Programming Languages:
Imperative Program Construction
8. Case Studies

Shin-Cheng Mu

Autumn Term, 2024

1 Faster Division

Quotient and Remainder

- Recall the problem:

$$\begin{array}{l}\mathbf{con}\ A, B : \mathit{Int}\ \{0 \leqslant A \land 0 < B\} \\ \mathbf{var}\ q, r : \mathit{Int} \\ ? \\ \{A = q \times B + r \land 0 \leqslant r < B\}\ .\end{array}$$

- Recall: recognising the postcondition as a conjunction, we use $A = q \times B + r \land 0 \leqslant r$ as the invariant and $\neg\,(r < B)$ as the guard.

- The program we came up with:

$$\begin{array}{l}q, r := 0, A \\ \{A = q \times B + r \land 0 \leqslant r, \mathit{bnd} : r\} \\ \mathbf{do}\ B \leqslant r \rightarrow q := q + 1 \\ \qquad\qquad\quad r := r - B \\ \mathbf{od} \\ \{A = q \times B + r \land 0 \leqslant r < B\}\ .\end{array}$$

- In each iteration of the loop, r is decreased by B.

- We can probably get a quicker program by decreasing r by ... $2 \times B$, when possible.

- What about decreasing r by $4 \times B$, $8 \times B$,... etc?

1.1 Division in $O(\log(A/B))$ Time

Strategy...

$$\begin{array}{l}\mathbf{con}\ A, B : \mathit{Int}\ \{0 \leqslant A \land 0 < B\} \\ \mathbf{var}\ q, r, b, k : \mathit{Int} \\ \ldots \\ \{0 \leqslant k \land b = 2^k \times B \land A < b\} \\ \ldots \\ \{A = q \times b + r \land 0 \leqslant r < b \land \\ \quad 0 \leqslant k \land b = 2^k \times B, \mathit{bnd} : b\} \\ \mathbf{do}\ b \neq B \rightarrow \ldots\mathbf{od} \\ \{A = q \times B + r \land 0 \leqslant r < B\}\end{array}$$

Generating $2^k \times B$

- It is easy to satisfy $b = 2^k \times B \land A < b$.

$$\begin{array}{l}b, k := B, 0 \\ \mathbf{do}\ b \leqslant A \rightarrow b, k := b \times 2, k + 1\ \mathbf{od} \\ \{0 \leqslant k \land b = 2^k \times B \land A < b\}\end{array}$$

- What are the loop invariant and the bound?

- Initialisation for the next loop easily follows:

$$\begin{array}{l}\{0 \leqslant k \land b = 2^k \times B \land A < b\} \\ q, r := 0, A \\ \{A = q \times b + r \land 0 \leqslant r < b \land \\ \quad 0 \leqslant k \land b = 2^k \times B\}\end{array}$$

Decreasing b

- What needs to be done before we decrement b by half?

$$\begin{array}{l}\quad (A = q \times b + r \land 0 \leqslant r < b)[b \backslash b\ /\ 2] \\ \equiv (A = q \times (b\ /\ 2) + r \land 0 \leqslant r < b\ /\ 2)\end{array}$$

- We can restore the invariant by $q := q \times 2$...

$$\begin{aligned} & (A = q \times (b\,/\,2) + r \wedge 0 \leqslant r < b\,/\,2)[q\backslash q \times 2] \\ \equiv\ & A = (q \times 2) \times (b\,/\,2) + r \wedge 0 \leqslant r < b\,/\,2 \\ \Leftarrow\ & A = q \times b + r \wedge 0 \leqslant r < b\,/\,2 \wedge b = 2^k \times B \end{aligned}$$

- only if we already have $r < b\,/\,2$!

- That gives us one guarded command:

$$r < b\,/\,2 \to q, b, k := q \times 2, b\,/\,2, k - 1$$

Decreasing b — The Other Case

- What about the case when $b\,/\,2 \leqslant r < b$?

- The task is to find a substitution such that

$$\begin{aligned} & (A = q \times (b\,/\,2) + r \wedge 0 \leqslant r < b\,/\,2)[?\backslash ?] \\ \Leftarrow\ & A = q \times b + r \wedge b\,/\,2 \leqslant r < b \wedge b = 2^k \times B \end{aligned}$$

- Comparing $0 \leqslant r < b/2$ and $b/2 \leqslant r < b$, one might want to try a substitution containing $[r\backslash r - b\,/\,2]$.

$$\begin{aligned} & (0 \leqslant r < b\,/\,2)[r\backslash r - b\,/\,2] \\ \equiv\ & 0 \leqslant r - b\,/\,2 < b\,/\,2 \\ \Leftarrow\ & b\,/\,2 \leqslant r < b \wedge b = 2^k \times B \quad . \end{aligned}$$

- Consider the former half of the expression:

$$\begin{aligned} & (A = q \times (b\,/\,2) + r)[r\backslash r - b\,/\,2] \\ \equiv\ & A = q \times (b\,/\,2) + r - b\,/\,2 \\ \equiv\ & A = (q - 1) \times (b\,/\,2) + r \quad . \end{aligned}$$

- Applying $[q\backslash q \times 2 + 1]$ gives us back $A = q \times b + r$.

- Therefore, another guarded command:

$$\begin{aligned} & b\,/\,2 \leqslant r \to q, b, k, r := \\ & \quad q \times 2 + 1, b\,/\,2, k - 1, r - b\,/\,2 \end{aligned}$$

The Program

$$\begin{array}{l} \mathbf{con}\ A, B : \mathit{Int}\ \{0 \leqslant A \wedge 0 < B\} \\ \mathbf{var}\ q, r, b, k : \mathit{Int} \\ b, k := B, 0 \\ \mathbf{do}\ b \leqslant A \to b, k := b \times 2, k + 1\ \mathbf{od} \\ \{0 \leqslant k \wedge b = 2^k \times B \wedge A < b\} \\ q, r := 0, A \\ \{A = q \times b + r \wedge 0 \leqslant r < b \wedge \\ \quad 0 \leqslant k \wedge b = 2^k \times B, \mathit{bnd} : b\} \\ \mathbf{do}\ b \neq B \to \\ \quad \mathbf{if}\ r < b\,/\,2 \to q, b, k := q \times 2, b\,/\,2, k - 1 \\ \quad \mid\ b\,/\,2 \leqslant r \to q, b, k, r := q \times 2 + 1, b\,/\,2, \\ \qquad\qquad\qquad\qquad\qquad\qquad k - 1, r - b\,/\,2 \\ \quad \mathbf{fi} \\ \mathbf{od} \\ \{A = q \times B + r \wedge 0 \leqslant r < B\} \end{array}$$

1.2 Alternative Programs

Existential Quantification

- The variable k is used in the proofs, but not needed for computing the output.

- Such a variable is called a "ghost variable" in Kaldewaij [Kal90].

- One can remove k, and the program would still work.

- For its reasoning, we need to use existential quantification in the assertions to talk about properties involving k.

$$\begin{array}{l} \mathbf{con}\ A, B : \mathit{Int}\ \{0 \leqslant A \wedge 0 < B\} \\ \mathbf{var}\ q, r, b : \mathit{Int} \\ b := B \\ \mathbf{do}\ b \leqslant A \to b := b \times 2\ \mathbf{od} \\ \{\langle \exists k : 0 \leqslant k : b = 2^k \times B\rangle \wedge A < b\} \\ q, r := 0, A \\ \{A = q \times b + r \wedge 0 \leqslant r < b \wedge \\ \quad \langle \exists k : 0 \leqslant k : b = 2^k \times B\rangle, \mathit{bnd} : b\} \\ \mathbf{do}\ b \neq B \to \\ \quad \mathbf{if}\ r < b\,/\,2 \to q, b := q \times 2, b\,/\,2 \\ \quad \mid\ b\,/\,2 \leqslant r \to q, b, r := q \times 2 + 1, b\,/\,2, \\ \qquad\qquad\qquad\qquad\qquad\qquad r - b\,/\,2 \\ \quad \mathbf{fi} \\ \mathbf{od} \\ \{A = q \times B + r \wedge 0 \leqslant r < B\} \end{array}$$

In developing such programs,

- We can introduce variables such as k, and realise that they are ghost variables and remove them later.

- Or we can have existential quantification in assertions to begin with, if you are sure that the quantified variables won't be needed.

Alternative Program

Kaldewaij [Kal90] presented the following alternative. Do you prefer this program?

$$
\begin{array}{l}
\mathbf{con}\ A, B : \mathit{Int}\ \{0 \leqslant A \wedge 0 < B\} \\
\mathbf{var}\ q, r, b, k : \mathit{Int} \\
b, k := B, 0 \\
\mathbf{do}\ b \leqslant A \to b, k := b \times 2, k+1\ \mathbf{od} \\
q, r := 0, A \\
\mathbf{do}\ b \neq B \to \\
\quad q, b, k := q \times 2, b \mathbin{/} 2, k-1 \\
\quad \mathbf{if}\ r < b \to \mathit{skip} \\
\quad \mid\ b \leqslant r \to q, r := q+1, r-b \\
\quad \mathbf{fi} \\
\mathbf{od} \\
\{A = q \times B + r \wedge 0 \leqslant r < B\}
\end{array}
$$

- The program has the advantage that we do not need to have $b \mathbin{/} 2$ in the guards.
- Note what the first assignment establishes:

$$
\begin{array}{l}
\{A = q \times b + r \wedge 0 \leqslant r < b \wedge \\
\quad 0 \leqslant k \wedge b = 2^k \times B \wedge b \neq B\} \\
q, b, k := q \times 2, b \mathbin{/} 2, k-1 \\
\{A = q \times b + r \wedge 0 \leqslant r < 2 \times b \wedge \\
\quad 0 \leqslant k \wedge b = 2^k \times B\}
\end{array}
$$

A Historical Note

- The correctness of the **if** in the loop was actually a key example in Dahl [DDH72], one of the earliest book on *structured programming*:

$$
\begin{array}{l}
\{0 \leqslant r < b\} \\
b := b \mathbin{/} 2 \\
\mathbf{if}\ r < b \to \mathit{skip} \\
\mid\ b \leqslant r \to r := r - b \\
\mathbf{fi} \\
\{0 \leqslant r < b\}
\end{array}
$$

- In Dahl [DDH72], Dijkstra needed about one page of textual proof.
- These days we can prove its correctness by routine symbolic manipulation. It shows how much symbolic reasoning has advanced since then.

2 Binary Search Revisited

Binary Search

- Given a sorted array of N numbers and a key, either locate the position where the key resides in the array, or report that the key does not present in the array, in $O(\log N)$ time.
- A possible spec:

$$
\begin{array}{l}
\mathbf{con}\ N, K : \mathit{Int}\ \{0 < N\} \\
\mathbf{con}\ F : \mathbf{array}\ [0..N)\ \mathbf{of}\ \mathit{Int}\ \{F\ \mathit{ascending}\} \\
\mathbf{var}\ l, r : \mathit{Int} \\
\mathit{bsearch} \\
\{F[l] = K \vee ...\}\ .
\end{array}
$$

2.1 The van Gasteren-Feijen Approach

- Van Gasteren and Feijen [vGF95] pointed a surprising fact: binary search does not apply only to sorted lists!
- In fact, they believe that comparing binary search to searching for a word in a dictionary is a major educational blunder.
- Their binary search: let Φ be a predicate on two integers with some additional constraints to be given later:

$$
\begin{array}{l}
\mathbf{con}\ M, N : \mathit{Int}\ \{M < N \wedge \Phi\ M\ N \wedge ...\} \\
\mathbf{var}\ l, r : \mathit{Int} \\
\mathit{bsearch} \\
\{M \leqslant l < N \wedge \Phi\ l\ (l+1)\}\ .
\end{array}
$$

Invariant and Bound

- Invariant: $\Phi\ l\ r \wedge M \leqslant l < r \leqslant N$, loop guard: $l+1 \neq r$.
- Initialisation: $l, r := M, N$.
- Bound: $r - l$.
- For any m such that $l < m < r$, we have $r - m < r - l$ and $m - l < r - l$. Therefore both $l := m$ and $r := m$ decrease the bound.

Constructing the Loop Body

- For $l := m$ we calculate.

$$\begin{aligned} & (\Phi\ l\ r \wedge M \leqslant l < r \leqslant N)[l \backslash m] \\ \equiv\ & \Phi\ l\ m \wedge M \leqslant m < r \leqslant N \\ \Leftarrow\ & \Phi\ l\ m \wedge M \leqslant l < m < r \leqslant N \quad . \end{aligned}$$

- That $l{<}m{<}r$ is our assumption. The leftover $\Phi\ l\ m$ gives rise to a guarded command: $\Phi\ l\ m \rightarrow l := m$.

- The case with $r := m$ is similar.

The Program Skeleton

$$\begin{array}{l} \{M < N \wedge \Phi\ M\ N\} \\ l, r := M, N \\ \{\Phi\ l\ r \wedge M \leqslant l < r \leqslant N, \mathit{bnd} : r - l\} \\ \mathbf{do}\ l + 1 \neq r \rightarrow \\ \quad \{\ldots \wedge l + 2 \leqslant r\} \\ \quad m := \text{anything s.t. } l < m < r \\ \quad \{\ldots \wedge l < m < r\} \\ \quad \mathbf{if}\ \Phi\ m\ r \rightarrow l := m \\ \quad \ \mid\ \Phi\ l\ m\ \rightarrow r := m \\ \quad \mathbf{fi} \\ \mathbf{od} \\ \{M \leqslant l < N \wedge \Phi\ l\ (l+1)\} \end{array}$$

Note: $m := (l + r) \,/\, 2$ is a valid choice, thanks to the precondition that $l + 2 \leqslant r$.

Constraints on Φ

- But we need the **if** to be total.

- Therefore we demand a constrant on Φ:

$$\Phi\ l\ r \Rightarrow \Phi\ l\ m \vee \Phi\ m\ r, \text{ if } l < m < r. \qquad (1)$$

- Some Φ satisfying (1) (for F of appropriate type):
  - $\Phi\ l\ r \equiv F[l] \neq F[r]$,
  - $\Phi\ l\ r \equiv F[l] < F[r]$,
  - $\Phi\ l\ r \equiv F[l] \leqslant A \wedge A \leqslant F[r]$,
  - $\Phi\ l\ r \equiv F[l] \times F[r] \leqslant 0$,
  - $\Phi\ l\ r \equiv F[l] \vee F[r]$,
  - $\Phi\ l\ r \equiv \neg\,(Q\ l) \wedge Q\ r$.

- Van Gasteren and Feijen believe that $\Phi\ l\ r = F[l] \neq F[r]$ is a better example when explaining binary search.

2.2 Searching for a Key

- The case $\Phi\ l\ r \equiv \neg\ (Q\ l)\ \wedge\ Q\ r$ is worth special attention.

- Choose $Q\ i \equiv K < F[i]$ for some K.

- Therefore $\Phi\ l\ r \equiv F[l] \leqslant K < F[r]$.

- That constitutes the binary search we wanted!

- The postcondition: $M \leqslant l{<}N \wedge F[l] \leqslant K{<}F[l{+}1]$.

- Note that we do *not* yet need F to be sorted!

- The algorithm gives you some l such that $F[l] \leqslant K < F[l{+}1]$. If there are more than one such l, one is returned non-deterministically.

Sortedness

- That F is sorted comes in when we need to establish that there is at most one l satisfying the postcondition.

- That is, either $F[l] = K$, or $\neg\ \langle \exists i : M \leqslant i < N : F[i] = K \rangle$.

The Program... Or A Part Of It

- Let $\Phi\ l\ r = F[l] \leqslant K < F[r]$.

- Processing the array fragment $F\ [a\,..\,b]$:

$$\begin{array}{l} l, r := a, b \\ \{\Phi\ l\ r \wedge a \leqslant l < r \leqslant b, \mathit{bnd} : r - l\} \\ \mathbf{do}\ l + 1 \neq r \rightarrow \\ \quad m := (l + r) \,/\, 2 \\ \quad \mathbf{if}\ F[m] \leqslant K \rightarrow l := m \\ \quad \ \mid\ K < F[m] \rightarrow r := m \\ \quad \mathbf{fi} \\ \mathbf{od} \\ \{a \leqslant l < b \wedge F[l] \leqslant K < F[l + 1]\} \end{array}$$

- Note that $F[a]$ and $F[b]$ are never accessed.

- This program is not yet complete....

Initialisation

- But wait.. to apply the algorithm to the entire array, we need the precondition $\Phi\ 0\ N$, that is $F[0] \leqslant K < F[N]$. Is that true? (We don't even have $F[N]$.)
- One can rule out cases when the precondition do not hold (and also deal with empty array). E.g.

$$
\begin{array}{l}
\textbf{if } 0 = N \to p := \mathit{False} \\
\mid\ 0 < N \to \\
\quad \textbf{if } K < F[0] \to p := \mathit{False} \\
\quad \mid\ F[N-1] = K \to p, l := \mathit{True}, N-1 \\
\quad \mid\ F[0] \leqslant K < F[N-1] \to \\
\qquad\quad a, b := 0, N-1 \\
\qquad\quad \mathit{program\ above} \\
\qquad\quad p := F[l] = K \\
\quad \textbf{fi} \\
\textbf{fi}
\end{array}
$$

- where p is True iff. K presents in F.

Pseudo Elements

- But there is a better way... introduce two pseudo elements!
- Let $F[-1] = -\infty$ and $F[N] = \infty$.
- Initially, $\Phi\ 0\ N$ is satisfied.
- In the code, $F[-1]$ and $F[N]$ are never accessed. Therefore we do not actually have to represent them!
- We need to be careful interpreting the result, once the main loop terminates, however.

The Program (1)

Let $\Phi\ l\ r = F[l] \leqslant K < F[r]$.

$$
\begin{array}{l}
\textbf{con } N, K : \mathit{Int}\ \{0 \leqslant N\} \\
\textbf{con } F : \textbf{array } [0..N) \textbf{ of } \mathit{Int}\ \{F\ \mathit{ascending} \wedge \\
\quad F[-1] = -\infty \wedge F[N] = \infty\} \\
\textbf{var } l, m, r : \mathit{Int} \\
\textbf{var } p : \mathit{Bool} \\
l, r := -1, N \\
\{\Phi\ l\ r \wedge -1 \leqslant l < r \leqslant N, \mathit{bnd} : r - l\} \\
\textbf{do } l + 1 \neq r \to \\
\quad m := (l + r) / 2 \\
\quad \textbf{if } F[m] \leqslant K \to l := m \\
\quad \mid\ K < F[m] \to r := m \\
\quad \textbf{fi} \\
\textbf{od} \\
\{-1 \leqslant l < N \wedge F[l] \leqslant K < F[l+1]\}
\end{array}
$$

The Program (2)

$$
\begin{array}{l}
\{-1 \leqslant l < N \wedge F[l] \leqslant K < F[l+1]\} \\
\textbf{if } -1 = l \to p := \mathit{False} \\
\mid\ 0 \leqslant l \quad\ \to p := F[l] = K \\
\textbf{fi} \\
\{p = \langle \exists i : 0 \leqslant i < N : F[i] = K \rangle \wedge \\
\quad p \Rightarrow F[l] = K\}
\end{array}
$$

Alternative Program

- Kaldewaij [Kal90, Sec. 6.3] derived an alternative program that introduces only $F[N] = \infty$ (but not $F[-1] = -\infty$), while requiring the array to be non-empty.
- The main loop is the same. It is only post-loop interpretation that is different.

2.3 Searching with Premature Return

A More Common Program

- Recall that Bentley [Ben86, pp. 35-36] proposed using binary search as an exercise.
- Bentley's solution can be rephrased below:

$$
\begin{array}{l}
l, r, p := 0, N-1, \mathit{False} \\
\textbf{do } l \leqslant r \to \\
\quad m := (l + r) / 2 \\
\quad \textbf{if } F[m] < K \to l := m + 1 \\
\quad \mid\ F[m] = k\ \to p := \mathit{True}; \mathit{break} \\
\quad \mid\ K < F[m] \to r := m - 1 \\
\quad \textbf{fi} \\
\textbf{od}
\end{array}
$$

A More Common Program

I'd like to derive it, but

- it is harder to formally deal with break.
  - Still, Bentley employed a semi-formal reasoning using a loop invariant to argue for the correctness of the program.
- To relate the test $F[m] < K$ to $l := m + 1$ we have to bring in the fact that F is sorted earlier.

Comparison

- The two programs do not solve exactly the same problem (e.g. when there are multiple Ks in F).
- Is the second program quicker because it assigns l and r to $m+1$ and $m-1$ rather than m?
  - $l := m+1$ because $F[m]$ is covered in another case;
  - $r := m-1$ because a range is represented differently.
- Is it quicker to perform an extra test to $return$ early?
  - When K is not in F, the test is wasted.
  - Rolfe [Rol97] claimed that single comparison is quicker in average.
  - Knuth [Knu97, Exercise 23, Section 6.2.1]: single comparison needs $17.5 \lg N + 17$ instructions, double comparison needs $18 \lg N - 16$ instructions.

References

[Ben86] J. L. Bentley. *Programming Pearls.* Addison-Wesley, 1986.

[DDH72] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming.* Academic Press, 1972.

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

[Knu97] D. E. Knuth. *The Art of Computer Programming Volume 3: Sorting and Searching, 3rd Edition.* Addison Wesley, 1997.

[Rol97] T. J. Rolfe. Analytic derivation of comparisons in binary search. *SIGNUM Newsletter*, 32(4):15–19, October 1997.

[vGF95] A. J. M. van Gasteren and W. H. J. Feijen. The binary search revisited. AvG127/WF214, November 1995.