Programming Languages: Imperative Program Construction 10. Swaps in Arrays

Shin-Cheng Mu

Autumn Term, 2021

1 Swaps

- Extend the notion of function alteration to two entries.

$$\begin{array}{rll}(f : x, y \mapsto e_1, e_2)\ z &= e_1 & \text{, if } z = x, \\ &= e_2 & \text{, if } z = y, \\ &= f\ z & \text{, otherwise.}\end{array}$$

- Given array $h\ [0..N)$ and integer expressions E and F, let $swap\ \ h\ \ E\ \ F$ be a primitive operation such that:

$$\begin{array}{l} wp\ (swap\ h\ E\ F)\ P = def\ (h[E]) \wedge def\ (h[F]) \wedge \\ \quad P[h \backslash (h : E, F \mapsto h[F], h[E])]\ \ . \end{array}$$

- Intuitively, $swap\ \ h\ \ E\ \ F$ means "swapping the values of $h[E]$ and $h[F]$. (See the notes below, however.)

Complications

- $swap\ \ h\ \ E\ \ F$ does not always literally "swaps the values." For example, it is *not* always the case that

$$\{h[E] = X\}\ swap\ h\ E\ F\ \{h[F] = X\}\ \ .$$

- Consider $h[0] = 0 \wedge h[1] = 1$. This does not hold:

$$\{h[h[0]] = 0\}\ swap\ h\ (h[0])\ (h[1])\ \{h[h[1]] = 0\}\ \ .$$

- In fact, after swapping we have $h[0] = 1 \wedge h[1] = 0$, and hence $h[h[1]] = 1$.

A Simpler Case

- However, when h does not occur free in E and F, we do have

$$\begin{array}{l} (\{\langle \forall i : i \neq E \wedge i \neq F : h[i] = H\ i \rangle\} \wedge \\ \quad h[E] = X \wedge h[F] = Y) \\ swap\ h\ E\ F \\ (\{\langle \forall i : i \neq E \wedge i \neq F : h[i] = H\ i \rangle\} \wedge \\ \quad h[E] = Y \wedge h[F] = X)\ \ . \end{array}$$

- It is a convenient rule we use when reasoning about swapping.

Note: Kaldewaij's Swap

- Kaldewaij [Kal90, Chapter 10] defined $swap\ h\ E\ F$ as an abbreviation of

$$|[\ \mathbf{var}\ r; r := h[E]; h[E] := h[F]; h[F] := r\ ]|\quad ,$$

- where r is a fresh name and |[...]| denotes a program block with local constants and variables. We have not used this feature so far.

- I do not think this definition is correct, however. The definition would not behave as we expect if F refers to $h[E]$.

2 The Dutch National Flag

- Let $RWB = \{R, W, B\}$ (standing respecively for red, white, and blue).

$$
\begin{array}{l}
\textbf{con } N : \mathit{Int}\ \{0 \leqslant N\} \\
\textbf{var } h : \textbf{array } [0..N) \textbf{ of } \mathit{RWB} \\
\textbf{var } r, w : \mathit{Int} \\
\mathit{dutch\_national\_flag} \\
\{0 \leqslant r \leqslant w \leqslant N \wedge \\
\quad \langle \forall i : 0 \leqslant i < r : h[i] = R \rangle \wedge \\
\quad \langle \forall i : r \leqslant i < w : h[i] = W \rangle \wedge \\
\quad \langle \forall i : w \leqslant i < N : h[i] = B \rangle \wedge \}
\end{array}
$$

- The program shall manipulate h only by swapping.
- Denote the postcondition by Q.

Invariant

- Introduce a variable b.
- Choose as invariant $P_0 \wedge P_1$, where

$$
\begin{array}{ll}
P_0 & \equiv P_r \wedge P_w \wedge P_b \\
P_1 & \equiv 0 \leqslant r \leqslant w \leqslant b \leqslant N \\
P_r & \equiv \langle \forall i : 0 \leqslant i < r : \ h[i] = R \rangle \\
P_w & \equiv \langle \forall i : r \leqslant i < w : h[i] = W \rangle \\
P_b & \equiv \langle \forall i : b \leqslant i < N : h[i] = B \rangle
\end{array}
$$

- $P_0 \wedge P_1$ can be established by $r, w, b := 0, 0, N$.
- If $w = b$, we get the postcondition Q.

The Plan

$$
\begin{array}{l}
r, w, b := 0, 0, N \\
\{P_0 \wedge P_1, \mathit{bnd} : b - w\} \\
\textbf{do } b \neq w \to \textbf{if } h[w] = R \ \to S_r \\
\qquad\qquad\qquad\ \mid \ h[w] = W \to S_w \\
\qquad\qquad\qquad\ \mid \ h[w] = B \ \to S_b \\
\qquad\qquad\qquad \textbf{fi} \\
\textbf{od} \\
\{Q\}
\end{array}
$$

Observation

- Note that
  - r is the number of red elements detected,
  - $w - r$ is the number of white elements detected,
  - $N - b$ is the number of blue elements detected.
- Therefore, S_w should contain $w := w + 1$, S_b should contain $b := b - 1\cdot$
- S_r should contain $r, w := r + 1, w + 1$, thus r increases but $w - r$ is unchanged.
- The bound decreases in all cases! Good sign.

White

- The case for white is the easiest, since

$$
\begin{array}{l}
P_0 \wedge P_1 \wedge h[w] = W \Rightarrow \\
\quad (P_0 \wedge P_1)[w \backslash w + 1] \ .
\end{array}
$$

- It is sufficient to let S_w be simply $w := w + 1$.

Blue

- We have

$$
\begin{array}{l}
\{P_r \wedge P_w \wedge P_b \wedge w < b \wedge h[w] = B\} \\
\mathit{swap}\ h\ w\ (b - 1) \\
\{P_r \wedge P_w \wedge P_b \wedge w < b \wedge h[b - 1] = B\} \\
b := b - 1 \\
\{P_r \wedge P_w \wedge P_b \wedge w \leqslant b\}
\end{array}
$$

- Thus we choose $\mathit{swap}\ h\ w\ (b - 1); b := b - 1$ as S_b.

Red

- Precondition: $P_r \wedge P_w \wedge P_b \wedge w < b \wedge h[w] = R$.
- It appears that $\mathit{swap}\ h\ w\ r$ establishes $P[w \backslash w + 1]$. But we have to see what $h[r]$ is before we can increment r.
- P_w implies $r < w \Rightarrow h[r] = W$. Equivalently, we have $r = w \vee h[r] = W$.

Red: Case $r = w$

- We have

$$
\begin{array}{l}
\{P_r \wedge P_w \wedge P_b \wedge r = w < b \wedge h[w] = R\} \\
\mathit{swap}\ h\ w\ r \\
\{P_r \wedge P_w \wedge P_b \wedge w < b \wedge h[r] = R\} \\
r, w := r + 1, w + 1 \\
\{P_r \wedge P_w \wedge P_b \wedge r = w \leqslant b\}
\end{array}
$$

Red: Case $h[r] = W$

- We have

$$
\begin{array}{l}
\{P_r \wedge P_w \wedge P_b \wedge w < b \wedge h[r] = W \wedge h[w] = R\} \\
\mathit{swap}\ h\ w\ r \\
\{P_r \wedge h[r] = R \wedge \langle \forall i : r + 1 \leqslant i < w : h[i] = W \rangle \wedge \\
\quad h[w] = W \wedge P_b \wedge w < b\} \\
r, w := r + 1, w + 1 \\
\{P_r \wedge P_w \wedge P_b \wedge r = w \leqslant b\}
\end{array}
$$

- In both cases, $\mathit{swap}\ h\ w\ r; r, w := r + 1, w + 1$ is a valid choice.

The Program

$$
\begin{array}{l}
\mathbf{con}\ N : \mathit{Int}\ \{0 \leqslant N\} \\
\mathbf{var}\ h : \mathbf{array}\ [0..N)\ \mathbf{of}\ \mathit{RWB} \\
\mathbf{var}\ r, w, b : \mathit{Int} \\
r, w, b := 0, 0, N \\
\{P_0 \wedge P_1, \mathit{bnd} : b - w\} \\
\mathbf{do}\ b \neq w \rightarrow \mathbf{if}\ h[w] = R \ \rightarrow \mathit{swap}\ h\ w\ r \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad r, w := r + 1, w + 1 \\
\qquad\qquad\quad \mid\ h[w] = W \rightarrow w := w + 1 \\
\qquad\qquad\quad \mid\ h[w] = B \ \rightarrow \mathit{swap}\ h\ w\ (b - 1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad b := b - 1 \\
\qquad\qquad\quad \mathbf{fi} \\
\mathbf{od} \\
\{Q\}
\end{array}
$$

3 Rotation

Rotation

- Given: $h : \mathbf{array}\ [0..N)\ \mathbf{of}\ A$ with integer constants $0 \leqslant K < N$.

- Task: rotate h over K places. That is, $h[0]$ is moved to $h[K]$, $h[1]$ to $h[(1 + K)\ \mathbf{mod}\ N]$, $h[2]$ to $h[(2 + K)\ \mathbf{mod}\ N]$...

- using swap operations only.

Specification

- $$
\begin{array}{l}
\mathbf{con}\ K, N : \mathit{Int}\ \{0 \leqslant K < N\} \\
\mathbf{var}\ h : \mathbf{array}\ [0..N)\ \mathbf{of}\ A \\
\{\langle \forall i : 0 \leqslant i < N : h[i] = H[i] \rangle\} \\
\mathit{rotation} \\
\{\langle \forall i : 0 \leqslant i < N : h[(i + K)\ \mathbf{mod}\ N] = H[i] \rangle\}\ .
\end{array}
$$

- To eliminate $\mathbf{mod}$, the postcondition can be rewritten as:

$$
\begin{array}{l}
\langle \forall i : 0 \leqslant i < N - K : h[i + K] = H[i] \rangle \wedge \\
\quad \langle \forall i : N - K \leqslant i < N : h[i + K - N] = H[i] \rangle\ .
\end{array}
$$

- Or, $h[K..N) = H[0..N - K) \wedge h[0..K) = H[N - K..N)$.

Abstract Notations

- For this problem we benefit from using more abstract notations.

- Segments of arrays can be denoted by variables. E.g. $X = H[0..N - K)$ and $Y = H[N - K..N)$.

- Concatenation of arrays are denoted by juxtaposition. E.g. $H[0..N) = XY$.

- Empty sequence is denoted by $[\,]$.

- Length of a sequence X is denoted by $l\ X$.

- Specification:

$$
\begin{array}{l}
\{h = XY\} \\
\mathit{rotation} \\
\{h = YX\}
\end{array}
$$

- When $l\ X = l\ Y$ we can establish the postcondition easily — just swap the corresponding elements.

- Denote swapping of equal-lengthed array segments by $\mathit{SWAP}\ X\ Y$.

Thinking Lengths

- When $l\ X < l\ Y$, h can be written as $h = XUV$,

- where $l\ U = l\ X$ and $UV = Y$.

- Task:

$$
\begin{array}{l}
\{h = XUV \wedge l\ U = l\ X\} \\
\mathit{rotation} \\
\{h = UVX\}
\end{array}
$$

- Strategy:

$$
\begin{array}{l}
\{h = XUV \wedge l\ U = l\ X\} \\
\mathit{SWAP}\ X\ U \\
\{h = UXV\} \\
?? \\
\{h = UVX\}
\end{array}
$$

- The part $??$ shall transform XV into VX — a problem having the same form as the original!

- Some (including myself) would then go for a recursive program. But there is another possibility.

Leading to an Invariant...

- Consider the symmetric case where $l\ X > l\ Y$.

$$
\begin{array}{l}
\{h = UVY \wedge l\ V = l\ Y\} \\
SWAP\ V\ Y \\
\{h = UYV\} \\
?? \\
\{h = YUV\}
\end{array}
$$

- In general, the array is of them form $AUVB$, where UV needs to be transformed into VU, while A and B are parts that are done.

The Invariant

- Strategy:

$$
\begin{array}{l}
\{h = XY\} \\
A, U, V, B := [\,], X, Y, [\,] \\
\{h = AUVB \wedge YX = AVUB, bnd : l\ U + l\ V\} \\
\mathbf{do}\ U \neq [\,] \wedge V \neq [\,] \rightarrow ...\mathbf{od} \\
\{h = YX\}
\end{array}
$$

- Call the invariant P. Intuitively it means "currently the array is $AUVB$, and if we exchange U and V, we are done."
- Note the choice of guard: $P \wedge (U = [\,] \wedge V = [\,]) \Rightarrow h = YX$.

An Abstract Program

$$
\begin{array}{l}
A, U, V, B := [\,], X, Y, [\,] \\
\{h = AUVB \wedge YX = AVUB, bnd : l\ U + l\ V\} \\
\mathbf{do}\ U \neq [\,] \wedge V \neq [\,] \rightarrow \\
\quad \mathbf{if}\ l\ U \geqslant l\ V \rightarrow \quad \text{-- } l\ U_1 = l\ V \\
\qquad \{h = AU_0U_1VB \wedge YX = AVU_0U_1B\} \\
\qquad SWAP\ U_1\ V \\
\qquad \{h = AU_0VU_1B \wedge YX = AVU_0U_1B\} \\
\qquad U, B := U_0, U_1B \\
\qquad \{h = AUVB \wedge YX = AVUB\} \\
\quad \mid\ l\ U \leqslant l\ V \rightarrow \quad \text{-- } l\ V_0 = l\ U \\
\qquad \{h = AUV_0V_1B \wedge YX = AV_0V_1UB\} \\
\qquad SWAP\ U\ V_0 \\
\qquad \{h = AV_0UV_1B \wedge YX = AV_0V_1UB\} \\
\qquad A, V := AV_0, V_1 \\
\qquad \{h = AUVB \wedge YX = AVUB\} \\
\quad \mathbf{fi} \\
\mathbf{od}
\end{array}
$$

Representing the Sequences

- Introduce $a, b, k, l : Int$.
- $A = h[0..a)$;
- $U = h[a..a + k)$, hence $l\ U = k$;
- $V = h[b - l..n)$, hence $l\ V = l$;
- $B = h[b..N)$.
- Additional invariant: $a + k = b - l$.
- Why having both k and l? We will see later.

A Concrete Program

- Represented using indices:

$$
\begin{array}{l}
a, k, l, b := 0, N - K, K, N \\
\mathbf{do}\ k \neq 0 \wedge l \neq 0 \rightarrow \\
\quad \mathbf{if}\ k \geqslant l \rightarrow SWAP\ (b - l)\ l\ (-l) \\
\qquad\qquad\quad k, b := k - l, b - l \\
\quad \mid\ k \leqslant l \rightarrow SWAP\ a\ k\ k \\
\qquad\qquad\quad a, l := a + k, l - k \\
\quad \mathbf{fi} \\
\mathbf{od}
\end{array}
$$

- where $SWAP\ x\ num\ off$ abbreviates

$$
\begin{array}{l}
|[\ \mathbf{var}\ n : Int \\
\quad n := x \\
\quad \mathbf{do}\ n \neq x + num \rightarrow swap\ h\ n\ (n + off) \\
\qquad\qquad\qquad\qquad n := n + 1 \\
\quad \mathbf{od} \\
]|
\end{array}
$$

- that is, starting from index x, swap num elements with those off positions away.

Greatest Common Divisor

- To find out the number of swaps performed, we use a variable t to record the number of swaps.
- If we keep only computation related to t, k, and l:

$$
\begin{array}{l}
k, l, t := N - K, K, 0 \\
\mathbf{do}\ k \neq 0 \wedge l \neq 0 \rightarrow \\
\quad \mathbf{if}\ k \geqslant l \rightarrow t := t + l; k := k - l \\
\quad \mid\ k \leqslant l \rightarrow t := t + k; l := l - k \\
\quad \mathbf{fi} \\
\mathbf{od}
\end{array}
$$

- Observe: the part concerning k and l resembles computation of greatest common divisor.
- In fact, $gcd\ k\ l = gcd\ N\ (N - K)$, which is $gcd\ N\ K$.
- When the program terminates, $k + l = gcd\ N\ K$.
- It's always true that $t + k + l = N$.
- Therefore, the total number of swaps is $t = N - (k + l) = N - gcd\ N\ K$.

References

[Kal90] A. Kaldewaij. *Programming: the Derivation of Algorithms*. Prentice Hall, 1990.