüência duas vezes. A solução apresentada em sala de aula, que aplica a idéia, da associatividade à esquerda compara x com cada elemento da seqüência apenas uma vez.

Exercício 1. *Considere ainda o problema de apresentar soluções funcionais para busca de um elemento em seqüência.*

(1) *Mostre que a solução acima implementa diretamente o resultado proposto* $\exists y : y \in S : x = y$.
(2) *Use os combinadores* ***foldl*** *e* ***foldr*** *para produzir soluções funcionais para o problema.*
(3) *Produza um algoritmo recursivo para resolver esse problema.*
(4) *Prove a correção das suas soluções.*

2.1.2. *A solução imperativa.* Considere agora que a seqüência é representada por um vetor: $s.0..m-1 \in Int$ e $x \in Int$. Determine

$$R = (\exists i : 0 \leq i < m : x = s.i)$$

Notação 3. *A notação associada a vetores é essencialmente emprestada da linguagem Ada:*

- $S.i$ *ou* $S(i)$*, elemento da posição* i *de* S*;*
- $S.0..m-1$ *ou* $S(0..m-1)$*, vetor com* m *elementos, indexados de* 0 *a* m*;*
- *Dado um operador relacional* $\oplus$*,*

$$B(i..j) \oplus S(i..j) \stackrel{\triangle}{=} \forall k : i \leq k \leq j : B.k \oplus S.k$$

$$x \oplus S(i..j) \stackrel{\triangle}{=} \forall k : i \leq k \leq j : x \oplus S.k$$

Motivação:

Relaxe o objetivo, e assuma que $x \notin S$. No último passo da computação seria válido:

$$x \notin S.0..i-1 \wedge i = m$$

e o passo final consistiria em testar atingir determinar que não existe o que buscar porque $i = m$.

```
linserach(S,x,i) is
{m ≥ 0 ∧ S.0..m − 1 ∈ Int ∧ x ∈ Int}
⟦ initially i=0
    {invariant P}
    {bound T}
    do
      G0: m−i ≠ 0 cand x ≠ S.i → i:=i+1
    od
    {P ∧ ¬B}

  ; R:=i=m
  {R .=. ∃j : 0 ≤ j < m : x = S.j}
⟧
```

FIGURA 1. Pesquisa linear

Generalizando para qualquer passo e voltando à especificação original onde x pode ocorrer em S, o último passo da computação assumiria:

- x não ocorreu ainda na seqüência: $x \notin S.0..i-1$, para algum i;
- i é um índice válido na seqüência.

Isso dá a seguinte candidata a invariante:

$$P \triangleq 0 \leq i \leq m \wedge \forall j : 0 \leq i < j : x \neq S.j$$

O progresso na computação ocorre pelo esgotamento da seqüência a pesquisar:

Variável derivada (ou *bound function)*: $T \triangleq m - i$

A computação termina quando

- seqüência está esgotada: $m - i = 0$
- seqüência não está esgotada, mas $x = S.i$;

o que define o ponto-fixo do loop: $(m - i = 0 \textbf{ cor } x = S.i)$. Sua negação, B, dá a condição para computação continuar:

$$B \triangleq (m - i \neq 0 \textbf{ cand } x \neq S.i)$$

Note que

$$(P \wedge \neg B) \Rightarrow \forall k : 0 \leq k < i : x \neq S.k$$

o que, quando $i = m$, significa que x não ocorre em S.

O algoritmo aparece na figura 1. Note que ele aparece decorado com a invariante do loop e a variável derivada, que define o *bound* do loop.

Note que esta solução, ao contrário da funcional, só percorre a seqüência uma vez: mas... cada passo da computação usa duas comparações de inteiros, então...

A correção. A prova de correção do programa é composta dos seguintes passos:

- A correção parcial: a prova de que o programa, se termina, produz o resultado desejado. Essa prova será decomposta nas seguintes fases:
  - prova de que P vale inicialmente;
  - prova de que P é uma invariante do loop;
  - prova de que $(P \wedge \neg B) \Rightarrow \forall k : 0 \leq k < i : x \neq S.k$
- A prova de que o programa termina. A prova será constituída dos seguintes estágios:
  - Prova de que $P \wedge B \Rightarrow T > 0$
  - Prova de que o loop eventualmente termina.
- Prova de eventualmente o programa computa o R desejado.

Teorema 1. *P vale inicialmente.*

Prova.

Primeiro fator de P.

$$\begin{aligned} & \langle \textit{initially} \rangle \\ & i = 0 \wedge m \geq 0 \\ = & \langle \textit{aritmética} \rangle \\ & (0 \leq i \leq m) \end{aligned}$$

Segundo fator de P.

$$\begin{aligned} & \langle \textit{initially} \rangle \\ & i = 0 \wedge m \geq 0 \\ \Rightarrow & \langle \textbf{Eliminação de } \wedge \rangle \\ & i = 0 \\ \Rightarrow & \langle \textit{lógica} \rangle \\ & \forall k : 0 \leq k < i : x \neq S.k \end{aligned}$$

□

Teorema 2. *P é uma invariante do loop.*

Prova.

O objetivo é provar que

$$\{P \wedge B\} G_0 \{P\}$$

Se P é válido e a guarda B é válida, G_0 é executado e, depois, P ainda será válido.

$$
\begin{aligned}
& \langle P \textit{ depois é } P' \rangle \\
& P' = P[i := i+1] \\
= & \langle \textit{substituição de } i \rangle \\
& 0 \leq i+1 \leq m \wedge \forall k : 0 \leq k < i+1 : x \neq S.k \\
= & \langle \textit{aritmética} \rangle \\
& -1 \leq i < m \wedge \forall k : 0 \leq k < i+1 : x \neq S.k \\
= & \langle \textit{lógica} \rangle \\
& -1 \leq i \leq m \wedge m \neq i \wedge \forall k : 0 \leq k < i : x \neq S.k \wedge x \neq S.i \\
\Leftarrow & \langle \textit{weakening} \rangle \\
& (0 \leq i \leq m \wedge \forall k : 0 \leq k < i : x \neq S.k) \wedge (m - i \neq 0 \wedge x \neq S.i) \\
= & \langle \textit{premissa do teorema} \rangle \\
& P \wedge B
\end{aligned}
$$

□

A prova acima consite em mostrar que

$$P \wedge B \;.\Rightarrow.\; \mathbf{wp}(i := i+1, P)$$

onde $\mathbf{wp}(\alpha, P)$ é o transformador de predicados de Dijkstra[5].

Definição 2. *A precondição mais simples* (weakest precondition) *da atribuição é dada por:*

$$\mathbf{wp}(x := e, P) = P[x := e]$$

Teorema 3. *Enquanto a guarda B for válida, a variável derivada T é maior do que zero:*

$$P \wedge B. \Rightarrow .T > 0$$

Prova.

$\langle initially\ P \rangle$
$0 \leq i \leq m \wedge \forall k : 0 \leq k < i : x \neq S.k$
$\Rightarrow \langle pela\ validade\ de\ B \rangle$
$0 \leq i \leq m \wedge \forall k : 0 \leq k < i : x \neq S.k \wedge m - i \neq 0$
$= \langle eliminação\ da\ conjunção\ e\ aritmética\ por\ i \leq m \wedge m - i \neq 0 \rangle$
$i < m$
$= \langle aritmética \rangle$
$0 < m - i$
$= \langle definição\ de\ T \rangle$
$0 < T$
$= T > 0$

□

Teorema 4. *Se B é válida, T decresce.*

Prova.

Trivial: Se B é válida, i cresce e $m - i$ decresce.

□

Teorema 5. *O programa da figura 1 termina.*

Prova.

Se inicialmente $m = 0$, B é falso e o programa termina de imediato. Se $m > 0$, existem dois casos:

(1) *eventualmente x é igual a algum $S.i$ e B se torna falso, terminando o loop e o programa, ou*
(2) *T decresce (provado anteriormente) e atinge zero, tornando B falso, o que termina o loop e o programa.*

Teorema 6. *Ao final do loop, quando $\neg B$ é válido, $x \in S$ se e somente se $i \neq m$.*

$$P \wedge \neg B. = .(i \neq m) = \exists k : 0 \leq k < m : x = S.k$$

Prova.

$$
\begin{aligned}
& P \wedge \neg B \\
=& 0 \leq i \leq m \wedge x \notin S.0..i-1 \wedge \neg(m - i \neq 0 \ \textbf{cand}\ x \neq S.i) \\
=& \langle \textbf{de Morgan sobre cand} \rangle \\
& 0 \leq i \leq m \wedge x \notin S.0..i-1 \wedge (m = i \ \textbf{cor}\ x = S.i) \\
=& \langle \textbf{distributividade de} \wedge \textbf{sobre cor} \rangle \\
& (0 \leq i \leq m \wedge x \notin S.0..i-1 \wedge m = i) \\
& \qquad \textbf{cor}\ (0 \leq i \leq m \wedge x \notin S.0..i-1 \wedge m \neq i \wedge x = S.i) \\
=& (i = m \wedge \neg\exists k : 0 \leq k < m : x = S.k) \\
& \qquad \textbf{cor}\ (0 \leq i < m \wedge \exists k : 0 \leq k < m : x = S.k) \\
=& (i = m) = \neg\exists k : 0 \leq k < m : x = S.k)
\end{aligned}
$$

□

Teorema 7. *O programa da figura 1 termina.*

Prova.

Os valores de $m - i$ formam uma cadeia finita decrescente porque

(1) *inicialmente $m \geq 0$;*
(2) *m não muda;*
(3) *inicialmente $i =$;*
(4) *i é incrementado a cada passo.*

Logo, eventualmente, $m - i$ e loop termina, mesmo que $x \notin S$.

□

Teorema 8. *O programa da figura 1 termina e computa*

$$R = x \in S$$

Prova.

Provado nos teoremas acima.

□

O que foi feito até aqui? Um programa imperativo para o problema de pesquisa linear foi derivado no sentido de que sua construção obedeceu a uma seqüência que passou por:

- definir claramente o objetivo do mesmo;
- estabelecer claramente suas pré-condições;
- definir uma invariante para o loop da computação;
- estabelecer a condição de progresso e a variável derivada;
- definir a guarda do loop;
- definir o código do programa;

- inverter este roteiro para encontrar a prova de correção do programa.

Note que cada passo de cada uma das provas está lógicamente fundamentado. Ao longo deste curso, só serão aceitas provas apresentadas dessa forma. Construa para todos os outros programas desta nota (e deste curso) provas de correção que apresentem o mesmo rigor.

Exercício 2. *Volte à idéia da relaxação: se for possível assumir que x ocorre na seqüência, pode-se contruir algoritmo para encontrar a posição de sua ocorrência em até m comparações. Use essa idéia para derivar um algoritmo que faz no máximo $m + 2$ comparações para determinar*

$$(0 \leq i < m \wedge x = S.i) \textbf{ cor } (i = m \wedge x \notin S)$$

2.2. **Pesquisa bidimensional.** Considere agora que uma seqüência bidimensional é dada: $S(0..m-1, 0..n-1)$. O objetivo consiste em determinar outra vez

$$R = \exists i, j : 0 \leq i < m \wedge 0 \leq j < n : S.i.j = x$$

2.3. **Pesquisa em seqüência ordenada.** Dados $x \in Int$ e $S.0..m-1 \in Int$ e $\forall i, j : 0 \leq i < j < m : S.i \leq S.j$, determine

$$present = \exists i : 0 \leq i < m : x = S.i$$

O algoritmo exposto a seguir é apresentado por Dijkstra em [3].[2]

Para efeito de desenvolvimento, parece razoável relaxar o objetivo e assuma que a seqüência se estende até a posição e que $x < S.m$.

Por outro lado, dado que o objetivo consiste em computar um quantificador existencial, parece intuitivo usar uma leitura construtiva desse quantificador e propor construir um i tal que:

- $S.i = x$ determina a presença de x em S;
- $S.i \neq x$ garante que $x \notin S$.

O objetivo de determinar o valor de *present* passa a ser construído em duas etapas:

- construir um i tal que $S.i \leq x < S.i + 1$, proposição
- estabelecer o valor de *present*

$$present := S.i = x$$

Um primeiro refinamento para o algoritmo aparece na figura 2.

No estado inicial é válido que

$$S.0 \leq x < S.m$$

[2]Ao ler estas notas você estará lendo minha interpretação, o que certamente não deveria ser substituto para a leitura do original.

```
binsearch(x, S.0..m−1, present) is
{x ∈ Int ∧ m>0 ∧ S.0..m − 1 ∈ Int ∧ ordered.S(0..m − 1)}

⟦ Var i ∈ Int

    establish R
    ; present := x=S.i

⟧
```

FIGURA 2. Pesquisa binária: primeiro refinamento

```
binsearch(x, S.0..m−1, present) is
{x ∈ Int ∧ m>0 ∧ S.0..m − 1 ∈ Int ∧ ordered.S(0..m − 1)}

⟦ Var i ∈ Int
  initially i=1∧j=m

  do
    B → shrink(i,j)
  od
  {P ∧ ¬B}
  {R}
  present := x=S.i
⟧
```

FIGURA 3. Pesquisa binária: segundo refinamento

Generalizando esta proposição produzimos a invariante

$$P \triangleq S.i \leq x < S.j \wedge 0 \leq i < j \leq m$$

O objetivo do algoritmo consiste em reduzir a seção entre i e j até que j seja $i+1$, quando a condição R estará estabelecida. A variável derivada será

$$T \triangleq j - i$$

e a guarda do loop

$$B \triangleq j \neq i+1$$

O novo algoritmo aparece na figura 3.

```
shrink(i,j) is
[[ let h :- i<h<j
   in if
         x < S.h → j:=h
       [] S.h ≤ x → i:=h
       fi
   tel
]]
```

FIGURA 4. Pesquisa binária: shrink

A seção entre i e j pode ser reduzida por:

- redução de j para algum h tal que
$$x < S.h \wedge i \leq h < j$$
que mantém a validade da invariante;
- incremento de i para algum h tal que
$$S.h \leq x \wedge i < h \leq j$$
que também mantém a validade da invariante.

Se existem duas ou mais alternativas para a mesma ação, considere um comando alternativo: **if**. Veja o algoritmo de *shrink* na figura 4.

Duas construções importantes se destacam na especificação de *shrink*: o uso do comando alternativo e o uso da atribuição não determinística.

O comando alternativo da linguagem de comandos guardados de Dijkstra é não-determinístico. O trecho da figura 5 define que:

- ao menos um dos B_i $(0 \leq i \leq n)$ deve ser verdadeiro;
- se mais de um B_i for válido, um deles será selecionado sendo a ação S_i respectiva executada.

Provar a correção desse comando demanda provar que:

$$(\exists i : 0 \leq i \leq n : B_i) \wedge (\forall i : 0 \leq i \leq n : Q \wedge B_i \Rightarrow P$$

A idéia de usar atribuições não determinísticas em especificações de programas parece ter sido proposta originialmente por Morris, ver por exemplo [10]. A semântica dessa atribuição é dada pelo seguinte cálculo de pré-condição:

$$\mathbf{wp}(z \; \mathfrak{Q}, P) = (\forall x : Q.x \Rightarrow P)$$

Nessa especificação, o valor de h é escolhido arbitrariamente de um range válido. No entanto, se o valor de h for a média aritmética de i e j, cada ciclo do loop reduzirá a seção do array a considerar pela

$\{Q\}$
if
B_0 -> S_0
[]B_1 -> S_1
...
[]B_n -> S_n
fi $\{P\}$

FIGURA 5. Comando if

```
binsearch(x, S.0..m-1, present) is
{x ∈ Int ∧ m>0 ∧ S.0..m − 1 ∈ Int ∧ ordered.S(0..m − 1)}
[[ Var i,j,h ∈ Int
   initially i=0∧j=m
   do {P}
   j ≠ i+1 → {P ∧ B}
             let h = (i+j)/2
              in {P ∧ B ∧ i < h < j}
                 if x < S.h → j:=h
                                {P}
                  [] S.h≤x → i:=h
                              {P}
             fi {P}
   od
   {P ∧ ¬B}
   {R}
   present:= x=S.i
]]
```

FIGURA 6. Pesquisa binária

metade: o que significa esgotar o array em $O(\lg m)$ passos. O algoritmo completo segue na figura 6.

Considere também nas provas a seguir que:

$$B_0 \triangleq x < S.h$$

$$B_1 \triangleq S.h \leq x$$

e que:

$$S_0 \stackrel{\triangle}{=} j := h$$

$$S_1 \stackrel{\triangle}{=} i := h$$

Assuma que

$$S.0 \leq x < S.m \wedge S(m-1) < S.m$$

e prove os teoremas a seguir. Os teoremas 19 e 18 consideram os casos em que $x < S.0$ e $x > S.m - 1$.

Teorema 9. *Prove que*

$$P \wedge B \Rightarrow \mathbf{wp}(h := (i+j)/2, i < h < j)$$

Prova.

Trivial: aritmética.

□

O teorema a seguir estabelece que, em qualquer ciclo em que P e B são válidos, o comando alternativo não bloqueia.

Teorema 10. *Prove que*

$$P \wedge B \Rightarrow (B_0 \vee B_1)$$

Prova.

Trivial. B_0 é a negação de B_1: a implicação é sempre verdadeira.

□

Teorema 11. *Prove que P vale inicialmente.*

Prova.

Exercício.

□

Os teoremas a seguir provam que o comando alternativo preserva a invariante do loop.

Teorema 12. *Prove que*

$$P \wedge B \wedge i < h < j \vdash B_0 \Rightarrow \mathbf{wp}(S_0, P)$$

Prova.

$$\begin{aligned} &P \wedge B \wedge i < h < j \vdash B_0 \Rightarrow \mathbf{wp}(S_0, P) = \quad \langle \textit{Teorema da dedução} \rangle \\ &\vdash P \wedge B \wedge i < h < j \wedge B_0 \Rightarrow \mathbf{wp}(S_0, P) \end{aligned}$$

□

O objetivo, então, consistirá em provar que:

$$P \wedge B \wedge i < h < j \wedge B_0 \Rightarrow \mathbf{wp}(S_0, P)$$

$$\begin{aligned}
& \mathbf{wp}(S_{,)}\, 0P \\
=\ & \\
& 0 \leq i < h \leq m \wedge S.i \leq x < S.h \\
=\langle & \textit{lógica} \rangle \\
& 0 \leq i \wedge i < h \wedge h \leq m \wedge S.i \leq x \wedge x < S.h \\
\Leftarrow\langle & \textbf{Eliminação de } \wedge \rangle \\
& 0 \leq i \wedge i < h \wedge h < j \wedge j \leq m \wedge S.i \leq x \wedge x < S.h \\
& = \langle \textit{trasitividade de } < \rangle \\
& 0 \leq i < j \leq m \wedge h < j \wedge S.i \leq x \wedge x < S.h \\
\Leftarrow\langle & h < j \textit{ e } S \textit{ ordenado} \rangle \\
& 0 \leq i < j < m \wedge S.i \leq x \wedge x < S.j \wedge x < S.h \wedge i < h < j \\
= & \\
& P \wedge B \wedge i < h < j \wedge B_0
\end{aligned}$$

Teorema 13. *Prove que*

$$P \wedge B \wedge i < h < j \vdash B_1 \Rightarrow \mathbf{wp}(S_1, P)$$

Prova.

Parecido com anterior. Exercício.

□

Teorema 14. *Prove que P é uma invariante do loop do algoritmo da figura 6.*

Prova.

P vale inicialmente, de acordo com teorema 11, e validade de P é mantida pelo loop, segundo teorema 12.

□

Teorema 15. *Prove que*

$$P \wedge B \Rightarrow T > 0$$

Prova.

Exercício.

□

Teorema 16. *Prove que T decresce, provando que*

$$P \wedge B \wedge i < h < j \vdash \forall v : T = v \Rightarrow \mathbf{wp}(S_0, T > v)$$

e que

$$P \wedge B \wedge i < h < j \vdash \forall v : T = v \Rightarrow \mathbf{wp}(S_0, T > v)$$

Prova.

Exercício.

□

Teorema 17. *Prove que o loop do algoritmo da figura 6 termina.*

Prova.

Exercício.

□

Teorema 18. *Assuma que $x > S(m-1)$ e prove que P ainda é uma invariante do algoritmo da figura 6.*

Prova.

Exercício.

□

Teorema 19. *Assuma que $x < S.0$ e prove que*

$$i = 0 \wedge x < S.0$$

é uma invariante do loop do algoritmo da figura 6.

Prova.

Exercício.

□

Teorema 20. *Prove que o algoritmo estabelece*

$$present = \exists k : 0 \leq k < m : S.k = x$$

Prova.

Exercício.

□

Algumas observações finais sobre o algoritmo de pesquisa binária:

- A derivação do algoritmo é feita passo a passo baseada em condições logicamente consistentes: o resultado inevitável é um algoritmo correto. Na verdade, a metodologia de programação de Dijkstra dá relevância a esse processo de derivação do algoritmo ficando a prova em segundo plano: por exemplo, [3] não contém a prova de correção do algoritmo final.

- A prova de correção levou em conta três: x ser menor do que $S.0$, x estar entre $S.0$ e $S.m-1$ e x ser maior do que $S.m-1$. É possível, no entanto, estabelecer um invariante mais forte para o mesmo algoritmo e construir provas para x genérico.
- Embora a prova assuma que $S.m > S.m-1$, $S.m$ não é usado no algoritmo. *No segmentation fault!*

3. Operações sobre os elementos de uma seqüência

3.1. **Soma dos elementos de uma seqüência.** Dada uma seqüência S de m inteiros, calcule a soma dos elementos de S.

O objetivo consiste em estabelecer:

$$R = \sum_{0 \le i < m} S.i$$

3.1.1. *Solução funcional.* A solução funcional é dada a partir da derivação a seguir:

$$\begin{aligned} R &= \sum i : 0 \le i < m : S.i \\ &= (S.0 + S.1 + \cdots + S.m - 1) \\ &= ((\cdots((S.0 + S.1) + S.2) + \cdots) + S.m - 1) \\ &= ((\cdots((0 + S.0) + S.1) + S.2) + \cdots) + S.m - 1) \end{aligned}$$

Que é calculado em Haskell, assumindo que S é agora representado por uma lista, por

```
foldl (+) 0 S
```

As definições de *foldl* e *foldr* são padrão:

```
foldl op z [] = z
foldl op a (x:xs) = foldl op (op a x) xs

foldr op z [] = z
foldr op a (x:xs) = op x (foldr op a xs)
```

Note que a associatividade à esquerda da operação (+), implementada através do combinador *foldl*, pode ser reescrita como associatividade à direita, implementada via *foldr*.

```
foldr (+) 0 S
```

que, por sua vez, implementa o padrão recursivo da função *seqsum* a seguir:

```
seqsum [] = 0
seqsum (x:xs) = x+(seqsum xs)
```

Nessa função fique mais explícita a complexidade da soma dos elementos de uma seqüência. Ela demanda:

- m operações de soma;
- $m + 1$ testes de fim de seqüência;
- m operações para recuperar o *tail* da seqüência atual.

3.1.2. *Solução imperativa.* A solução imperativa é facilmente obtida de uma generalização do output desejado

$$R = \sum_{0 \leq i < m} S.i$$

a partir da obervação da associatividade à esquerda do operador $(+)$:

- ao final do loop, m elementos terão sido acumulados produzindo o resultado desejado;
- em um ciclo k, os k primeiros elementos terão sido acumulados.

Na fórmula de R, generaliza-se a constante M para uma variável k e insere-se a restrição de que k deve ser maior ou igual a zero elementos e menor ou igual a m elementos.

$$P = (0 \leq k \leq m \wedge R = \sum_{0 \leq i < k} S.i)$$

P será obviamente a invariante do loop. A guarda do loop deve indicar quantos elementos ainda devem ser acumulados:

$$B = (m - k \neq 0)$$

o que dá a variável derivada

$$T = m - k$$

e a condição inicial

$$\textbf{initially } k = 0 \wedge R = 0$$

A derivação final do programa consiste apenas em definir o comando guardado do loop que decrescerá T, mantendo a invariante P. Fica como exercício.

Exercício 3. *Prove que o programa derivado para cálculo da soma dos elementos de uma seqüência está correto.*

(1) *Prove que ele está parcialmente correto.*

(a) *Prove que P é válido inicialmente.*
(b) *Prove que $P \wedge B \mathrel{.\Rightarrow} \mathbf{wp}(\alpha, P)$, onde α é o comando guardado do loop.*
(c) *Prove que $P \wedge \neg B \mathrel{.\Rightarrow} R$*

(2) *Prove que o loop termina.*
(a) *Prove que $P \wedge B \mathrel{.\Rightarrow .} T > 0$*
(b) *Prove que eventualmente $T = 0$ e o loop termina.*

4. Permutações de uma seqüência

Esta seção aborda algoritmos que tratam de variantes de um mesmo problema: calcular alguma permutação de uma seqüência.

4.1. **Section swap.** Dado um vetor $S.0..m-1 \in Int$ e dados inteiros p, q, n tais que

$$n > 0 \wedge 0 \leq p \leq p+n \leq q \leq q+n \leq m$$

calcule in-locu uma permutação de S que consiste em trocar as seções[3] $S.p..p+n-1$ por $S.q..q+n-1$. Assumindo que B contém o valor inicial de S, o objetivo consiste em calcular [4]

$$\begin{aligned} R \stackrel{\triangle}{=} & S(0..p-1) = B(0..p-1) \\ & \wedge S(p..p+n-1) = B(q..q+n-1) \\ & \wedge S(p+n..q-1) = B(p+n..q-1) \\ & \wedge S(q..q+n-1) = B(p..p+n-1) \\ & \wedge S(q+n..m-1) = B(q+n..m-1) \end{aligned}$$

A invariante P do algoritmo para calcular o swap de seções do array é obtida generalizando R, via troca da constante n por uma variável k, que deve ser um inteiro entre 0 e n.

$$\begin{aligned} P \stackrel{\triangle}{=} & 0 \leq k \leq n \\ & \wedge S(0..p-1) = B(0..p-1) \\ & \wedge S(p..p+k-1) = B(q..q+k-1) \\ & \wedge S(p+k..q-1) = B(p+k..q-1) \\ & \wedge S(q..q+k-1) = B(p..p+k-1) \\ & \wedge S(q+k..m-1) = B(q+k..m-1) \end{aligned}$$

[3] Exemplo retirado de [7].
[4] B é introduzido aqui como variável auxiliar para facilitar a derivação do programa.

```
swapsec(S,p,q,n) is
{m ≥ 0 ∧ S.0..m − 1 ∈ Int ∧ p, q, n ∈ Int ∧ n ≥ 0}
⟦ Var k ∈ Int;
    initially B=S ∧ i=0 ∧ m≥0 ∧ n≥0
    {P}
    do
      n−k>0 → S(p+k),S(q+k),k:=S(q+k),S(p+k),k+1
    od
    {P ∧ ¬B}

  {R}
⟧
```

FIGURA 7. Array swap

A variável derivada é óbvia:

$$T \triangleq n - k$$

Essa variável deve decrescer até atingir o valor 0, o que dá a guarda do loop:

$$B_0 \triangleq n - k > 0$$

Progresso no loop demanda reduzir T, via incremento de k, mantendo a validade de P. A ação associada à transição do loop, facilmente derivada a partir de P e do incremento de k, fica:

$$S_0 \triangleq S(p+k), S(q+k), k := S(q+k), S(p+k), k+1$$

O algoritmo está na figura 7.

Exercício 4. *Prove que:*

(1) $P \wedge \Rightarrow T > 0$
(2) *O programa da figura 7 termina.*
(3) $P \wedge B_0 \Rightarrow \mathbf{wp}(S_0, P)$
(4) P *é uma invariante do loop do programa da figura 7.*
(5) $P \wedge \neg B_0 \Rightarrow R$

O primeiro objetivo consiste em provar que a execução de S_0 não altera a validade de P. Antes, porém, observe que, após S_0, um novo S, que chamaremos de S', é produzido.

$$S' \triangleq (S; p+k : S(q+k); q+k : S(p+k))$$

Note que S' define os novos valores de S e que as igualdades abaixo de verificam e elas mostram que seções de S não foram alteradas.

$$\begin{aligned} S'(0..p+k-1) &= S(0..p+k-1) \\ S'(p+k+1..q+k-1) &= S(p+k+1..q+k-1) \\ S'(q+k+1..m-1) &= S(q+k+1..m-1) \end{aligned}$$

As igualdades a seguir mostram as posições de S que foram alteradas:

$$\begin{aligned} S'(p+k) &= S(q+k) \\ S'(q+k) &= S(p+k) \end{aligned}$$

Além disso, observe que

$$\begin{aligned} S(q+k) &= B(q+k) \\ S(p+k) &= B(p+k) \end{aligned}$$

Esses dois conjutnos de igualdades são usadas na prova do teorema a seguir.

Teorema 21. $P \wedge B_0 \Rightarrow \mathbf{wp}(S_0, P)$

Prova.

$$
\begin{aligned}
&\mathbf{wp}(S_0, P)\\
=&-1 \leq k \leq n-1\\
&\wedge S'(0..p-1) = B(0..p-1)\\
&\wedge S'(p..p+k) = B(q..q+k)\\
&\wedge S'(p+k+1..q-1) = B(p+k+1..q-1)\\
&\wedge S'(q..q+k) = B(p..p+k)\\
&\wedge S'(q+k+1..m-1) = B(q+k+1..m-1)\\
=&-1 \leq k \leq n \wedge k \neq n\\
&\wedge S'(0..p-1) = B(0..p-1)\\
&\wedge S'(p..p+k-1) = B(q..q+k-1)\\
&\wedge S'(p+k) = B(q+k)\\
&\wedge S'(p+k..q-1) = B(p+k..q-1)\\
&\wedge S'(q..q+k-1) = B(p..p+k-1)\\
&\wedge S'(q+k) = B(p+k)\\
&\wedge S'(q+k..m-1) = B(q+k..m-1)\\
=&\langle \textit{subsituindo } S' \rangle\\
&-1 \leq k \leq n \wedge k \neq n\\
&\wedge S(0..p-1) = B(0..p-1)\\
&\wedge S(p..p+k-1) = B(q..q+k-1)\\
&\wedge B(q+k) = B(q+k)\\
&\wedge S(p+k..q-1) = B(p+k..q-1)\\
&\wedge S(q..q+k-1) = B(p..p+k-1)\\
&\wedge B(p+k) = B(p+k)\\
&\wedge S(q+k..m-1) = B(q+k..m-1)\\
=&\langle \textit{simplificando } \wedge \rangle\\
&-1 \leq k \leq n \wedge k \neq n\\
&\wedge S(0..p-1) = B(0..p-1)\\
&\wedge S(p..p+k-1) = B(q..q+k-1)\\
&\wedge S(p+k..q-1) = B(p+k..q-1)\\
&\wedge S(q..q+k-1) = B(p..p+k-1)\\
&\wedge S(q+k..m-1) = B(q+k..m-1)\\
\Leftarrow&\langle \textbf{Introdução de } \vee \rangle\\
&0 \leq k \leq n \wedge k \neq n\\
&\wedge S(0..p-1) = B(0..p-1)\\
&\wedge S(p..p+k-1) = B(q..q+k-1)\\
&\wedge S(p+k..q-1) = B(p+k..q-1)\\
&\wedge S(q..q+k-1) = B(p..p+k-1)\\
&\wedge S(q+k..m-1) = B(q+k..m-1)\\
=&P \wedge B_0
\end{aligned}
$$

□

4.2. **Reverso de uma seqüência.** Dada uma seqüência, cálcule seu reverso.

4.2.1. *A lógica.* Gries e Schneider, em [7], apresentam o seguinte conjunto de axiomas para definir o reverso de um seqüência.

$$rev.\varepsilon = \varepsilon \tag{6}$$

$$rev.c \lhd x = (rev.x)^\frown\langle c\rangle \tag{7}$$

onde

$$\varepsilon^\frown y = y \tag{8}$$

$$(c \lhd x)^\frown y = c \lhd (x^\frown y) \tag{9}$$

A solução funcional, em Haskell, para o problema do reverso de uma seqüência é retirada diretamente dos axiomas 6 e 7.

Talvez seja mais intuitivo ler y como reverso de x quando: o último elemento de x é o primeiro de y, o primeiro de x é último em y e a subseqüência que vai do segundo ao penúltimo elemento de y é o reverso da subseqüência que vai do segundo ao penúltimo elemento de x.

$$reverse.\langle\rangle = \langle\rangle \tag{10}$$

$$reverse.\langle a\rangle = \langle a\rangle \tag{11}$$

$$reverse(a \lhd x \rhd b) = b \lhd (reverse.x) \rhd a \tag{12}$$

onde (13)

$$x \rhd a = x^\frown\langle a\rangle \tag{14}$$

Exercício 5. *Prove que:*

(1) $rev.\langle a\rangle^\frown x = (rev.x)^\frown\langle a\rangle$
(2) $rev.x^\frown\langle a\rangle = \langle a\rangle^\frown rev.x$
(3) $rev.\langle a\rangle^\frown x^\frown\langle b\rangle = \langle b\rangle^\frown(rev.x)^\frown\langle a\rangle$

A última prova do exercício 5 fornece a base a definição da invariante do algoritmo imperativo para cálculo do reverso de uma seqüência.

4.2.2. *Solução imperativa.* A invariante do algoritmo para cálculo do reverso de uma seqüência surge da generalização do teorema provado no exercício 5.

Exercício 6. *Prove que*

(1) $rev.x^\frown y = (rev.y)^\frown(rev.x)$
(2) $rev.x^\frown w^\frown y^\frown z = (rev.z)^\frown(rev.w^\frown y)^\frown(rev.x)$

Uma definição alternativa para o reverso de uma seqüência, mais adequada ao tratamento de seqüência como array.

Definição 3.

$$\begin{aligned} S.i..j = rev(B.h..k) \stackrel{\triangle}{=} \\ j - i = k - h \\ \wedge.\ j - i < 0 \\ \vee (S.i = B.k \wedge S(i+1..j) = rev(B.h..k-1)) \end{aligned}$$

Exercício 7. *Apresente uma definição formal de seção de uma seqüência S e prove que para qualquer inteiro i não negativo, menor do que zero:*

$$\begin{aligned} rev.S(0..m-1) =& (rev.S(m-1-i..m-1)) \\ & \hat{} (rev.S(i..m-1-i)) \\ & \hat{} (rev.S(0..i)) \end{aligned}$$

Exercício 8. *Apresente uma algoritmo para cálculo do reverso de uma seqüênciae prove sua correção.*

(1) *Use teorema provado no exercício 7 para definir a invariante P do loop do algoritmo.*
(2) *Use a variável derivada*

$$T = m - 2i$$

(3) *Define a guarda B do loop e prove que*

$$P \wedge B \ .\Rightarrow.\ \mathbf{wp}(\alpha, P)$$

onde α é

$$S.i, S(m-1-i), i := S(m-1-i), S.i, i+1$$

4.3. **Partição de uma seqüência por um pivô.** Dada uma seqüência S e um elemento indicado como pivô, o objetivo consiste em particionar a seqüência original em duas seqüências: de elementos menores do que o pivôe de elementos maiores do que o pivô. Acrescentada a restrição de que a seqüência seja representada em um vetor e de que as trocas devem ser realizadas no próprio vetor, evitando a geração de seqüências adicionais, o objetivo consiste em encontrar uma permutação de S e um valor p tal que

$$R \stackrel{\triangle}{=} 0 \leq p < m \wedge S(0..p-1) < S.p \wedge S(p+1..m-1) \geq S.p$$

Dado que a posição inicial do pivô não foi especificada, é razoável arbitrar que ele se encontrar inicialmente na posição 0 e que o objetivo consiste em estabelecer para algum i que

$$S(0..i-1) < S.0 \wedge S(i..m-1) > S.0$$

para, então, computar R.

Note que se

$$\beta \stackrel{\triangle}{=} S.0, S(i-1), p := S(i-1), S.0, i-1$$

então

$$\begin{aligned}
&\mathbf{wp}(\beta, R)\\
=&0 \leq i-1 < m\\
\wedge&S'(0..i-1-1) < S'(i-1)\\
\wedge&S'(i-1+1..m-1) \geq S'(i-1)\\
\wedge&i-1 = i-1\\
=&\langle \text{aritmética} \rangle\\
\wedge&1 \leq i \leq m\\
\wedge&S'(0..i-1-1) < S'(i-1)\\
\wedge&S'(i-1+1..m-1) \geq S'(i-1)\\
=&1 \leq i \leq m\\
\wedge&S'(0) < S'(i-1) \wedge S'(1..i-2) < S'(i-1)\\
\wedge&S'(i..m-1) \geq S'(i-1)\\
=&\langle S' = (S; 0 : S(i-1); i-1 : S.0) \rangle\\
&1 \leq i \leq m\\
\wedge&S(i-1) < S(0) \wedge S(1..i-2) < S(0)\\
\wedge&S(i..m-1) \geq S(0)\\
=&1 \leq i \leq m\\
\wedge&S(1..i-1) < S(0)\\
\wedge&S(i..m-1) \geq S(0)
\end{aligned}$$

A precondição derivada define o estado final do loop que construirá a permutação de S:

$$1 \leq i \leq m \wedge S(1..i-1) < S(0) \wedge S(i..m-1) \geq S(0)$$

Uma generalização dessa condição, substituindo m por j, define parte da invariante:

$$1 \leq i \leq m \wedge S(1..i-1) < S(0) \wedge S(i..j-1) \geq S(0)$$

A invariante é obtida fortalecendo essa condição pela observação de que:

- os elementos menores do que $S(0)$ estão compreendidos entre as posições 1 e $i-1$;
- os elementos restantes estão compreendidos entre as posições i e $j-1$, o que impõem $i \leq j$, devendo j ficar limitado a m.

$$P \stackrel{\triangle}{=} 1 < i \leq j \leq m \wedge S(1..i-1) < S(0) \wedge S(i..j-1) \geq S(0)$$

O loop analisará cada elemento da seqüência da esquerda para direita, incorporando a cada transição um elemento uma das partições e reduzindo a secão não particionada do vetor, que é $S(j..m-1)$. Sua variável derivada

$$T \stackrel{\triangle}{=} (m-j)$$

A guarda do loop

$$B \stackrel{\triangle}{=} m - j \neq 0$$

O progresso do loop em direção ao ponto-fixo demanda decrementar T, que só pode ocorrer incrementando j, já que m é constante. Por outro lado, o incremento de j pode ocorrer sem perturbar invariante em duas condições:

- quando B_0 é válido

  $$B_0 \stackrel{\triangle}{=} S.j \geq S.0$$

  ;
- quando B_1 é válido

  $$B_1 \stackrel{\triangle}{=} S.j < S.0$$

  que demanda incorporar $S.j$ à partição dos elementos menores do que $S.0$.

A figura 8 apresenta o algoritmo, que Bentley em [1] atribui a Lomut. Note que cada comando apresenta suas precondições e poscondições detalhadas. As ações S_0 e S_1 são facilmente derivadas:

$$S_0 \stackrel{\triangle}{=} j := j+1$$

$$S_1 \stackrel{\triangle}{=} S.i, S.j, i, j := S.j, S.i, i+1, j+1$$

O próximo passo consiste em provar a correção do algoritmo da figura 8.

Teorema 22. *Prove que P é válida inicialmente.*

Prova.

```
partition(S,p) is
{S.0..m − 1 ∈ Int ∧ m ≥ 0}

[[ Var B.0..m−1 ∈ Int; i,j ∈ Int;
   initially B=S ∧ i=1 ∧ j=1
   do {P}
     B → if {P ∧ B}
           B0 → {P ∧ B ∧ B0}S0{P}
         [] B1 → {P ∧ B ∧ B1}S1{P}
         fi {P}
   od
   {P ∧ ¬B}
   S.0,S(i−1),p:=S(i−1),S.0,i−1
   {R}
]]
```

FIGURA 8. Partição de Lomut

Exercício.

□

O teorema a seguir estabelece que quando a invariante P e a guarda do loop são válidas, ao menos uma das guardas do comando **if** será válida.

Teorema 23. *Prove que*

$$P \wedge B \Rightarrow (B_0 \vee B_1)$$

Prova.

B_0 e B_1 são complementares.

□

O próximo teorema estabelece que a ação associada à primeira guarda do **if**, ação S_0 da guarda B_0, preserva a invariante do loop. Sua prova fica como exercício.

Teorema 24. *Provar que*

$$P \wedge B \wedge B_0 \Rightarrow \mathbf{wp}(S_,)\, 0P)$$

Prova.

Exercício: veja prova a seguir.

□

O teorema a seguir estabelece que a ação S_1, associada à guarda B_1 preserva a invariante P.

Teorema 25. *Provar que*

$$P \wedge B \wedge B_1 \Rightarrow \mathbf{wp}(S_{,)}\, 1P)$$

Prova.

$$
\begin{array}{ll}
 & \mathbf{wp}(S_1, P) \\
= & \\
 & 1 < i+1 \leq j+1 \leq m \\
 & \wedge\, S'(1..i+1-1) < S'(0) \wedge S'(i+1..j+1-1) \geq S'(0) \\
= & \\
 & -1 < i \leq j \leq m-1 \\
 & \wedge\, S'(1..i) < S'(0) \wedge S'(i+1..j) \geq S'(0) \\
= & \\
 & 0 \leq i \leq j \leq m \\
 & \wedge\, (m \neq j \wedge S'(1..i) < S'(0) \wedge S'(i+1..j) \geq S'(0)) \\
= & \\
 & (0 = i \vee 0 < i \leq j \leq m) \\
 & \wedge\, (m \neq j \wedge S'(1..i) < S(0) \wedge S'(i+1..j) \geq S'(0)) \\
= & \\
 & (0 = i \wedge m \neq j \wedge S'(1..i) < S'(0) \wedge S'(i+1..j) \geq S'(0)) \\
 & \vee\, (0 < i \leq j \leq m \wedge m \neq j \\
 & \quad \wedge\, S'(1..i) < S'(0) \wedge S'(i+1..j) \geq S'(0)) \\
\Leftarrow & \langle \textbf{Introdução de } lor \rangle \\
 & (0 < i \leq j \leq m) \wedge m \neq j \\
 & \wedge\, S'(1..i) < S'(0) \wedge S'(i+1..j) \geq S'(0) \\
= & \langle \textit{trocando } S' \textit{ por seus valores em } S \rangle \\
 & (0 < i \leq j \leq m) \wedge m \neq j \\
 & \wedge\, S(1..i-1) < S(0) \wedge S(i) \geq S(0) \\
 & \wedge\, S(i+1..j-1) \geq S(0) \wedge S(j) < S(0) \\
= & \\
 & P \wedge B \wedge B_1
\end{array}
$$

□

Estas notas foram completamente alteradas desde a última versão, mas ainda necessitam muitas correções e adições em termos de:

- novos exemplos;
- novos exercícios;
- explanação da semântica da linguagem usada;
- substituição total dos exemplos em Haskell por equivalentes em lógica;
- validações usando algum assistente de prova.

Agradeço contribuições nesse sentido.

Referências

1. Jon Bentley, *Programming pearls*, second ed., Addison-Wesley, 2000.
2. Haskell B. Curry, *Foundations of mathematical logic*, Dover Publications, Inc., 1977.
3. Edsger W. Dijkstra, *Fillers at the YoP intitute*, in *Formal Development of Programs and Proofs* [4].
4. Edsger W. Dijkstra (ed.), *Formal development of programs and proofs*, Addison-Wesley, 1990.
5. E.W. Dijkstra, *A discipline of programming*, Prentice Hall, Englewood Cliffs, 1976.
6. David Gries, *The science of programming*, Springer-Verlag, New York, 1981.
7. ———, *The science of programming*, Springer-Verlag, 1985.
8. David Gries and Fred B.Schneider, *A logical approach to discrete mathematics*, Springer-Verlag, 1993.
9. Donald E. Knuth, *The art of computer programmimg, volume 3: Sorting and searching*, Addison-Wesley, 1998.
10. Joseph M. Morris, *Piecewise data refinment*, in Dijkstra [4].