Proofs of Correctness

- History and Motivation
- Basic Concepts
- The "Structured Theorem"
- Proofs Involving Sequence
- Proofs Involving Selection
- Proofs Involving Iteration
- Example Proof
- Strategies
- Potential Traps and Pitfalls
- Advanced Concepts
- Advantages and Disadvantages
- So What???
- References for More Information

History and Motivation

- History
  - Corrado Boehm and Giuseppe Jacopini, 1966
    - *"Control logic of any flowchartable program can be expressed without gotos, using only sequence, selection, and iteration"*
  - Edsger Dijkstra, 1970
    - "No Goto" proposal was motivated by his desire to shorten proofs of correctness
  - Harlan Mills, 1987
    - Cleanroom Software Engineering

- Motivation: change the nature of programming
  - From: a private, puzzle solving activity
  - To: a public, mathematics based activity of translating specifications into programs
  - Proven programs can be expected to both run and do the right thing with little or no debugging

- Foundation
  - Formal Logic
  - Predicate Calculus

Basic Concepts

- Definition of "Proof"
  - *"Establishing the validity of a statement esp. by derivation from other statements in accordance with the principles of reasoning"* [Websters]

- Any program (or module) that terminates can be regarded as a rule for a mathematical function that converts an initial state into a final state (whether the original problem is "mathematical" or not)
  - Matrix inversion program
  - Payroll program

- Non-terminating modules (e.g., an operating system) can be expressed as terminating modules in a non-terminating loop

- The function defined by any such terminating module is a set of ordered pairs: the set of initial states, I, and final states, F, that arise in execution
  - (I1,F1), (I2,F2), …, (In,Fn)

- A module is correct with respect to a specification if and only if for every initial state permissible by the specification, the module produces the final state that corresponds to that initial state in the specification

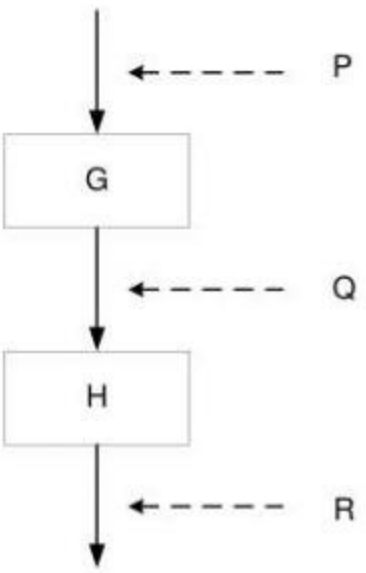
Structured Theorem

- Derived from Boehm and Jacopini
  - There exists a structured program for any problem that permits a flowchartable solution
- Structured programs define a natural hierarchy among parts, which are repeatedly nested into larger and larger parts, by sequence, selection, and iteration
- Each part defines a sub-hierarchy, which can be executed independently of its surroundings in the hierarchy
- Any such part can be called a stub and given a name - but, more importantly, it can be described in a specification that has no control properties, only the effect of that stub on the program's data
  - {pre-condition} function {postcondition}
    - *"if pre-condition is true before the function is executed, then postcondition will be true after"*

Proofs Involving Sequence

- Formal Definition (Proof Rule for Sequences)
  - If: ({P} G {Q}, {Q} H {R})
  - Then: ({P} GH {R})
  - *"If executing G when P gives Q and executing H when Q gives R, then executing G followed by H when P gives R"*

- Graphically:



Sequence (cont)

- Example:

{ P: a,b integer } c := min (a,b) { Q: a,b,c integer, c=lesser of a,b }
{ Q: a,b,c integer, c=lesser of a,b } c := c-1 { R: a,b,c integer, c=(lesser of a,b)-1 }

Becomes

{ P: a,b integer } c := min (a,b); c := c-1 { R: a,b,c integer, c=(lesser of a,b)-1 }

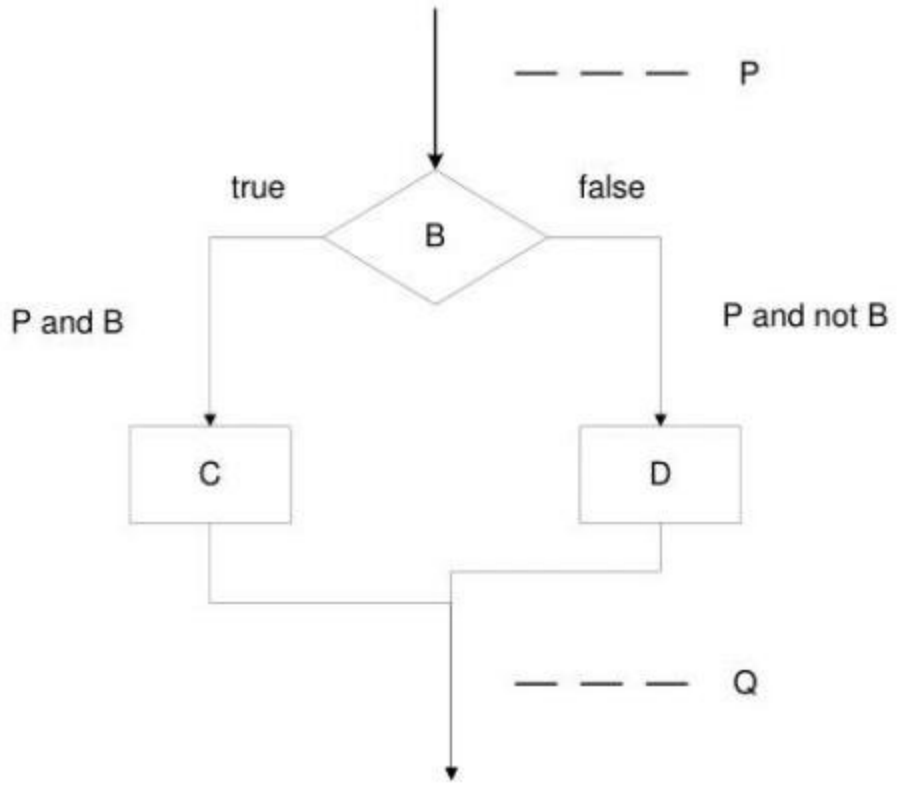
- Or, using a more PDL-ish style:

{ P: a,b integer }
c := min (a,b)
{ Q: a,b,c integer, c=lesser of a,b }
c := c-1
{ R: a,b,c integer, c=(lesser of a,b)-1 }

Proofs Involving Selection

- Formal Definition (Proof Rule for Selection)
  - If: ({P and B} C {Q}, {P and not B} D {Q})
  - Then: ({P} if B then C else D {Q})

- Graphically:



Selection (cont)

- Example

{ P: z integer }

if z<0

then { P and B: z integer, z<0 }

k := -z

{ z,k integer, z<0, k=-z }

else { P and not B: z integer, z>=0 }

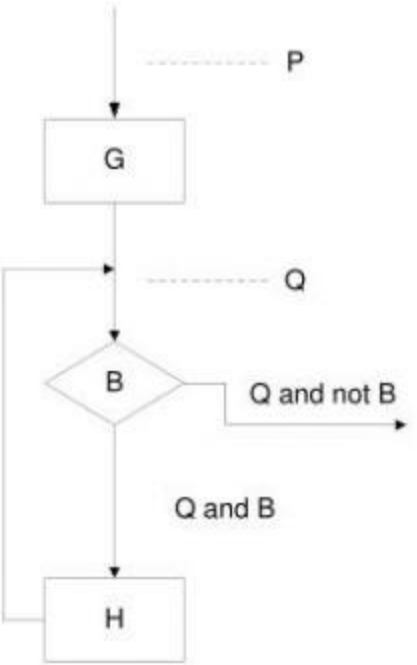
k := z

{ z,k integer, z<0, k=z }

{ Q: z,k integer, if z<0 then k=-z else k=z }

Proofs Involving Iteration

- Formal Definition (Proof Rule for Iteration)
  - If ({P} G {Q}, {Q and B} H {Q})
  - Then ({P} G; while B do H {Q and not B})
    - This rule uses the form of proof-by-induction
    - G is the loop initialization, H is the loop body
    - Q is the "loop invariant", B is the loop condition
- Graphically:



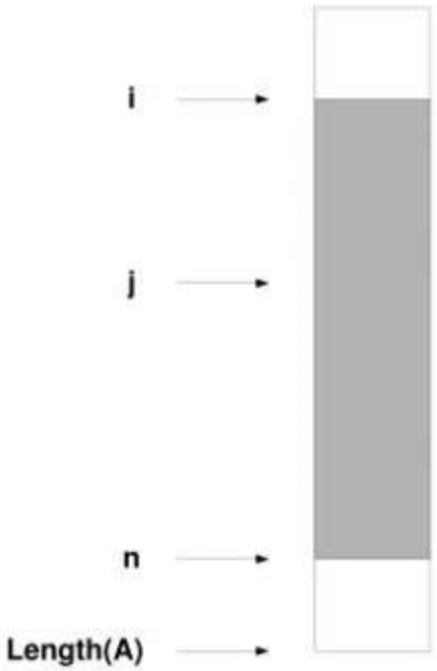
Iteration (cont)

- Loop Invariant
  - Probably the most difficult concept to understand in proofs of correctness
  - Think of it as the condition(s) that must be true at the beginning and end of each cycle through the loop
  - Hint: start by looking at the loop's desired post-condition, the loop invariant will look something like that (some of the loop's work will have been completed and some of the work will remain to be done), e.g.,
    - Imagine the post-condition of a loop to is to:
      - "{ R: have calculated Foo for n items }"
    - The loop invariant would be something like:
      - "{ Q: have calculated Foo for k items, where 0<=k<=n }"
        - » The items from 1..k have been calculated
        - » The items from k+1..n have not
    - (Note: the loop condition, B, is obviously k=n)

- The loop body, H, has two distinct jobs
  - Modify the loop variable(s) that drive the loop condition, B, causing the loop to eventually terminate
    - But, modifying the loop variable(s) will temporarily invalidate the loop invariant, Q
  - The remainder of H serves to carry out the work necessary to re-establish Q based on the updated loop variable(s)

Example Proof

- Find smallest item A[j] in segment A[i..n] of array A
- Precondition, P
  { P: A is integer array, i,n integer, 1<=i<=n<=length(A) }
- Postcondition, R
  { R: P and j integer,
  1<=i<=j<=n<=length(A), A[j] = min(A[i..n]) }



Example Proof (cont)

"executable" PDL is bold

{ proof statements are comments in plain text }

{ *helpful hints are comments in italics* }

```
{ P: A is integer array, i,n integer, 1<=i<=n<=length(A) }
j := i          { initialization, G, to make Q, the loop invariant true }
k := i
{ Q is the loop invariant }
{ Q: P and j,k integer, i<=j<=k<=n, A[j] = min(A[i..k]) }
while k<n       { the test, B }
 do             { L1: Q and B }
                k := k + 1    { need to eventually make B false, but invalidates Q }
                { L2: P and  j,k integer, i<=j<k<=n, A[j] = min(A[i..k-1]) }
                { need to reestablish Q, therefore }
                if A[j] > A[k]
                 then          { T1: L2 and A[k] < A[j] }
                               j := k
                               { T2: Q }
                 { "else" F: P, j,k integer, i<=j<=k<=n, A[j] = min(A[i..k]) }
                { L3: Q (follows from T2 or F) }
 { exit from loop }
{ Q and not B, which means P and j,k integer, i<=j<=k<=n,
                                              A[j] = min(A[i..k]),
                                              and k = n
    therefore
    R: P and j integer, 1<=i<=j<=n<= length(A), A[j] = min(A[i..n]) }
```

Strategies

- Top-down Corollary
  - The lines of a structured program can be written chronologically so that every line can be verified only by reference to lines already written, and not to lines not yet written

- "Top-down Structured Programming"
  - Start with a statement of some problem
    - an informal statement
    - a formal specification
  - Repeat until there are no more non-executable statements:
    - Substitute a non-executable statement by:
      - a sequence
      - a selection
      - an iteration
      - an executable statement
    - Verify the substitution

Potential Traps and Pitfalls

- There is a mistake in the substitution and/or its verification, i.e., the rules of logic do not support such a derivation

- Not determining the proper loop invariant, Q–an invariant is an assertion that never changes, i.e., Q is always true
  - Immediately before loop executes the first time
  - At the end of each iteration of the loop
  - Upon termination of the loop

- Not verifying that the loop terminates properly–does not form an infinite loop

Advanced Concepts

- We have talked about "Axiomatic Verifications"
  - Easier to learn
  - But does not scale well to industrial-sized problems
- Functional Verification
  - Reportedly scales well to industrial-sized problems
  - Reportedly harder to learn
- Mills "Cleanroom Software Engineering"
  - Defect prevention instead of defect removal
    - Mills says debugging tends to violate original architectural assumptions, compromising the design and leading to further defects
  - Highly disciplined approach based on Top-Down Structured Programming
    - Formal specifications (Pre-, Post-condition)
    - Formal proofs of correctness on algorithms
    - Peer reviews of proofs before coding
    - Code generated directly from proofs
    - Statistical based testing
  - Features reduced dependence on debugging
  - Reported to have delivered significantly lower defect densities

Advantages and Disadvantages

- Advantages
  - Based on formal mathematics
  - Can significantly reduce the need for debugging
  - Defects that "survive" into test are easier to find
  - Defects that "survive" into test are easier to repair
  - Resulting systems tend to be far more robust
  - Allows significantly more confidence in the system
  - …
- Disadvantages
  - Difficult to do in practice
  - It's just plain old-fashioned hard work
  - It's certainly not glamorous
  - Proofs can suffer from same logic bugs as code
  - Not "popular"
  - Usually mis-understood
  - …

So What???

- Limitations aside, a well written module is like a well written proof
  - Every step is necessary (no fluff)
  - All steps are sufficient (no gaps)
  - Each step constitutes progress toward the goal
  - Clear, logical progression from step to step
- We could (should) apply the ideas of Top-Down Structured Programming in developing module specifications for our Structured Designs
  - See SE-512, Chapter 17 Structured Design, lecture notes #9, page 5
  - Applies to specifying services in an object-oriented system, as well
- Optimization should start with a system that is known to be logically correct
  - No sense wasting your time on optimizing garbage

References for More Information

- Harlan Mills, "Structured Programming: Retrospect and Prospect", *IEEE Software*, November, 1986
- Edsger Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1986
- David Gries, *The Science of Programming*, Springer-Verlag, 1981
- Harlan Mills, anything on "Cleanroom Software Engineering"