Finding Invariants

Part 1: Replacing Constants by Variables

CS 536: Science of Programming, Spring 2021

3/27: p.5

A. Why

• It is easier to write good programs and check them for defects than to write bad programs and then debug them.
• The hardest part of programming is finding good loop invariants.
• There are heuristics for finding them but no algorithms that work in all cases.
• Changing how we re-establish a loop invariant can greatly speed up the code.

B. Objectives

At the end of this class you should

• Know how to generate possible invariants using “replace a constant by a variable” or more generally “add or modify a parameter” and to be familiar with some examples of these techniques.

C. Finding Invariants

• The key (and often, hardest) part of writing correct programs involves finding invariants for our loops.
  • We need to find an invariant and loop test that establishes the desired postcondition:
    \( \{\text{inv } p\} \text{ while } B \text{ do } ??? \text{ od } \{p \land \neg B\} \{r\} \)
  • The invariant should be easy to establish with some easy initialization code: \(\{p_0\} S_0 \{p\}\).
  • The loop body maintains the invariant: \(\{p \land B\} \text{ loop body } \{p\}\).
  • When the loop terminates, the postcondition we want holds: \(p \land \neg B \rightarrow r\). (Sometimes you have finalization code that you need, then you need \(\{p \land \neg B\} \text{ code } \{r\}\).
• There exist various general heuristics for finding invariants.
  • (Not every way applies to every situation.)
  • General idea: Take the postcondition and weaken it somehow. The loop test is determined by how and how much you weaken the postcondition.
    • One way to weaken the postcondition: Add more states to it. Possibilities include
      • Adding a new parameter, as in Replace a Constant by a Variable, or Split One Variable Into Two.
      • Making a relation more general. E.g., change \(a = b\) to \(a \leq b\) or to an equivalence relation.
• Add a disjunction (generalize the postcondition \( r \) using some \( r' \) to get \( r \lor r' \) as a possible invariant).
• Another way to weaken the postcondition: Stop removing states from it.
• Drop a conjunct (if the postcondition is \( p \land q \), try using just \( p \) or just \( q \) for the invariant).

**D. Replace A Constant By A Variable**

• The technique “Replace a constant by a variable” produces a candidate invariant by adding a new parameter to a predicate.
• We take the postcondition and replace a literal or symbolic constant \( c \) with a fresh variable \( x \):

  Given postcondition \( r \), look for a predicate \( r' \), a new variable \( x \), and a constant \( c \) such that \( r'[c/x] \leftrightarrow r' \). Our possible loop is \( \{ \text{inv } r' \} \text{ while } x \neq c \text{ do } \ldots \text{ od } \{ r' \land x = c \} \{ r \} \). Note: It can be useful to add the range of \( x \) as part of \( r' \).
• Depending on how and what we replace, we get different candidates for invariants, which typically requires different loop tests, initialization code, and loop bodies.

**Example 1:** The summation loops

• The postcondition \( s = \text{sum}(0, n) \) has two constants 0 and \( n \).
• Try replacing \( n \) by a variable \( i \) in the range 0, ..., \( n \). Initialize \( i = 0 \) and increase it until \( i = n \).
  \[
  \{ \text{inv } s = \text{sum}(0, i) \land 0 \leq i \leq n \} \{ \text{bd } n-i \}
  \]
  \[
  \text{while } i \neq n \text{ do } \text{ make } i \text{ larger } \ldots \text{ od}
  \]
  \[
  \{ s = \text{sum}(0, i) \land 0 \leq i \leq n \land i = n \}
  \]
  \[
  \{ s = \text{sum}(0, n) \}
  \]
• Or, replace 0 by a variable \( j \) in the range 0, ..., \( n \). Initialize \( j = n \) and decrease it until \( j = 0 \).
  \[
  \{ \text{inv } s = \text{sum}(j, n) \land 0 \leq j \leq n \} \{ \text{bd } j \}
  \]
  \[
  \text{while } j \neq 0 \text{ do } \text{ make } j \text{ smaller } \ldots \text{ od}
  \]
  \[
  \{ s = \text{sum}(j, n) \land 0 \leq j \leq n \land j = 0 \}
  \]
  \[
  \{ s = \text{sum}(0, n) \}
  \]

**Example 2:** Integer square root

• To take the integer square root of an \( n \geq 0 \) means to find an \( x \) such that \( x \leq \sqrt{n} < x+1 \).
• Let’s rewrite the postcondition as \( x^2 \leq n < (x+1)^2 \). We can weaken it by replacing the 1 by a new variable, say \( y \) and get \( x^2 \leq n < (x+y)^2 \) as a possible invariant. Loop initialization (not shown) sets \( y \) to something large; the loop body makes \( y \) smaller or \( x \) larger
  \[
  \{ \text{inv } x^2 \leq n < (x+y)^2 \land 1 \leq y \} \{ \text{bd } n+y-x \}
  \]
  \[
  \text{while } y \neq 1 \text{ do } \text{ make } x \text{ larger or } x+y \text{ smaller } \ldots \text{ od}
  \]
  \[
  \{ x^2 \leq n < (x+y)^2 \land 1 \leq y \land y = 1 \}
  \]
  \[
  \{ x^2 \leq n < (x+1)^2 \}
  \]
• An extended version of the “Replace a constant by a variable” principle is “Replace an expression by a variable”. E.g., we might change the variable $x$ in $x+1$ to $y$ and get $y+1$ or we could replace the expression $x+1$ by $y$, so $x^2 \leq n < (x+1)^2$ becomes $x^2 \leq n < y^2$. The loop body either increases $x$ or decreases $y$.

\[
\{ \text{inv } 0 \leq x^2 \leq n < y^2 \} \{ \text{bd } y-x \}
\]

\[
\text{while } y \neq x+1 \text{ do } \ldots \text{ make } x \text{ larger or make } y \text{ smaller } \ldots \text{ od}
\]

\[
\{ 0 \leq x^2 \leq n < y^2 \wedge y = x+1 \}
\]

\[
\{ 0 \leq x^2 \leq n < (x+1)^2 \}
\]

• For termination, $0 \leq x^2 < y^2$ implies $y \geq x+1$, so $y-x \geq 0$, and reducing $y$ or increasing $x$ reduces $y-x$.

**Loop Initialization and Variable Ranges When Replacing a Constant by a Variable**

• For loop initialization, we typically establish the invariant by setting variables to some boundary values.
  
  • E.g., if $c_0 \leq v \leq c_1$, try $v := c_0$ or $v := c_1$ as initializations.

  • When we add a variable, it’s clearly a good thing if we have an idea of what range of values the new variable can have. We always have \textit{variable} = \textit{constant} where the constant is the one we're replacing. Initialization should provide a second value the variable can have. Hopefully, this gives us an idea of the range of the new variable. In the other direction, when we ask “Can we replace this constant by a new variable?”, if we can’t find a reasonable initialization value and some idea of what the variable range should be, then maybe the answer to “Can we replace...?" is “No.”

• **Example 3**: Summation loops
  
  • For the invariant $s = \text{sum}(0, i) \wedge 0 \leq i \leq n$, setting $i := 0$ or $i := n$ seems natural:
  
  • $wp(i := 0, p) = s = \text{sum}(0, 0) \wedge 0 \leq 0 \leq n$ is easy to establish with $s := 0$ (and the assumption $n \geq 0$).
  
  • But $wp(i := n, p) = s = \text{sum}(0, n) \wedge 0 \leq n \leq n$ is hard to satisfy (in fact, it’s our original postcondition).

• **Example 4**: For $x^2 \leq n < y^2 \wedge x < y$, try $x := 0$ or $x := 1$ (these imply we need $0^2 \leq n$ or $1^2 \leq n$ respectively). For $y$, we can try $y := n$ (if we know $n > 1$, so that $n < n^2$) or $y := n+1$ (if we know only $n \geq 1$) or $y := n+2$ (if we know only $n \geq 0$).
Ensuring Loop Termination When Replacing a Constant by a Variable

- A loop always has to include at least one progress statement; a statement that gets us closer to termination. If a progress statement \( S_2 \) is put at the end of the loop body, then the rest of the loop body \( S_1 \) has to satisfy
  \[
  \{ p \land B \land t = t_0 \} S_1 \{ \text{wp}(S_2, \ p \land t < t_0) \}
  \]
- So as our loop, we have \( \{ \text{inv} \ p \} \{ \text{bd} \ t \} \) while \( B \{ p \land B \land t = t_0 \} S_1 ; S_2 \{ p \land t < t_0 \} \) od.
- When replacing a constant by a variable, the progress statement takes the variable closer to the target constant.

Two Simple Assignments for Establishing the Value of a Variable

- Say we want \( S \) such that \( \{ v = e_1 \} S \{ v = e_2 \} \). Two simple ways are:
  - \( \{ v = e_1 \} v := v + e_2 - e_1 \{ v = e_2 \} \)
  - \( \{ v = e_1 \} v := v \cdot e_2 \div e_1 \{ v = e_2 \} \)  // (assuming \( e_1 \) divides \( e_2 \))
- One example was in the summation loop: We needed \( s = \text{sum}(0, i+1) \) but had \( s = \text{sum}(0, i) \). We use
  \[
  \{ s = \text{sum}(0, i) \} s := s + (i+1) \{ s = \text{sum}(0, i+1) \}
  \]
  because it is equivalent to (the harder-to-calculate)
  \[
  \{ s = \text{sum}(0, i) \} s := s + \text{sum}(0, i+1) - \text{sum}(0, i) \{ s = \text{sum}(0, i+1) \}
  \]

  - Example 5: (Integer log base 2) Find the largest power of 2 that is \( \leq x \).
    - Say our invariant is \( y = 2k \leq x \land 0 \leq k \) (we loop while \( 2^y \leq x \)) and our progress step is \( k := k+1 \), so the wp of the progress step is \( y = 2k+1 \leq x \land 0 \leq k+1 \).
    - So we need code to establish \( \{ y = 2k \land \ldots \}; y := ??? \{ y = 2k+1 \land \ldots \} k := k+1 \{ y = 2k \land \ldots \} \)
      - One possibility is \( y := y + 2k+1 - 2k \). I.e., \( y := y + 2k \), or just \( y := y + y \), since \( y = 2k \).
      - Another possibility for our statement is \( y := y \cdot 2k+1 \div 2k \), which simplifies to \( y := y \cdot 2 \).

Replacing a Constant by a Variable Can Fail

- Not every constant when replaced yields an invariant that works well.
- E.g. take the postcondition \( x^2 \leq n < (x+1)^2 \) and replace one (or say both) of the 2’s with a new variable \( y \). We loop while \( y \neq 2 \) with a proposed invariant of
  - \( x^2 \leq n < (x+1)^2 \) plus something for the range of \( y \).
  - or \( x^y \leq n < (x+1)^2 \) plus something for the range of \( y \).
• How would we initialize y? If we're using \( x^y \leq n \) we could try \( y := 0 \) so we'd need \( 1 = x^0 \leq n \). Less obvious what to use if we're trying \( n < (x+1)^y \). Maybe \( x := n; y := 1 \)? But we'd need \( n^2 \leq n < (n+1)^1 \), and \( n^2 \leq n \) requires \( n = 0 \) or \( 1 \), which seems kind of limiting (joke) [3/27].

• Progress step: If \( y := y+1 \) (for example) is the progress step, then the rest of the loop body needs to be the missing code in

\[
\{x^y \leq n < (x+1)^2 \land y \neq 2\} \ldots \{x^{y+1} \leq n < (x+1)^2\} \ y := y+1 \ \{x^y \leq n < (x+1)^2\}
\]

• What could the missing code possibly be? Time to give up and look for a different invariant.

Adding or Modifying Parameters [3/27]

• Replacing a constant by a variable is a simple version of the more general notion of adding or modifying the parameters of a predicate. Other versions of this technique include other changes:

  • **Replace:** One occurrence | Multiple occurrences
  • **Of:** A Constant | A Constant-Valued Expression | A Variable | An Expression
  • **With:** A New Variable | An Expression with one or more New Variables
Finding Invariants

Part 1: Replacing Constants by Variables

A. Why

• It is easier to write good programs and check them for defects than to write bad programs and then debug them.
• The hardest part of programming is finding good loop invariants.
• There are heuristics for finding them but no algorithms that work in all cases.

B. Objectives

At the end of this activity assignment you should

• Be able to how to generate possible invariants using “replace a constant by a variable” or more generally “add a parameter”.

C. Problems

1. What are the constants in the postcondition \( x = \max(b[0], b[1], \ldots, b[n-1]) \)? Using the technique “replace a constant by a variable,” list the possible invariants for this postcondition. Also, what would the loop tests be? (Assume \( n-1 \) is a constant.)

2. Repeat, on the postcondition \( x = n! \), where \( n! \) is short for a function call \( \text{product}(1, n) \).

3. Repeat, on the postcondition \( \forall i . \ 0 \leq i < n \rightarrow b[i] = 3 \).

4. Repeat, on the postcondition \( \forall i . \ \forall j . \ 0 \leq i < K \land K \leq j < n \rightarrow b[i] < b[j] \). (Every value in \( b[0...K-1] \) is < every value in \( b[K...n-1] \).)
Solution to Practice 19 (Finding Invariants; Examples)

1. Certainly 0 is a constant; if we replace it by a variable \( i \), we get
   \[
   \{ \text{inv } x = \max(b[i], \ldots, b[n-1]) \land 0 \leq i \leq n-1 \} \text{ while } i \neq 0 \text{ do } \ldots
   \]
   As a constant, \( n-1 \) seems better than just \( n \) or 1 by themselves:
   \[
   \{ \text{inv } x = \max(b[0], \ldots, b[j]) \land \quad 0 \leq j \leq n-1 \} \text{ while } j \neq n-1 \text{ do } \ldots
   \]
   If you want to treat just \( n-1 \) as a constant and replace it by a variable \( j \), we get
   \[
   \{ \text{inv } x = \max(b[0], \ldots, b[j-1]) \land \quad 1 \leq j \leq n \} \text{ while } j \neq n \text{ do } \ldots
   \]
   Similarly, if you want replace just the 1 in \( n-1 \) by with \( j \), we get
   \[
   \{ \text{inv } x = \max(b[0], \ldots, b[j]) \land\quad 1 \leq j \leq n \} \text{ while } j \neq n \text{ do } \ldots
   \]

2. We can replace \( n \) by a variable and get
   \[
   \text{inv } x = i! \land \quad 1 \leq i \leq n \}
   \text{ while } i \neq n \text{ do } \ldots
   \]
   We can replace 1 and get
   \[
   \{ \text{inv } x = j*(j+1)*\ldots*n \land \quad 1 \leq j \leq n \} \text{ while } j \neq 1 \text{ do } \ldots
   \]

3. For \( \forall i . \quad 0 \leq i < n \rightarrow b[i] = 3 \) as the postcondition, we can replace 0 or \( n \) or 3.
   Replace 0 by \( k \):
   \[
   \{ \text{inv } 0 \leq k \leq n-1 \land \forall i . \quad k \leq i < n \rightarrow b[i] = 3 \} \text{ while } k \neq 0 \text{ do } \ldots
   \]
   Replace \( n \) by \( k \)
   \[
   \{ \text{inv } 0 \leq k \leq n \land \forall i . \quad 0 \leq i < k \rightarrow b[i] = 3 \} \text{ while } k \neq n \text{ do } \ldots
   \]
   Replace 3 by \( k \) (this doesn't look useful)
   \[
   \{ \text{inv } \forall i . \quad 0 \leq i < n \rightarrow b[i] = k \} \text{ while } k \neq 3 \text{ do } \ldots
   \]

4. For \( \forall i . \forall j . \quad 0 \leq i < K \land K \leq j < n \rightarrow b[i] < b[j] \), we have constants 0, \( n \), the two occurrences of \( K \).
   Replace 0 by \( k \):
   \[
   \{ \text{inv } 0 \leq k \leq K \land \forall i . \quad \forall j . \quad k \leq i < K \land K \leq j < n \rightarrow b[i] < b[j]\}
   \text{ while } k \neq 0
   \]
   Replace left \( K \) by \( k \):
   \[
   \{ \text{inv } 0 \leq k \leq \forall i . \quad \forall j . \quad 0 \leq i < k \land K \leq j < n \rightarrow b[i] < b[j]\}
   \text{ while } k \neq K
   \]
   Replace right \( K \) by \( k \):
   \[
   \{ \text{inv } K \leq k \leq n \land \forall i . \quad \forall j . \quad 0 \leq i < K \land k \leq j < n \rightarrow b[i] < b[j]\}
   \text{ while } k \neq K
   \]
   Replace \( n \) by \( k \):
   \[
   \{ \text{inv } K \leq k \leq n \land \forall i . \quad \forall j . \quad 0 \leq i < K \land k \leq j < n \rightarrow b[i] < b[j]\}
   \text{ while } k \neq n
   \]
   You could argue that the ranges for \( k \) could be \( 0 \leq k < n \), \( 0 \leq k < n \), \( 0 \leq k \leq n \), and \( 0 \leq k \leq n \) for the four cases above; it depends on knowing more about the context of the problem.