Finding Invariants

Part 1: Adding Parameters by Replacing Constants by variables

CS 536: Science of Programming, Spring 2023

2023-03-27 pp. 3, 5, 6; 2023-04-29 pp. 1, 4

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 an 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: $\{\textbf{inv}\ p\}$ **while** B **do** $???$ **od** $\{p \wedge \neg B\}\ \{q\}$
  - The invariant should be easy to establish with some easy initialization code: $\{p_0\}S_0\{p\}$.
  - The loop body maintains the invariant: $\{p \wedge B\}$ *loop body* $\{p\}$.
  - When the loop terminates, the postcondition we want holds: $p \wedge \neg B \rightarrow q$. (Sometimes you have finalization code that you need, then you need $\{p \wedge \neg B\}$ *code* $\{q\}$.
- Note: $p \wedge \neg B$ is stronger than q but p itself is weaker than q ([2023-04-29] i.e., $q \rightarrow p$). This suggests the main way people find invariants: Take the postcondition q and weaken it somehow. In particular, weaken it in a way that combining with $\neg B$ gives us q.
- However, it's not required that $p \wedge B \rightarrow \neg q$. If $\neg B$ is too strong, we could be missing out on some states in q, which would make our loop too restrictive in the states it terminates in. So finding the right B is a balancing act.

General Heuristics for Finding Invariants

- There exist various general heuristics for finding invariants. Not every way applies to every situation, but all have the pattern of weakening the postcondition, with the loop test shaped by how and how much we weaken the postcondition.

Add More States to q

- Adding states to q will weaken it. We want B to include the states that are added, and $\neg B$ avoids those states so that $p \wedge \neg B$ only includes states from q. We can try to:
  - ***Add parameters***, as in Replace a Constant by a Variable, or more generally, Replace an Expression With Another Expression..
  - ***Generalize Relations*** (e.g., change an = to a ≤ or to an equivalence relation).
  - ***Add a Disjunction*** (for some r, use $q \vee r$ an invariant).

Remove Fewer States from q

- Say q is constructed as a conjunction of various properties $q_1 \wedge q_2 \wedge q_3$ etc. We can see this as starting with q_1, then adding q_2 to remove some states, then adding q_3 to remove more states, and so on. If we drop a conjunct from this removal process, then the result is weaker than q .

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.
  - We can't phrase this as saying $p \equiv q[x/c]$ because substitution replaces a variable with an expression and also, in the simplest case, we only replace one occurrence of c with x, not all occurrences. So the correct phrasing is $q \equiv p[c/x]$, with p being the candidate invariant. Our candidate loop will be *{* ***inv*** *p }* ***while*** $x \neq c$ ***do*** … ***od*** $\{p \wedge x = c\}$ $\{q\}$.
- Depending on how many constants appear where in q , there can be multiple candidate invariants, which will typically have different loop tests, initialization code, and/or progress steps.
- In addition, not all the candidate invariants may be unusable, generally because there's no good initialization code or progress step.

Loop Initialization

- When we add a variable, we should look for the range of values the new variable can have. We always have $x = c$ where the constant is the one we're replacing. Initialization should provide a second value d the variable can have.
- If c and d form a natural boundary for a range of values, say $c \leq d$, then we can initialize $x := d$ and the range for x would be $c \leq x \leq d$. Progressing toward termination would take x from d down to c. Symmetrically, if $d \leq c$, then progression would take x from d up to c.

- Note we might find a d to initialize x to yet still have no obvious range of values for x.
- Another possibility is that there's no clear value we can initialize x to. When that happens, the candidate invariant fails.

- ***Example 1***: Summation loops
  - The postcondition $s = sum(0, n)$ has two constants 0 and n.
  - We can try replacing n by a variable k. Initialize $k = 0$ and increase it until $k = n$, the range of k will be 0, …, n.

```
{ inv $s = sum(0, k) \land 0 \le k \le n$ } { bd $n - k$ }
while $k \ne n$ do
    … make $k$ larger …
od
{ $s = sum(0, k) \land 0 \le k \le n \land k = n$ }
{ $s = sum(0, n)$ }
```

  - Or, we can try replacing 0 by a variable k. Initialize $k = n$ and decrease it until $k = 0$. Again, the range of k will be 0, …, n.

```
{ inv $s = sum(k, n) \land 0 \le k \le n$ } { bd $k$ }
while $k > 0$ do
    … make $k$ smaller …
od
{ $s = sum(k, n) \land 0 \le k \le n \land k = 0$ }
{ $s = sum(0, n)$ }
```

- ***Example 2***: Initialization of summation loops
  - For an invariant $s = sum(0, k) \land 0 \le k \le n$, setting $k := 0$ or $k := n$ seems natural.
    - If we set $k := 0$, it's easy to establish $wp(i := 0, p) \equiv s = sum(0, 0) \land 0 \le 0 \le n$ via $s := 0$ (and the assumption $n \ge 0$).
    - But setting $k := n$ leads us to $wp(i := n, p) \equiv s = sum(0, n) \land 0 \le n \le n$, which is hard to satisfy (in fact, it's our original postcondition).

- ***Example 3***: Integer square root (replacing a constant by a variable)
  - To take the integer square root of an $n \ge 0$ means to find an x such that $x \le sqrt(n) < x+1$, or equivalently, $x^2 \le n < (x+1)^2$. Let that be q.
  - We can weaken *q [2023-03-27]* by replacing the 1 in $x+1$ with a new variable, say y, and get $x^2 \le n < (x+y)^2$ as a candidate invariant. For a range, we want $y \ge 1$. Loop initialization will

set x to something small like 2 or 1 and set y to something large (like n, if $n > 0$, or $n+1$ if $n \geq 0$)[*].

- Whatever we initialize y to, progress toward termination consists of making x larger or $x+y$ smaller. This suggests a bound function of $x+y-x$, which simplifies to y.

```
{ **inv** $x^2 \leq n < (x+y)^2 \wedge 1 \leq y$ } { **bd** $n+y-x$ }
**while** $y \neq 1$ **do** [2023-04-29]
    ... make $x$ larger or $x+y$ smaller ...
**od**
{ $x^2 \leq n < (x+y)^2 \wedge 1 \leq y \wedge y = 1$ }
{ $x^2 \leq n < (x+1)^2$ }
```

More General Replacements

- To replace a constant by a variable is the easiest way to add a parameter to a predicate. A slight generalization is to Replace an entire expression by a variable.

- ***Example 4***: Integer square root (replacing an expression by a variable)
  - For the integer square root problem, we can weaken $0 \leq x^2 \leq n < (x+1)^2$ by replacing $x+1$ with y to get $x^2 \leq n < y^2$ (and looping until $y = x+1$). For a range, we know $y \geq x+1$, and initialization is similar to that used in Example 3.

```
{ **inv** $0 \leq x^2 \leq n < y^2$ } { **bd** $y-x$ }
**while** $y > x+1$ **do**
    ... make $x$ larger or make $y$ smaller ...
**od**
{ $0 \leq x^2 \leq n < y^2 \wedge y = x+1$ }
{ $0 \leq x^2 \leq n < (x+1)^2$ }
```

The Loop Body

- Recall that a progress statement is a statement that gets us closer to termination. We need to execute at least one every iteration, along every execution path for the loop body.
- One way to organize a loop is *{ **inv** p }{ **bd** t } **while** B **do** S ; R **od***, where R is a progress step and S establishes the wp of R and the invariant:

$$\{\,\textbf{inv}\ p\,\}\ \{\,\textbf{bd}\ t\,\}\ \textbf{while}\ B\ \textbf{do}\ \{\,p \wedge B \wedge t = t_0\,\}\ S\,;\ \{\,wp(R,\ p \wedge t < t_0)\,\}\ R\ \{\,p \wedge t < t_0\,\}\ \textbf{od}$$

- For replacing a constant by a variable in particular, R takes us closer to the target constant by modifying the new variable.
- Say we want S such that $\{\,v = e_1\,\}\ S\ \{\,v = e_2\,\}$. Two simple ways are:

[*] The more we know about n, the better we might be at figuring out a range. For example, if $n > 4$, then $(n/2)^2 > n$, so we could initialize $y := n/2 - 1$.

- $\{v = e_1\}\ v := v + e_2 - e_1\ \{v = e_2\}$
- $\{v = e_1\}\ v := v * e_2 \div e_1\ \{v = e_2\}$ // (assuming e_1 divides e_2)

- One example was in the summation loop: We needed $s = sum(0, k+1)$ but had $s = sum(0, k)$. Following the first pattern above, we could guarantee progress by using

  $\{s = sum(0, k)\}\ s := s + sum(0, k+1) - sum(0, k)\ \{s = sum(0, k+1)\}$

- Of course, this isn't practical, but since $sum(0, k+1) - sum(0, k) = k+1$, we can (and did) instead use

  $\{s = sum(0, k)\}\ s := s + (k+1)\ \{s = sum(0, k+1)\}$

- ***Example 5***: (Integer log base 2) Find the largest power of 2 that is ≤ 2 .
  - Say our invariant is $y = 2 \,\hat{}\, k \leq x \wedge 0 \leq k$ (we loop ***while*** $2 * y \leq x$) and our progress step is $k := k+1$, so the *wp* of the progress step is $y = 2 \,\hat{}\, (k+1) \leq x \wedge 0 \leq k+1$.
  - So we need to establish $\{y = 2 \,\hat{}\, k \wedge \ldots\};\ y := ???\ \{y = 2 \,\hat{}\, (k+1) \wedge \ldots\}\ k := k+1\ \{y = 2 \,\hat{}\, k \wedge \ldots\}$.
  - Both patterns above for changing y can be used here:
    - One possibility is $y := y + 2 \,\hat{}\, (k+1) - 2 \,\hat{}\, k$. Since $2 \,\hat{}\, (k+1) - 2 \,\hat{}\, k = 2 \,\hat{}\, k = y$, this simplifies to $y := y+y$.
    - Another possibility is $y := y * 2 \,\hat{}\, (k+1) \div 2 \,\hat{}\, k$, which simplifies to $y := y * 2$.

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*** $y \neq 2$ [2023-03-27] with candidate invariants $x \,\hat{}\, y \leq n < (x+1)^2$ or $x^2 \leq n < (x+1) \,\hat{}\, y$.
- Initialization:
  - If we're trying $x \,\hat{}\, y \leq n$ we could try $y := 0$, and so we'd need $1 = x^0 \leq n$.
  - If we're trying $n < (x+1) \,\hat{}\, y$, the situation is much less obvious 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. Clearly this is a dead end.
- Progress step:
  - Going back to trying $x \,\hat{}\, y \leq n$ as the invariant, if we initialize y to 0, then we need to make it larger. The simplest way is $y := y+1$, and the rest of the loop body establishes the *wp* of this assignment and the invariant:

    $\{x \,\hat{}\, y \leq n < (x+1)^2 \wedge y \neq 2\}$[2023-03-27] // *Invariant* $\wedge$ *loop test*
    … // *Needed code*
    $\{x \,\hat{}\, (y+1) \leq n < (x+1)^2\}$ // *wp(progress step, invariant)*
    $y := y+1$ // *The progress step*
    $\{x \,\hat{}\, y \leq n < (x+1)^2\}$ // *The invariant*

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

Adding or Modifying Parameters more Generally

- 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
- For example [2023-03-27]
  - Variable with another variable: $q \equiv x < f(x)$ becomes $x < f(y)$ or $y < f(x)$ loop ***while*** $y \neq x$.
  - Expression with variable: $q \equiv x < f(x)$ becomes $x < y$ ***while*** $y \neq f(x)$.
  - Expression with one or more new variables.: $q \equiv x < f(x)$ becomes $g(y, z) < f(x)$ ***while*** $x \neq g(y, z)$.
- Whether any of these above candidates work will depend on what $f(\ldots)$ and $g(\ldots, \ldots)$ do.