Feijen's Table of Cubes Problem

Technical Report UMCIS–1994–05

H. Conrad Cunningham
cunningham@cs.olemiss.edu

Software Architecture Research Group
Department of Computer and Information Science
University of Mississippi
203 Weir Hall
University, Mississippi 38677 USA

March 1994
Revised August 1996
Reprinted January 2006

H. Conrad Cunningham, D.Sc.
Associate Professor
Department of Computer and Information Science
University of Mississippi
203 Weir Hall
University, Mississippi 38677
USA

cunningham@cs.olemiss.edu

Acknowledgements

This is a set of notes for an introductory lecture on program derivation. It is based, in part, on Section 10.4 of Edward Cohen's textbook *Programming in the 1990's: An Introduction to the Calculation of Programs* (Springer-Verlag, 1990). Cohen acknowledged that he based his presentation on a lecture presented by Wim Feijen at Harvard University in May 1988.

The preparation of this document was supported by the National Science Foundation under Grant CCR-9210342 and by the Department of Computer and Information Science at the University of Mississippi.

Problem Description

Write a program to print the cubes of the first N natural numbers for some arbitrary $N \geq 0$. The program cannot use an array or the exponentiation or multiplication operators. The program can only print one integer at a time.

Note: For computing scientists, natural numbers are the integers 0, 1, 2, ...

Program Specification

First, we must formalize the problem by giving a *specification* in terms of a *precondition* and a *postcondition* for a program. We state these as logical assertions about the state of the program, i.e., about the relationships among the values of the program variables, parameters, and constants.

A precondition is an assertion that describes the assumed state of the program before it begins execution—the "input" conditions.

A postcondition is an assertion that describes the desired state of the program when it terminates—the "output" or "result" conditions.

We often write a program specification as a Hoare triple

$$\{\ Q\ \}\ \text{Program}\ \{\ R\ \}$$

where Q is the precondition assertion and R is the postcondition assertion. Informally, the above Hoare triple means that, if the *Program* begins execution in any state in which Q is true, then it must always terminate in a state in which R is true.

For the table of cubes program, the only precondition we have is that N be a natural number (i.e., an integer at least zero). The postcondition is that the program has "printed" the cubes of the first N natural numbers (i.e., printed the cubes for all integers in the range $0 \ldots (N-1)$). We leave the concept of "printed" as an informal notion. Thus the pre- and postconditions are assertions Q and R below.

(Precondition) $\quad Q : N \geq 0$
(Postcondition) $\quad R : (\forall\, i : 0 \leq i < N : i^3 \text{ printed})$

We read assertion R as "for all i such that i is at least 0 and less than N, i^3 has been printed".

Program Derivation

Programming is a goal-directed activity. That is, we focus on postcondition, the goal, more than we do the precondition.

We begin our program development by speculating on the program structure needed to reach the goal under the constraints imposed by the problem. That is, we begin by making an educated guess about the structure of the required program.

Most non-trivial programs require a loop. Is a loop needed in this case?

Yes, since N is an arbitrary natural number value, since we must print a value for each natural number less than N, and since only one value can be printed with each print operation.

In the Guarded Commands language we use, the basic looping construct is the **do**, which has the syntax

do $B \rightarrow$ body **od**

where predicate B is called the guard of the loop. The semantics (i.e., meaning) of the **do** is similar to that of the familiar "while" construct in Pascal or C. If the guard evaluates to true, the body is executed once and the loop is repeated. If the guard evaluates to false, the loop exits.

By examining the postcondition and using the formal semantics of the **do** construct, we structure a possible loop to meet the specification. At this stage, the loop will usually include several abstract components (i.e., components that are not fully implemented). For the table of cubes problem, we have following loop:

$\{ Q \}$
initialization $\{ P$, the loop invariant $\}$;
do $B \rightarrow \{ P \wedge B \}$
progress toward goal while preserving invariant
$\{ P \}$
od $\{ P \wedge \neg B \}$
$\{ R \}$

Above we show that an assertion must hold at some point in the program by inserting the assertion between braces at the appropriate point in the program text.

A loop invariant is an assertion about the program state that must hold *every time* that the loop's guard is evaluated. That is, it must hold just before the loop is entered, after each execution of the the loop body, and just after the loop is exited.

Thus the "initialization" code must make invariant assertion P true initially. The loop must keep P true and eventually make B false, causing the loop to terminate. At the end of the loop, $P \wedge \neg B$ must establish (i.e., imply) postcondition R.

Each abstract component is a programming subproblem that we must solve to construct a solution to the original problem. We must now refine the abstract parts of our program: the assertions P and B, the initialization, and the loop body.

Now let's focus on loop invariant P and guard B. Remembering that $P \wedge \neg B$ must imply R, we will manipulate R syntactically to find candidate P and B. The syntactical manipulation will use various *heuristics* to guide the programming effort.

Applying Heuristic: Replacing a Constant by a Variable

One of the most commonly used heuristics is called *replacing a constant by a variable*. The approach is to generalize the postcondition predicate in a way that will allow a candidate loop invariant and guard to be identified. We choose a constant that appears in the postcondition, replace it by a new variable, and require that this new variable have the constant as its value at termination.

In applying the heuristic to the table of cubes problem, we go through the following steps:

- Identify the constants in postcondition R: $\forall\, i : 0 \leq i < N : i^3$ printed).
  There are three obvious ones: 0, 3, and N.
- Choose to replace constant N by a new variable n.
- Identify the new postcondition $R' : (\forall\, i : 0 \leq i < n : i^3 \text{ printed}) \wedge n = N$.
- Continue by matching $P \wedge \neg B$ against shape of R'.

Applying Heuristic: Deleting a Conjunct

Given the generalized postcondition R', we can now apply the heuristic *delete a conjunct* to find values for the loop invariant P and guard B.

Matching $P \wedge \neg B$ against $(\forall\, i : 0 \leq i < n : i^3 \text{ printed})$, we can take (i.e., "delete") the $n = N$ component of the postcondition and make its negation guard B. The remaining portion of the postcondition becomes loop invariant P_0.

Often when deleting a conjunct like $n = N$ we need to add an additional invariant giving the valid range of values for the variable n. We do so in this program to get invariant P_1.

(Invariants) $P_0 : (\forall\, i : 0 \leq i < n : i^3 \text{ printed})$
$P_1 : 0 \leq n \leq N$

Henceforth, let P represent the conjunction (and-ing) of all the loop invariants identified, e.g., $P_0 \wedge P_1$ at this point in the derivation.

What initialization is needed to make P hold at the beginning of the loop? The statement $n := 0$ will make the range on i in P_0 to be empty and hence make the invariant trivially true.

What loop body is needed? We need to do two things: (1) cause progress to occur, that is, cause n to get closer to its final value N; and (2) restore the invariant for the next iteration of the loop.

Since the program can print only one integer at a time, we choose to make the value of variable n approach N in steps of one. Thus the progress statement in the loop would be $n := n + 1$.

To preserve the invariant given the choice for a progress statement, we include the statement $print.(n^3)$.

Thus we have the following program:

$\{\ Q\ \}$
$n := 0$; $\{$ invariant P: $P_0 \wedge P_1\ \}$
do $n \neq N \rightarrow \{\ P \wedge n \neq N\ \}$
$\quad print.(n^3);$
$\quad n := n + 1$
$\quad \{\ P\ \}$
od $\{\ P \wedge n = N\ \}$
$\{\ R\ \}$

The above solution is not yet satisfactory. It contains the expression n^3 which is not allowed. Thus we must come up with a way to compute and print this value without using exponentiation or multiplication.

Applying Heuristic: Strengthening the Invariant

A reasonable way to handle the problem of the unwanted n^3 seems to be to add a new variable, say x, change the program so that x always has the value n^3 at the print statement, and make sure that that the computation of n^3 does not violate the restrictions. This is an example of the heuristic called *strengthening the invariant.*

Thus we add variable x and a new invariant P_2:

(Invariant) $\quad P_2 : x = n^3$

Now we have the following program, where E is some abstract expression to be determined later. Given that n is initialized to 0, clearly x must also be initialized to 0 to establish the invariant P_2 at the beginning of the loop.

$\{\ Q\ \}$
$n,\ x := 0,\ 0\ ;\ \{$ invariant P: $P_0 \wedge P_1 \wedge P_2\ \}$
do $n \neq N \rightarrow \{\ P \wedge n \neq N\ \}$
$\quad\quad print.x;$
$\quad\quad n, x := n+1, E$
$\quad\quad \{\ P\ \}$
od $\{\ P_0 \wedge P \wedge n = N\ \}$
$\{\ R\ \}$

The statement $n,\ x := n+1,\ E$ denotes the *simultaneous* assignment of the value $n+1$ to variable n and value E to variable x.

Note that before execution of the assignment the invariants still hold (something that is not always the case in the middle of a loop body). In addition, n^3 has already been printed at that point.

The assignment must update x to have the value $(n+1)^3$ (here n denotes the value of the variable n before the assignment). Before the assignment, variable x has the value n^3.

We must now calculate an appropriate value for E to complete the program. The basic approach is to determine what MUST hold before execution to guarantee the desired post-condition and then compare that to what we know does hold before execution (from the precondition).

Calculation of Unknown Expression

In calculating the abstract expression E, we must use the semantics (i.e., meaning) of the assignment statement. We call this the Axiom of Assignment and write it as the Hoare triple

$$\{\ R(w, v := W, V)\ \}\ w, v := W, V\ \{\ R\ \}$$

where w and v are program variables, W and V are expressions on the program state, and $R^{w,v}_{W,V}$ denotes the expression R where variables w and v have been replaced textually by expressions W and V, respectively. To prove that specification $\{\ Q\ \}\ w, v := W, V\ \{\ R\ \}$ holds, we need to show $Q \Rightarrow R(w, v := W, V)$.

In our table of cubes program, we must then find an expression E such that

$$\{\ P \wedge n \neq N\ \}\ n, x := n+1, E\ \{\ P\ \}$$

holds. We can focus on P_2, the only invariant affected by x. That is, we must find E such that $P \wedge n \neq N \Rightarrow P_2(n, x := n+1, E)$.

We can reason equationally as follows. The text between angle-brackets gives a justification for the transformation from one step to the next (and vice versa).

$$
\begin{array}{ll}
 & P_2(n, x := n+1, E) \\
\equiv & \quad \langle \text{ textual substitution } \rangle \\
 & E = (n+1)^3 \\
\equiv & \quad \langle \text{ arithmetic } \rangle \\
 & E = n^3 + 3*n^2 + 3*n + 1 \\
\equiv & \quad \langle \ x = n^3 \ (P_2) \text{ holds beforehand } \rangle \\
 & E = x + 3*n^2 + 3*n + 1
\end{array}
$$

We have thus calculated a value for E that will make our program correct.

$\{\ Q\ \}$
$n,\ x := 0,\ 0\ \{$ invariant P: $P_0 \wedge P_1 \wedge P_2\ \}$;
do $n \neq N \rightarrow \{\ P \wedge n \neq N\ \}$
$\quad\quad$ *print.x*;
$\quad\quad$ $n, x := n+1, x + 3*n^2 + 3*n + 1$
$\quad\quad$ $\{\ P\ \}$
od $\{\ P \wedge n = N\ \}$
$\{\ R\ \}$

The expression $3*n^2 + 3*n + 1$, which involves both exponentiation and multiplication, is still not satisfactory. But note that the exponentiation is of a lower order, so we did make some progress.

Another Strengthening

To resolve this problem, we again strengthen the invariant by adding a new variable y for the troublesome expression.

(Invariant) $\quad P_3 : y = 3*n^2 + 3*n + 1$

Now we have the following program, where F is some unknown expression to be determined later. The initialization for y to establish the invariant is straightforward.

$\{\ Q\ \}$
$n, x, y := 0, 0, 1\ ;\ \{$ invariant P: $P_0 \wedge P_1 \wedge P_2 \wedge P_3\ \}$;
do $n \neq N \rightarrow \{\ P \wedge n \neq N\ \}$
$\quad\quad$ *print.x*;
$\quad\quad$ $n, x, y := n+1, x+y, F$
$\quad\quad$ $\{\ P\ \}$
od $\{\ P \wedge n = N\ \}$
$\{\ R\ \}$

Thus we must now calculate an appropriate value for F. We proceed as before but restrict our attention to invariant P_3 since none of the other invariants are affected by assignments to y.

$$
\begin{array}{ll}
 & wp.(n, x, y := n+1, x+y, F).(P_3) \\
\equiv & \quad \langle \text{ axiom of assignment } \rangle \\
 & F = 3 * (n+1)^2 + 3 * (n+1) + 1 \\
\equiv & \quad \langle \text{ arithmetic } \rangle \\
 & F = 3 * n^2 + 9 * n + 7 \\
\equiv & \quad \langle \ P_3 \ \rangle \\
 & F = y + 6 * n + 6
\end{array}
$$

Now we have gotten rid of the exponentiation, but we still have a multiplication. The troublesome expression can be rewritten as six additions, but let's take the derivation further, looking for a more elegant program.

Yet Another Strengthening

We again strengthen the invariant by adding a new variable z for the troublesome expression.

(Invariant) $\quad P_4 : z = 6 * n + 6$

Now we have the following program, where G is some unknown expression to be determined later. The initialization for z to establish the invariant is straightforward.

$\{ \ Q \ \}$
$n, x, y, z := 0, 0, 1, 6$; { invariant P: $P_0 \wedge P_1 \wedge P_2 \wedge P_3 \wedge P_4$ } ;
do $n \neq N \rightarrow \{ \ P \wedge n \neq N \ \}$
    $print.x$;
    $n, x, y, z := n+1, x+y, y+z, G$
    $\{ \ P \ \}$
od $\{ \ P \wedge n = N \ \}$
$\{ \ R \ \}$

Thus we must now calculate an appropriate value for G. We proceed as before, but restrict our attention to invariant P_4.

$$
\begin{array}{ll}
 & wp.(n, x, y, z := n+1, x+y, y+z, G).(P_4) \\
\equiv & \quad \langle \text{ axiom of assignment } \rangle \\
 & G = 6 * (n+1) + 6 \\
\equiv & \quad \langle \text{ arithmetic } \rangle \\
 & G = 6 * n + 12 \\
\equiv & \quad \langle \ P_4 \ \rangle \\
 & G = z + 6
\end{array}
$$

Thus we have the final program:

$$
\begin{array}{l}
\{ \ Q \ \} \\
n, x, y, z := 0, 0, 1, 6 \ ; \ \{ \text{ invariant } P\text{: } P_0 \wedge P_1 \wedge P_2 \wedge P_3 \wedge P_4 \ \} \ ; \\
\textbf{do } n \neq N \rightarrow \{ \ P \wedge n \neq N \ \} \\
\qquad \mathit{print}.x; \\
\qquad n, x, y, z := n+1, x+y, y+z, z+6 \\
\qquad \{ \ P \ \} \\
\textbf{od } \{ \ P \wedge n = N \ \} \\
\{ \ R \ \}
\end{array}
$$

We are done! We found an efficient linear-time solution that was not easy to see initially. Moreover, we developed it in such a way that we have confidence that the program satisfies its specification.

Review of Derivation

Now let's review what we have done.

- We first stated precise specification consisting of precondition and postcondition.
- We determined that a loop was necessary.
- We manipulated the postcondition using the heuristics "replacing a constant by a variable" and "deleting a conjunct" to arrive at the basic structure for the loop—a guard ($n \neq N$), a progress statement ($n := n + 1$), and an initialization ($n := 0$).
- We attempted to calculate the values of the abstract expressions of the loop by using the formal semantics of the language constructs (i.e., the axiom of assignment) and logical and arithmetic manipulations.
- When we derived an expression that needed to be simplified further (i.e., that didn't satisfy the restrictions or could not be expressed in terms of the values of the variables), we applied the heuristic "strengthening the invariant" to add new variables.

- We continued until we derived a program satisfying the requirements of the problem.

In this case study, we derived a program that would have been difficult to find otherwise. Logic, semantics, and heuristics provided systematic problem-solving method.

A formal proof of the program would essentially be the derivation with the steps reversed.

Excerpts from Dijkstra

Note: From Edsger W. Dijkstra. "On the Cruelty of Really Teaching Computer Science", *CACM*, Vol. 32, No. 12, December 1989.

Well, when all is said and done, the only thing computers can do for us is to manipulate symbols and produce the result of such manipulations. ...

But before a computer is ready to perform a class of meaningful manipulations—or calculations, if you prefer—we must write a program.

What is a program? Several answers are possible.

We can view the program as what turns the general-purpose computer into a special-purpose symbol manipulator, and it does so without the need to change a single wire. ...

I prefer to describe it the other way around. The program is an abstract symbol manipulator which can be turned into a concrete one by supplying a computer to it.

After all, it is no longer the purpose of programs to instruct our machines; these days, it is the purpose of machines to execute our programs.

So, we have to design abstract symbol manipulators. We all know what they look like. They look like programs or—to use somewhat more general terminology—usually rather elaborate formulae from some formal system. ...

[The] programmer ... has to derive that formula; he has to derive that program.

We know of only one reliable way of doing that, *viz.*, symbol manipulation.

And now the circle is closed. We construct our mechanical symbol manipulators by means of human symbol manipulation.

Hence, computing science is—and will always be—concerned with the interplay between mechanized and human symbol manipulation usually referred to as "computing" and "programming," respectively.

Teaching to unsuspecting youngsters the effective use of formal methods is one of the joys of life because it so extremely rewarding. Within a few months, they find their way into a new world with a justified degree of confidence that is radically novel for them; within a few months, their concept of intellectual culture has acquired a radically new dimension. To my taste and style, that is what education is about.