Finding Invariants

Part 2: Deleting Conjuncts; Adding Disjuncts; Examples

CS 536: Science of Programming, Spring 2023

2023-03-29: pp. 5, 6, 9, 10

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 “Drop a conjunct” or “Add a disjunct” and to be familiar with some examples of these techniques.

C. Finding Invariants - Review

- An invariant needs to be easy to establish with initialization code, it needs to establish the post-condition (when the loop test fails), and it has to be an approximation to the *sp* and *wp* of the loop body.
- There exist various general heuristics for finding invariants, though no heuristic works easily in every situation. The general idea is to weaken the postcondition somehow*;* the kind of weakening determines the loop test.
- We've looked at getting candidate invariants by adding parameters to the postcondition.
- ***Replacing a Constant by a Variable*** is the simplest way to add a parameter.
  - We take the postcondition q, find an occurrence of a constant c in it, and replace that occurrence with a new variable x. The result is a candidate invariant p where $p\,[\,c\,/\,x\,] \equiv q$.
  - The loop header becomes ***while*** $x \neq c$, and initialization code has to set the loop variable x to a value that satisfies p.
  - For loop termination, we make progress by changing x so that it's ”closer” to c.
  - For example, we saw changes to $q \equiv s = sum\,(\,0\,,\,n\,)$ where we replaced 0 by i or n by j.
- More generally, we can add parameters by replacing one or more occurrences of an expression with one or more new variables. The expression which might be a constant or a constant-valued expression, or a variable, or any other kind of expression.

- An example was integer square root, where $x^2 \le n < (x+1)^2$ became $x^2 \le n < y^2$ by replacing $(x+1)^2$ with y, as opposed to $x^2 \le n < (x+y)^2$ when we replaced 1 with y.

D. Adding a Disjunct

- Adding a disjunct is another way to find possible invariants. Say we want to establish postcondition q. For various possible B, we can try

  $\{$ **inv** $q \vee B\}$
  while B **do**
  $\{(q \vee B) \wedge B\}$
  Loop body
  $\{q \vee B\}$
  od
  $\{(q \vee B) \wedge \neg B\}$
  $\{q\}$
- Unlike first two methods, this one is very open-ended. The other techniques we've seen all take the postcondition and modify some identified part of it, but here we can use any testable predicate for B.
- Adding a disjunct lets us, e.g., generalize a relation like $i = n$ to $i \le n$ (i.e., $i = n \vee i < n$). This is one way to understand a loop like $\{$ **inv** $i \le n$ ... $\}$ **while** $i < n$ **do**... **od** $\{ i = n \}$: The invariant $i \le n$ is the postcondition $i = n$ plus the disjunct $i < n$ added.

E. Deleting A Conjunct

- A different way to find possible invariants is ***Deleting a Conjunct***. Say postcondition is q is a conjunction, $q_1 \wedge q_2 \ldots \wedge q_n$ where $n \ge 2$.
- We get n candidate invariants, each one being q less one conjunct, and the loop runs until the conjunct is true. For $k = 1, 2, \ldots, n$, let p_k be q less q_k; i.e., $p_k \equiv q_1 \wedge \ldots q_{k-1} \wedge q_{k+1} \wedge \ldots q_n$.
- The candidate loop using p_k is

  $\{$ **inv** $p_k\}$ //where $p_k \equiv q_1 \wedge \ldots q_{k-1} \wedge q_{k+1} \wedge \ldots q_n$.
  while $\neg q_k$ **do**
  $\{p_k \wedge \neg q_k\}$ ... $\{p_k\}$
  od
  $\{p_k \wedge q_k\}$
  $\{q\}$
- Adding a disjunct is one way to view deleting a conjunct.
  - Take $q_1 \wedge q_2$ and add the disjunct $(q_1 \wedge \neg q_2)$. The result is $(q_1 \wedge q_2) \vee (q_1 \wedge \neg q_2) \Leftrightarrow q_1$, in effect deleting q_2.

- Another example: Converting $p \wedge q$ to $p \vee q$ can be viewed as a generalization of $\wedge$ to $\vee$ or as taking $p \wedge q$ to $(p \wedge q) \vee (p \wedge \neg q) \vee q$.

F. Examples of Programs and Their Invariants

- We won't cover all of these in class and you're not expected to know these in detail. It would be good to look at each example and ask yourself what technique is being used to find the invariant, the invariant relates to the bound function, and how progress toward termination is made.

Example 1: Linear Search of an Array

- This will be an example of deleting a conjunct.
- Precondition: Array b has at least n elements $(n \geq 0)$ and the value x may or may not appear in $b[0..n-1]$.
- Postcondition: We find the index k of the leftmost occurrence of x in $b[0..n-1]$. If x doesn't appear in $b[0..n-1]$, then $k = n$. Note in either case, x doesn't appear in $b[0..k-1]$. We can formalize this as

  $0 \leq k \leq n \wedge x \notin b[0..k-1] \wedge (k < n \rightarrow b[k] = x)$

  where $x \notin b[0..k-1]$ means $\forall\, 0 \leq k' < k \,.\, x \neq b[k']$. Note if $k = 0$, then $b[0..k-1] = b[0..-1]$, which is the empty sequence of values.
- Since $0 \leq k \leq n$ is short for $0 \leq k \wedge k \leq n$, there are four conjuncts we can try deleting, which yields four possible loop/test combinations. Three of them don't yield a usable invariant, but the fourth one does.
- Dropping the first conjunct, $0 \leq k$, forces $k < 0$ in the loop body, which makes referencing $b[k]$ illegal. This sounds really unpromising.

  $\{$ ***inv*** $k \leq n \wedge x \notin b[0..k-1] \wedge (k < n \rightarrow b[k] = x)\}$ ***while*** $0 > k$ ***do*** ...
- Dropping $k \leq n$ has the symmetric problem: $k > n$ in the loop body makes $b[k]$ erroneous.

  $\{$ ***inv*** $0 \leq k \wedge x \notin b[0..k-1] \wedge (k < n \rightarrow b[k] = x)\}$ ***while*** $k > n$ ***do*** ...

[Start rewrite in v.2] [1]

- Dropping $x \notin b[0..k-1]$ means we'd use ***while*** $x \in b[0..k-1]$. First problem: How do we initialize k? Using $k := 0$ makes $x \notin$ the empty segment $b[0..k-1]$. Using $k := 1$ requires $b[0] = x$, which can be false, and using $k := n$ requires us to know $x \in b[0..n-1]$, which we don't.
- Dropping the fourth conjunct, $k < n \rightarrow b[k] = x$, however, works well.

  $\{$ ***inv*** $0 \leq k \leq n \wedge x \notin b[0..k-1]\}$ ***while*** $\neg(k < n \rightarrow b[k] = x)$ ***do*** ...
- Now we can rewrite $\neg(k < n \rightarrow b[k] = x)$ as $(k < n$ && $b[k] \neq x)$, where && is the short-circuiting operator found in C etc.: B_1 && $B_2 \equiv$ ***if*** B_1 ***then*** B_2 ***else*** F ***fi***.

---

[1] In class I said we could initialize $k=0$ but that's backward - we need $x \in b[0..k-1]$, not $\notin$.

- Initialization is easy: $k := 0$, since its *wp* is $0 \le 0 \le n \wedge x \notin b[0..0-1]$. The only nontrivial part is $n \ge 0$, which will be the initial precondition.

[End rewrite]

- Since k starts out at 0 and must increase to n, a progress step of $k := k+1$ seems pretty reasonable. The loop body so far is

  $\{ p \wedge k < n \wedge b[k] \neq x \}$ // Invariant∧ loop test
  ??? // Code to write
  $\{ 0 \le k+1 \le n \wedge x \notin b[0..k+1-1] \}$ // *wp* of progress step
  $k := k+1$ // Progress step
  $\{ 0 \le k \le n \wedge x \notin b[0..k-1] \}$ // Invariant

  where *???* is code that must take us from the precondition of the loop body to the *wp* of the loop body. But it turns out that the precondition implies the *wp*, so no code is needed.

- Convergence is easy: Since p includes $k \le n$ and k gets incremented, we can use $n-k$. So the whole loop is

  $\{ n \ge 0 \}$
  $k := 0$;
  $\{$ **inv** $p \equiv 0 \le k \le n \wedge x \notin b[0..k-1] \wedge n-k \ge 0 \}$ $\{$ **bd** $n-k \}$
  while $k < n$ && $b[k] \neq x$ **do**
  $\{ (0 \le k \le n \wedge x \notin b[0..k-1] \wedge n-k \ge 0) \wedge k < n \wedge b[k] \neq x \wedge n-k = t_0 \}$
  $\{ 0 \le k+1 \le n \wedge x \notin b[0..k+1-1] \wedge n-(k+1) < t_0 \}$
  $k := k+1$
  $\{ (0 \le k \le n \wedge x \notin b[0..k-1]) \wedge n-k < t_0 \}$
  od
  $\{ 0 \le k \le n \wedge x \notin b[0..k-1] \wedge (k < n \rightarrow b[k] = x) \}$

Example 2: Binary Search Example (Version 1)

- Binary search is a nice example of a loop that isn't a ***for*** loop. For termination, a loose upper bound (the distance between the endpoints) suffices.
- Binary search has a small subtlety about what to do when the left and right endpoints are adjacent: Because of integer division, $midpoint = (L+(L+1)) \div 2 = L$, which implies that the distance from L to the *midpoint* doesn't always decrease.
  - We'll see a couple of ways to handle this.
    - The first way uses a sentinel value, and when $R = L+1$, the loop halts.
    - The second way allows $R = L-1$, and $R < L$ causes halting.
  - The differences between the two approaches makes the postconditions different, which in turn makes the invariants different, and (as it turns out) the loop test is different.

Binary Search version 1

- Program specification: $\{ q_0 \}\ Binsearch(b, x, n)\ \{ r \}$ where
  - $q_0 \equiv Sorted(b, n) \wedge 1 \le n < |b| \wedge b[0] \le x < b[n]$ (writing $|b|$ for $size(b)$)
  - $Sorted(b, n) \equiv \forall\, 0 \le k < n-1 < |b| - 1 \,.\, b[k] \le b[k+1]$
  - $r \equiv 0 \le L < n \wedge (b[L] = x \vee b[L] < x < b[L+1]) \wedge (found \leftrightarrow x = b[L])$
- Having $x < b[n]$ means $b[n]$ is a sentinel value, not an actual data value.
- Since b and n are named constants, $Sorted(b, n)$ holds throughout the program, so we'll omit explicitly writing it in the conditions.
- For our invariant, let's generalize the loop [2023-03-29] postcondition $b[L] \le x < b[L+1]$ to $b[L] \le x < b[R]$ where $0 \le L < R \le n$. (We replaced the expression $L+1$ by R.) In addition, let's weaken the postcondition's $(found \leftrightarrow x = b[L])$ to just implication: $(found \rightarrow x = b[L])$; this lets us have $found = F$ while we search. For the bound function, we can use $R-L$; it's a loose termination bound but that's okay.
- For the loop body, we'll begin by calculating the midpoint $m := (L+R) \div 2$ (with truncating division). The search succeeds if $b[m] = x$; we can set $found$ to true and L to m and exit the loop.
- The loop so far is

$\{ q \equiv Sorted(b, n) \wedge n \ge 1 \wedge b[0] \le x < b[n] \}$
$L := 0 ; R := n ; found := F ;$
$\{ \textbf{inv}\ p \equiv 0 \le L < R \le n \wedge (b[L] = x \vee b[L] < x < b[R]) \wedge (found \rightarrow x = b[L]) \}$
$\{ \textbf{bd}\ R-L \}$ ***while*** $\neg found \wedge R \ne L+1$ ***do***
    $\{ p \wedge \neg found \wedge R \ne L+1 \wedge R-L = t_0 \}$
    $m := (L+R) \div 2 ;$
    $\{ p_1 \equiv p \wedge \neg found \wedge R \ne L+1 \wedge R-L = t_0 \wedge m = (L+R) \div 2 \}$
    if $b[m] = x$ ***then***
        $\{ p_1 \wedge b[m] = x \}\ found := T ; L := m ; R := L+1\ \{ p \wedge R-L < t_0 \}$ [2] [2023-03-29]
    else
        $\{ p_1 \wedge b[m] \ne x \}$ *... code to be filled in ...* $\{ p \wedge R-L < t_0 \}$
    fi
    $\{ p \wedge R-L < t_0 \}$
od
$\{ p \wedge (found \vee R = L+1) \}$
$\{ 0 \le L < n \wedge (b[L] = x \vee b[L] < x < b[L+1]) \wedge (found \leftrightarrow x = b[L]) \}$

- It's easy to verify that loop initialization is correct. Loop termination is also correct: Either $found$ is true and $b[L] = x$, or $found$ is false, $R = L+1$, and $b[L] < x < b[L+1]$, indicating the search has indeed failed.

[2] In class I said I thought this was a bug because we don't know $x \ne b[L+1]$, but now I remember that that's why things are written as $b[L] = x$ **OR** $b[L] < x < b[R]$.

- The loop body calculates the midpoint m and checks $b[m]$ against x. If $b[m] = x$, the search has succeeded and we set L, R, and $found$ accordingly.
- If $b[m] \neq x$, then there are two ways to make progress toward termination: $L := m$ and $R := m$. Both assignments have $p \wedge R–L < t_0$ as the postcondition, so we can calculate the wp of each assignment and see if the current precondition $p_1 \wedge b[m] \neq x$ is sufficient. We get

  $\{ p_1 \wedge b[m] \neq x \}$ ... $\{ p[m/L] \wedge R–m < t_0 \}\ L := m\ \{ p \wedge R–L < t_0 \}$

  $\{ p_1 \wedge b[m] \neq x \}$ ... $\{ p[m/R] \wedge m–L < t_0 \}\ R := m\ \{ p \wedge R–L < t_0 \}$
- Expanding,
  - $p_1 \wedge b[m] \neq x \equiv p \wedge \neg found \wedge R \neq L+1 \wedge R–L = t_0 \wedge m = (L+R) \div 2 \wedge b[m] \neq x$
  - Since $p \equiv 0 \leq L < R \leq n \wedge (b[L] = x \vee b[L] < x < b[R]) \wedge (found \rightarrow x = b[L])$
  - Substituting, $p[m / L] \equiv 0 \leq m < R \leq n \wedge (b[m] = x \vee b[m] < x < b[R]) \wedge (found \rightarrow x = b[m])$
  - And $p[m / R] \equiv 0 \leq L < m \leq n \wedge (b[L] = x \vee b[L] < x < b[m]) \wedge (found \rightarrow x = b[L])$
- Comparing (and omitting detailed calculations), we see that to imply the wp of $L := m$, the precondition $p_1 \wedge b[m] \neq x$ is not strong enough. We need to add $(m < R \wedge b[m] \leq x \wedge R–m < t_0)$.
- Similarly, to imply the wp of $R := m$ we need to add $(L < m \wedge b[m] > x \wedge m–L < t_0)$.
- We can determine $b[m] < x$ and $b[m] > x$ with a test (we already know $b[m] \neq x$).
- It turns out that *L < R [2023-03-29]* and $R \neq L+1$ imply all four of $m < R$, $R–m < t_0$, $L < m$, and $m–L < t_0$, so the wp's are satisfied[3].
- Adding the test for $b[m] <$ or $> x$ gives us a loop body partially outlined[4] as

```
{ q ≡ Sorted(b, n) ∧ n ≥ 1 ∧ b[0] ≤ x < b[n] }
L := 0 ; R := n ; found := F ;
{ inv p ≡ 0 ≤ L < R ≤ n ∧ ((b[L] = x ∨ b[L] < x < b[R]) ∧ (found → x = b[L])) }
{ bd R–L }  while ¬found ∧ R ≠ L+1  do
    { p ∧ ¬found ∧ R ≠ L+1 ∧ R–L = t0 }
    m := (L+R) ÷ 2 ;
    { p1 ≡ p ∧ ¬found ∧ R ≠ L+1 ∧ R–L = t0 ∧ m = (L+R) ÷ 2 }
    if b[m] = x  then
        found := T ; L := m
    else if b[m] < x  then
        L := m
    else // b[m] > x
        R := m
    fi fi
    { p ∧ R–L < t0 }
od
```

[3] Quick argument for $L < m < R$: Since $L+2 \leq R$, $m = (L+R) \div 2 \geq (2*L+2) \div 2 = L+1$ and also $\leq (2*R–2) \div 2 \leq R–1$.

[4] A nice at-home activity is to completely expand the annotation.

$\{ p \wedge (found \vee R = L+1) \}$
$\{ 0 \le L < n \wedge ((b[L] = x \vee b[L] < x < b[L+1])) \wedge (found \leftrightarrow x = b[L]) \}$

Example 3: Traditional Binary Search

- For contrast, let's look at a traditional version of binary search, where we stop if $L > R$.
- We won't have a sentinel value in b, so $b[n–1]$ is the last data value, and the precondition becomes $Sorted(b, n) \wedge n \ge 1 \wedge b[0] \le x \le b[n–1]$.
- The postcondition will be different: If we end with $R < L$ (in particular $R = L–1$) then the search has failed, otherwise $b[L] = x$ as before. Again, to distinguish between failure and success, we'll use $found$ to stop the search. At termination,

  $–1 \le L–1 \le R < n \wedge (found \rightarrow b[L] = x) \wedge (\neg found \rightarrow x \notin b[0..n–1])$

  - $–1 \le L–1 \le R < n$ summarizes the properties and relationships of L and R, namely $0 \le L < n$ and either $L \le R < n$ or $R = L–1$.
- For the invariant, we want to weaken $(\neg found \rightarrow x \notin b[0..n–1])$ to something that will be true during the search. We only change L and R in ways that don't alter "Is x in $b[L..R]$?" If $R < L$, we know the search has failed because $b[L..R] = \varnothing$. We should terminate the loop if $found$ or $(R < L \wedge \neg found)$.
- Now for a bound function. We can't use $R–L$ because it can be $–1$. We can almost use $R–L+1$, except that when find $b[m] = x$, all we do is set $found := T$ and $L := m$, which doesn't necessarily decrease $R–L+1$. To take $found$ into account, define $|F| = 0$ and $|T| = 1$, then the bound function can be $R–L+1 + |\neg found|$.
- Altogether, we get the following sketch for our binary search:

```
{ n > 0 ∧ Sorted(b, n) ∧ b[0] ≤ x ≤ b[n–1] } L := 0; R := n–1; found := F;
{ inv q ≡ –1 ≤ L–1 ≤ R < n ∧ (found → b[L] = x) ∧ (x ∈ b[0..n–1] ↔ x ∈ b[L..R]) }
{ bd R–L+1 + |¬found| }
while ¬found ∧ L ≤ R do
    m := (L+R) ÷ 2;
    { q ∧ ¬found ∧ L ≤ R ∧ t = t0 ∧ m = (L+R) ÷ 2 }  // where t ≡ R–L+1 + |¬found|
    if b[m] = x then
        found := T; L := m
    else if b[m] < x then
        L := m + 1
    else // b[m] > x
        R := m – 1
    fi fi { q ∧ t < t0 }
od
{ q ∧ (found ∨ L > R) }
{ –1 ≤ L–1 ≤ R < n ∧ (found → b[L] = x) ∧ (¬found → x ∉ b[0..n–1]) }
```

Example 4: Match Across Two Arrays

- We saw the three-array version of this problem when we looked at sequential nondeterminism. The context there was finding partial solutions and combining them nondeterministically. This time, we'll concentrate on the invariant and on termination. The arrays can have different lengths, and this is reflected in the bound function. To cut down on notation, we'll just match two arrays instead of three arrays.
- We start with two sorted arrays b_1 and b_2 and want to find the least indexes i and j that make $b_1[i] = b_2[j]$; if no such values exist, we halt with $i = n \vee j = m$ where $n = |b_1|$ and $m = |b_2|$.
  - We'll use a bound function of $t(i,j) \equiv (n-i) + (m-j)$. Defining it as an actual function lets us write, e.g., $t(i+1, j)$ later on, instead of $((n-i)+(m-j))[i+1/i]$.
  - We can initialize i and j to 0, increment at least one of them with each iteration and ensure that the invariant implies $0 \le i \le n \wedge 0 \le j \le m$.
- We aren't going to change b_1 or b_2, so we'll specify $Sorted(b_1, n) \wedge Sorted(b_2, m)$ in the initial precondition, but after that, omit it as being implicit.

  $Sorted(b, n) \equiv \forall\, 0 \le k \le n-2 \,.\, b[k] \le b[k+1]$
- It wasn't mentioned earlier, but that program stopped with the first match it found, and so will this one. We can formalize the "least indexes i and j" part of the postcondition as a property that says no value to the left of $b_1[i]$ matches any value to the left of $b_2[j]$:

  $noMatch(i,j) \equiv \forall\, 0 \le i' < i \le n \,.\, \forall\, 0 \le j' < j \le m \,.\, b_1[i'] \ne b_2[j']$
- We also define $InRange(i,j) \equiv 0 \le i \le n \wedge 0 \le j \le m$, so our postcondition is

  $q \equiv InRange(i,j) \wedge noMatch(i,j) \wedge (i < n \wedge j < m \rightarrow b_1[i] = b_2[j])$
- To get an invariant, we'll drop the third conjunct $(i < n \wedge j < m \rightarrow b_1[i] = b_2[j])$.

  $\{$ **inv** $p \equiv InRange(i,j) \wedge noMatch(i,j)\}\{$ **bd** $t(i,j)\}$
  while $\neg(i < n \wedge j < m \rightarrow b_1[i] = b_2[j])$
  do ...
  od
  $\{p \wedge (i < n \wedge j < m \rightarrow b_1[i] = b_2[j])\}\{q\}$
- As in linear search (Example 1), we'll rewrite the test as $B \equiv (i < n \wedge j < m \,\&\&\, b_1[i] \ne b_2[j])$. As a conditional expression, this is **if** $i < n \wedge j < m$ **then** $b_1[i] \ne b_2[j]$ **else** F **fi**.
- Let's consider loop initialization. As we begin, $noMatch(0,0)$ is all we know about the arrays, so we should set i and j to zero.

  $\{n \ge 0 \wedge m \ge 0 \wedge Sorted(b, n) \wedge Sorted(b_2, m)\}$
  $i := 0 ; j := 0$
  $\{inRange(0,0) \wedge noMatch(0,0)\}$
  $\{$ **inv** $p \equiv InRange(i,j) \wedge noMatch(i,j)\}\{$ **bd** $t(i,j) \equiv (n-i)+(m-j)\}$
  while $i < n \wedge j < m \,\&\&\, b_1[i] \ne b_2[j]$
  do ...

od
$\{ q \equiv p \wedge b \}$ // where $b \equiv i < n \wedge j < m \rightarrow b_1[i] = b_2[j]$

- The termination requirement that the invariant imply $t(i,j) \geq 0$ follows from $InRange(i,j)$.
- To get closer to termination, we'll use either $i := i + 1$ or $j := j + 1$. So our loop body will include finding code taking us from the invariant and loop test to the *wp* of each progress statement. With invariant $p \equiv InRange(i,j) \wedge noMatch(i,j)$, and $\neg B \Leftrightarrow i < n \wedge j < m \,\&\&\, b_1[i] \neq b_2[j]$, calculating the *wp's* of the progress steps gives us

$\{ p \wedge t(i,j) = t_0 \wedge \neg B \}$	// inv, bound, and test
???	// needed code
$\{ InRange(i+1,j) \wedge\ noMatch(i+1,j) \wedge t(i+1,j) < t_0 \}$	// *wp* of progress step
$i := i+1$	// increase i to make progress
$\{ p \wedge t < t_0 \}$	// inv and decreased bound

and

$\{ p \wedge t(i,j) = t_0 \wedge \neg B \}$	// inv, bound, and test
???	// needed code
$\{ InRange(i,j+1) \wedge\ noMatch(i,j+1) \wedge t(i,j+1) < t_0 \}$	// *wp* of progress step
$j := j+1$	// increase j to make progress
$\{ p \wedge t < t_0 \}$	// inv and decreased bound

- The *wp* range requirements are easy: $InRange(i+1,j)$ and $InRange(i,j+1)$ are both implied by $InRange(i,j) \wedge \ldots\ i < n \wedge j < m$. The bound requirements with $t(i+1,j) < t_0$ and $t(i,j+1) < t_0$, where $t_0 = t(i,j)$, are established by $i := i+1$ and $j := j+1$ respectively.
- So what remains is "How do we get from $p \wedge \neg B$ to $noMatch(i+1,j)$ or to $NoMatch(i,j+1)$?" The invariant tells us that $noMatch(i,j)$ holds, so no value in $b_1[0..i-1]$ equals any value in $b_2[0..j-1]$. Certainly if $b_1[i] = b_2[j]$, then we've found a match.
- If $b_1[i] > b_2[j]$, which is $\geq b_2[0..j–1]$, then no value in $b_1[0..i]$ equals any value in $b_2[0..j-1]$, so $noMatch(i+1,j)$ holds. Note we can't say $noMatch(i+1,j+1)$ holds because we don't know whether $b_1[i]$ is >, <, or= $b_2[j–1]$.
- Symmetrically, if $b_1[j] > b_2[i]$, which is $\geq b_1[0..i–1]$, then no value in $b_2[0..j]$ equals any value in $b_1[0..i–1]$, so $noMatch(i,j+1)$ holds. We can't say $noMatch(i+1,j+1)$ because we don't know whether $b_2[j]$ is >, <, or = $b_1[i–1]$.
- As partial solutions, we have (the nondeterministic)
  $\{ p \wedge \neg B \}$ ***if*** $b_1[i] < b_2[j] \rightarrow \{ p[i+1/i] \}\, i := i+1$ ***fi*** $\{ p \}$ and [2023-03-29] tests
  $\{ p \wedge \neg B \}$ ***if*** $b_1[i] > b_2[j] \rightarrow \{ p[j+1/j] \}\, j := j+1$ ***fi*** $\{ p \}$
- Combining these gives us

  $\{ p \wedge \neg B \}$
  if $b_1[i] < b_2[j] \rightarrow \{ p[i+1/i] \}\, i := i+1$ [2023-03-29] tests
  $\square\ b_1[i] > b_2[j] \rightarrow \{ p[j+1/j] \}\, j := j+1$
  fi
  $\{ p \}$

- We can make this ***if-fi*** the body of a loop that runs ***while*** $b_1[i] \neq b_2[j]$, and we know the ***if-fi*** won't cause a domain error (where neither of the tests hold). For a deterministic version, since we know $b_1[i] \neq b_2[j]$, then if $b_1[i] > b_2[j]$ is false, then $b_2[j] > b_1[i]$ must hold. We get

  $\{p \wedge \neg B\}$ ***if*** $b_1[i] < b_2[j]$ ***then*** $i := i+1$ ***else*** $j := j+1$ ***fi*** $\{p\}$ [2023-03-29] test
- Adding this to the loop framework (initialization and test), we get

  $\{n \geq 0 \wedge m \geq 0 \wedge Sorted(b, n) \wedge Sorted(b_2, m)\}$
  $i := 0;\ j := 0$
  $\{$ ***inv*** $p \equiv InRange(i,j) \wedge noMatch(i,j) \wedge n - i + m - j \geq 0\}$ $\{$ ***bd*** $n-i+m-j\}$
  while $\neg B$ ***do*** $\{p \wedge \neg B \wedge t(i,j) = t_0\}$ // $\neg B \Leftrightarrow i < n \wedge j < m \,\&\&\, b_1[i] \neq b_2[j]$
    if $b_1[i] < b_2[j]$ ***then*** [2023-03-29] test
      $\{p \wedge \neg B \wedge t(i,j) = t_0 \wedge b_1[i] > b_2[j]\}$
      $\{(p \wedge \neg B)[i+1/i] \wedge t(i+1, j) < t_0\}$
      $i := i+1$
      $\{p \wedge t(i,j) < t_0\}$
    else
      $\{p \wedge \neg B \wedge t(i,j) = t_0 \wedge b_1[i] < b_2[j]\}$
      $\{(p \wedge \neg B)[j+1/j] \wedge t(i, j+1) < t_0\}$
      $j := j+1$
      $\{p \wedge t(i,j) < t_0\}$
    fi $\{p \wedge t(i,j) < t_0\}$
  od
  $\{p \wedge B\}$
  $\{p \wedge (i < n \wedge j < m \rightarrow b_1[i] \neq b_2[j])\}$
- This program can easily be extended to 3 or more arrays.

Example 5: Multiply Integers x and y (version 1: Slowly)

- Our specification is $\{x = x_0 \wedge y = y_0\}\ S\ \{z = x_0 * y_0\}$. ($x_0$ and y_0 are the initial values of x and y.)
- When the loop ends, we want $z = x_0 * y_0$.
- When the loop begins, we have $x_0 * y_0 = x * y$ because $x = x_0 \wedge y = y_0$.
- To get an invariant, define z so that it covers both cases: $z = x_0 * y_0 - x * y$.
  - When the loop begins, $x = x_0$ and $y = y_0$, so $x_0 * y_0 = x * y$, so we'll set $z := 0$.
  - We can end the loop if x or $y = 0$, because $z = x_0 * y_0 - x * y = x_0 * y_0 - 0$.
  - Let's assume $x_0 \geq 0$ initially, so that we can maintain $0 \leq x \leq x_0$ and make progress toward termination by moving x from x_0 toward 0. For the progress step, let's use $x := x-1$.
- Combining everything so far with $x \neq 0$ as the loop test gives us

  $\{x = x_0 \geq 0 \wedge y = y_0\}\ z := 0;$
  $\{$ ***inv*** $p \equiv x \geq 0 \wedge z = x_0 * y_0 - x * y\}$ $\{$ ***bd*** $x\}$
  while $x \neq 0$

do
$\{ p \wedge x \neq 0 \}$
...code to write ... ;
$\{ w \}$ // where $w \equiv wp(x := x-1, p)$
$x := x-1 \{ p \}$
od
$\{ p \wedge x = 0 \} \{ z = x_0 * y_0 \}$

- Above, $w \equiv wp(x := x-1, p) \equiv p[x-1/x] \equiv (z = x_0 * y_0 - (x-1) * y \wedge x-1 \geq 0)$
- The loop body precondition $p \wedge x \neq 0 \equiv (z = x_0 * y_0 - x * y \wedge x \geq 0) \wedge x \neq 0$
- Note p implies $z = x_0 * y_0 - x * y$, but w requires $z = x_0 * y_0 - (x-1) * y$.
  - So we don't have $p \wedge x \neq 0 \rightarrow w$, so we need some code between them to establish this.
  - Recall one way to change $z = e_1$ to $z = e_2$ is $z := z + (e_2 - e_1)$. Here, $e_2 - e_1$ is $(x_0 * y_0 - x * y) - (x_0 * y_0 - (x-1) * y)$, which $= x * y - (x-1) * y$, which $= y$.
  - So $\{ p \wedge x \neq 0 \} z := z + y \{ w \} x := x-1 \{ p \}$
- Our program is

  $\{ x = x_0 \geq 0 \wedge y = y_0 \} z := 0 ;$
  $\{ \textbf{inv } p \equiv z = x_0 * y_0 - x * y \wedge x \geq 0 \} \{ \textbf{bd } x \}$
  while $x \neq 0$ ***do***
  $\{ p \wedge x \neq 0 \wedge x = t_0 \} \{ p[x-1/x][z+y/z] \wedge x-1 < t_0 \}$
  $z := z + y ; \{ p[x-1/x] \wedge x-1 < t_0 \}$
  $x := x-1 \{ p \wedge x < t_0 \}$
  od
  $\{ p \wedge x = 0 \} \{ z = x_0 * y_0 \}$
- Partial correctness of this outline is easy to verify. For total correctness, we need to make sure x can be a bound expression. This is easy: The invariant contains $x \geq 0$ as a conjunct, and the loop body always decrements x.

Example 6: Multiply Integers x and y (version 2: More Quickly)

- The program just finished to multiply integers has a runtime linear in x_0.
- ***The Progress Step Governs the Runtime:*** We can get a faster multiplication program if we make progress toward $x = 0$ more quickly. What if we try $x := x \div 2$?
  - We can still use x as the bound expression: The invariant still implies $x \geq 0$, and if $x \neq 0$, then $x := x \div 2$ brings us strictly closer to 0.
- Instead of a loop body of
  - $\{ p \wedge x \neq 0 \wedge x = t_0 \} z := z + y ; x := x-1 \{ p \wedge x < t_0 \}$

  we have

  $\{ p \wedge x \neq 0 \wedge x = t_0 \} ??? \{ w_1 \} x := x \div 2 \{ p \wedge x < t_0 \}$

where

$w_1 \equiv wp(x := x \div 2, p \wedge x < t_0)$
$\equiv (p \wedge x < t_0)[x \div 2 / x]$
$\equiv p[x \div 2 / x] \wedge x \div 2 < t_0$
$\equiv (z = x_0 * y_0 - (x \div 2) * y) \wedge x \div 2 \geq 0 \wedge x \div 2 < t_0$

- The missing statement has to take us from $p \wedge x \neq 0 \wedge x = t_0$ to w_1.
  - We're already ensured that the $x \div 2 \geq 0$ and $x \div 2 < t_0$ clauses of w_1 hold:
    - p implies $x \geq 0$, so we know $x \div 2 \geq 0$.
    - $x = t_0$ and $x \geq 0 \wedge x \neq 0$ implies $x \div 2 < t_0$.
- We need code to go from $(z = x_0 * y_0 - x * y)$ in p to $(z = x_0 * y_0 - (x \div 2) * y)$ in w_1.
  - If x is even, then $(x \div 2) * (2 * y) = x * y$.
    - So $\{p \wedge even(x)\}\, y := 2 * y;\, \{w_1\}\, x := x \div 2\, \{p\}$
- But we don't know that x is even. We could check for it:

  if $even(x)$
  then *...code above, which requires x to be even ...* $\{w_1\}$
  else
  $\{p \wedge x \neq 0 \wedge odd(x)\}$??? $\{w_1\}$ // Missing code handles x odd case?
  fi
- Or we could ***force*** x to be even:

  $\{p\}$
  if $odd(x)$ ***then*** ???; $x := x-1$ ***fi***; // Missing code enables $x := x-1$ to maintain p
  $\{p \wedge even(x)\}$
  ... above code ...
  $\{w_1\}$
- But ***we already know what we can use*** before the decrement of x.
  - We've already written it once: it's $z := z + y$.
- This completes the program:

  $\{x = x_0 \wedge y = y_0 \wedge x_0 \geq 0\}$
  $z := 0$;
  $\{$***inv*** $p \equiv z = x_0 * y_0 - x * y \wedge x \geq 0\}\, \{$***bd*** $x\}$
  while $x \neq 0$ ***do***
  if $odd(x)$ ***then*** $z := z + y;\, x := x-1$ ***fi***; $\{p \wedge even(x)\}$
  $y := 2 * y;\, x := x \div 2$
  od
  $\{p \wedge x = 0\}\, \{z = x_0 * y_0\}$
- This is a program that implements multiplication by repeated addition and bit-shifting. (Multiplication and division by 2 correspond to left and right bit shifting respectively.) It does roughly $log_2(x_0)$ iterations.

Example 7: Faster Integer Square Root

- For another example of how a faster progress step speeds up a program, recall the integer square root problem (from the previous class). One change: Instead of $n–x+y$ for the bound function, we will be able to use just y because we'll always decrease it (in addition to sometimes increasing x).

  $\{$ ***inv*** $p\}\{$ ***bd*** $y\}$ // where $p \equiv x^2 \le n < (x+y)^2 \wedge y \le 1$
  while $y \ne 1$
  do $\{p \wedge y = y_0\}$
  ... code to write ...
  $\{p\}$
  od
  $\{p \wedge y = 1\}$
  $\{x^2 \le n < (x+1)^2\}$

- To make progress, we need to decrease y. Instead of decrementing y by 1 as before, this time we'll divide it by 2: Using $y := y \div 2$, makes for a binary-search-like method: We test the midpoint $(x+y \div 2)^2$ against n and make it the new left or right endpoint accordingly.

- Here's a partial proof outline:

  $\{$ ***inv*** $p \wedge y \ge 1\}\{$ ***bd*** $y\}$
  while $y \ne 1$ ***do***
  if $(x+y \div 2)^2 > n$ ***then***
  $\{0 \le x^2 \le n < (x+y \div 2)^2 \wedge y \div 2 < t_0\}$
  $y := y \div 2$
  else // $(x+y \div 2)^2 \le n$
  $\{0 \le (x+y \div 2)^2 \le n < (x+y \div 2 + (y - y \div 2))^2 \wedge (y - y \div 2) < t_0\}$
  $x := x + y \div 2 ; y := y - y \div 2$
  fi; $\{0 \le x^2 \le n < (x+y)^2 \wedge y < t_0\}$
  od
  $\{0 \le x^2 \le n < (x+y)^2 \wedge y \ge 1\} \wedge y = 1\}$
  $\{0 \le x^2 \le n < (x+1)^2\}$

- ***Notes***: The invariant implies $y \ge 1$; that with the loop test $y \ne 1$ implies $y \ge 2$. That in turn implies that $y \div 2$ and $y - y \div 2$ are both $< y$, which ensures progress whether the ***if*** test succeeds or fails.