Finding Invariants

Part 2: Deleting Conjuncts; Adding Disjuncts

CS 536: Science of Programming, Spring 2020

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 postcondition (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.
• Replacing a Constant by a Variable involves finding a constant (literal or named) $c$ that appears in the postcondition $q$ and replacing it by a new variable $x$. If $\text{invariant}[c/x] = \text{postcondition}$, then the loop test is $\text{while } x \neq c$. The invariant has to be easily established by assigning $x$ (and any other loop variables) to some initial value, and we have to be able to approach terminal of the loop by making $x$ "closer" to $c$.
• Replacing a Constant by a Variable is an instance of a more general technique, Adding or Modifying Parameters, in which we replace an expression by an expression that includes one or more new variables. Examples of these two included summation, factorial, integer square root, and integer log base 2.
D. Deleting A Conjunct

- Deleting a conjunct is another way to find possible invariants. To use it, we need a postcondition that is the conjunction of multiple conjuncts. Say postcondition is $r$ is $p_1 \land p_2 \ldots \land p_n$ where $n \geq 2$.
- Let, $\text{Less}(r, k) = (p_1 \land p_2 \ldots \land p_{k-1}) \land (p_{k+1} \land \ldots \land p_n)$. I.e., $r$ “less” the conjunct $p_k$.
- There are $n$ possible invariants, one for each conjunct. In general, for conjunct $k$ we have
  
  $\{ \text{inv } p = \text{Less}(r, k) \}$

  while $\neg p_k$ do

  $(p \land \neg p_k) \ldots \{ p \}$

  od

  $(p \land p_k) \{ r \}$

Example 1: Linear Search of an Array

- 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

  \[ 0 \leq k \leq n \land x \notin b[0..k-1] \land (k < n \to b[k] = x) \]

  where $x \notin b[0..k-1]$ means $\forall 0 \leq k', k \cdot x \notin b[k']$. Note if $k = 0$, then $b[0..k-1] = b[0..-1]$ is the empty sequence of values.
- Since $0 \leq k \leq n$ is short for $0 \leq k \land k \leq n$, 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.

- $\{ \text{inv } k \leq n \land x \notin b[0..k-1] \land (k < n \to b[k] = x) \}$ // Drop the conjunct $0 \leq k$

  while $0 > k$ do ...

  If we use this, then in the loop body we have $k < 0$, which makes referencing $b[k]$ illegal. This sounds really unpromising.

- $\{ \text{inv } 0 \leq k \land x \notin b[0..k-1] \land (k < n \to b[k] = x) \}$ // Drop the conjunct $k \leq n$

  while $n > k$ do ...

  This has the symmetric problem: $k$ is too large to be an index.

- $\{ \text{inv } 0 \leq k \land x \notin b[0..k-1] \land (k < n \to b[k] = x) \}$ // Drop the conjunct $x \notin b[0..k-1]$

  while $x \in b[0..k-1]$ do ...

  There are two problems with this proposed invariant. First, how do we initialize $k$?

  Setting $k := 1$ would require $b[0] = x$, and setting $k := n$ requires knowing that $x$ doesn’t appear in $b[0..n-1]$. The second problem is that the test $x \in b[0..k-1]$ takes time proportional to $k$, since we’ll need a loop or recursion to write it.

- The fourth possibility, however, works well. Here, we drop $k < n \to b[k] = x$.  


Adding a Disjunct

• Adding a disjunct is another way to find possible invariants. Say we want to establish postcondition \( r \). For various possible \( B \), we can try

\[
\{ \text{inv } r \lor B \}\
\text{ while } B \text{ do }\
\{ (r \lor B) \land B \} \text{ Loop body } (r \lor B)\
\text{ od } \{ (r \lor B) \land \neg B \} \{ r \}
\]

• Unlike first two methods, this one is very open-ended — you can use any testable predicate for \( B \).

• Adding a disjunct lets us, e.g., generalize a relation like \( i = n \) to \( i \leq n \) (i.e., \( i = n \lor i < n \)). This is one way to understand a loop like \( \{ \text{inv } i \leq n \ldots \} \text{ while } i < n \text{ do } \ldots \text{ od } \{ i = n \} \): The postcondition \( i = n \) gets the disjunct \( i < n \) added and becomes \( i \leq n \) in the invariant.
• Adding a disjunct is one way to view deleting a conjunct: Changing \( p \land q \) to \((p \land q) \lor (p \land \neg q)\) yields something \( \leftrightarrow \) just \( p \).

• Converting \( p \land q \) to \( p \lor q \) can be viewed as a generalization of \( \land \) to \( \lor \) or as taking \( p \land q \) to \((p \land q) \lor (p \land \neg q) \lor q\).

\[ \]

**F. 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.

• Program specification: \( \{q_0\} \) Binsearch \((b, x, n) \{r\}\) where

  \[
  q_0 = \text{Sorted}(b, n) \land 1 \leq n < |b| \land b[0] \leq x < b[n] \quad \text{(writing } b \text{ | for size(b))}
  \]

  \[
  \text{Sorted}(b, n) = \forall 0 \leq i < n-1 < |b| \cdot -1. \ b[i] \leq b[i+1].
  \]

  \[
  r = 0 \leq L < n \land b[L] \leq x < b[L+1] \land (f \leftarrow x = b[L]) \quad \text{// } f = \text{"have we found } x\text{?"
  }
  \]

• Having \( x < b[n] \) means \( b[n] \) is a sentinel value, not an actual data value.

• Let’s treat \( b \) and \( n \) as named constants so that \( \text{Sorted}(b, n) \) can be used anywhere and doesn’t have to be part of the invariant.

• For our invariant, let’s generalize the loop precondition \( b[L] \leq x < b[L+1] \) to \( b[L] \leq x < b[R] \) where \( 0 \leq L \leq R \leq n \). (We replaced the expression \( L+1 \) by \( R \)) In addition, let’s weaken the postcondition’s \((f \leftarrow x = b[L])\) to just implication: \((f \rightarrow x = b[L])\); this lets us have \( f = F \) 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 \( m = (L+R)/2 \) (with truncating division). The search succeeds if \( b[m] = x \); we can set \( f \) to true and \( L \) to \( m \) and exit the loop.

• The loop so far is

  \[
  q = \text{Sorted}(b, n) \land n \geq 1 \land b[0] \leq x < b[n] \\
  L := 0 \ ; \ R := n \ ; \ f := F \ ; \ {\text{inv } \neg f \land R \neq L+1} \ do \\
  \{ \rho \land \neg f \land R \neq L+1 \land R-L = t_0 \} \\
  m := (L+R)/2 \ ; \\
  \{ \rho \land \neg f \land R \neq L+1 \land R-L = t_0 \land m = (L+R)/2 \} \\
  \text{if } b[m] = x \text{ then} \\
  \quad f := T \ ; \ L := m \ ; \ R := L+1 \\
  \text{else} \\
  \quad // \ldots \text{to be filled in ...} \\
  \fi \\
  \{ \rho \land R-L < t_0 \} \\
  \od \\
  \{ \rho \land (f \lor R = L+1) \} \\
  \{0 \leq L < n \land b[L] \leq x < b[L+1] \land (f \leftarrow x = b[L])\}
One possible at-home activity is to completely expand the annotation.

At termination, either $f$ is true and $b[L] = x$, or $f$ is false, $R = L + 1$, and $b[L] < x < b[L + 1]$, indicating the search has indeed failed.

If $b[m] ≠ x$, we make progress toward termination by setting $L$ or $R$ to $m$. To reestablish the invariant, we need

\[
\{ p[m/L] ∧ R-m < t_0 \} L := m \{ p ∧ R-L < t_0 \}
\]

or

\[
\{ p[m/R] ∧ m-L < t_0 \} R := m \{ p ∧ R-L < t_0 \}
\]

- In the first case, we need $0 ≤ m < R ≤ n ∧ b[m] ≤ x < b[R] ∧ (f → x = b[m] ∧ R-m < t_0)$.
- In the second case, we need $0 ≤ L < m ≤ n ∧ b[L] ≤ x < b[m] ∧ (f → x = b[L] ∧ m-L < t_0)$.

We already know $b[m] ≠ x$, so testing $b[m] < x$ vs $b[m] > x$ will establish which of these two cases we are in. We also need $R-m < t_0$ or $m-L < t_0$, where $t_0 = R-L$; these both follow from $L < m < R$, which in turn follows from $L < R ∧ R ≠ L+1$.

(Quick argument for $L < m < R$: Since, $L+2 ≤ R$, $m = (L+R)/2$ is $≥ (2*L+2)/2 = L+1$ and also $≤ (2*R-2)/2 < R$.)

This gives us a loop body partially outlined* as

\[
\{ p ∧ ¬f ∧ R ≠ L+1 ∧ R-L = t_0 \}
\]

\[
m := (L+R)/2 ;
\]

\[
\{ p_1 = p ∧ ¬f ∧ R ≠ L+1 ∧ R-L = t_0 ∧ m = (L+R)/2 \}
\]

if $b[m] = x$ then

\[
f := T ; L := m
\]

else if $b[m] < x$ then

\[
L := m
\]

else // $b[m] > x$

\[
R := m
\]

end if

\[
\{ p ∧ R-L < t_0 \}
\]

---

**Example 3: Traditional Binary Search**

- For contrast, let's look at a traditional version of binary search, where we stop if $L > R$.
- We begin with almost the same precondition, $\text{Sorted}(b, n) ∧ n ≥ 1 ∧ b[0] ≤ x ≤ b[n]$. (We weakened $x < b[n]$ to $x ≤ b[n]$.)
- 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 $f$ to stop the search. At termination,

\[
-1 ≤ -L-1 ≤ R < n ∧ (f → b[L] = x) ∧ (¬f → x \not∈ b[0..n-1])
\]

- (The first conjunct, $-1 ≤ L-1 ≤ R < n$, summarizes the properties and relationships of $L$ and $R$, namely $0 ≤ L ≤ n$ and either $L ≤ R < n$ or $R = L-1$.)

---

* One possible at-home activity is to completely expand the annotation.
For the invariant, we want to weaken \( \neg f \rightarrow x \not\in b[0..n-1] \) to something that will be true during the search. We only change \( L \) and \( R \) in ways that don’t alter “Is \( x \) in \( b[L..R] \)?” I’ll use \( (x \in b[L..R] \leftrightarrow x \in b[0..n-1]) \) with the understanding that when \( R = L-1 \) then \( b[L..R] = b[L..L-1] = \emptyset \). This way, if \( R < L \), we know the search has failed. We should terminate the loop if \( f \) or \( (R < L \text{ and } \neg f) \).

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 \( \text{find} \ b[m] = x \), all we do is set \( f := \text{true} \) and \( L := m \), which doesn’t necessarily decrease \( R-L+1 \). To take \( f \) into account, define \(|F| = 0\) and \(|T| = 1\), then we can use \( R-L+1+|\neg f| \) for the bound function.

Altogether, we get the following sketch for our binary search:

\[
\begin{align*}
\{n > 0 \land \text{Sorted}(b,n) \land b[0] \leq x \leq b[n-1]\} \\
L := 0; R := n-1; f := F; \\
\{\text{inv } q = -1 \leq L-1 \leq R < n \land (f \rightarrow b[L] = x) \land (x \in b[0..n-1] \leftrightarrow x \in b[L..R])\} \\
\{bd \ R-L+1+|\neg f|\} \\
\text{while } \neg f \land L \leq R \text{ do} \\
\quad m := (L+R)/2; \\
\quad \{q_1 = q \land \neg f \land L \leq R \land R-L+1+|\neg f| = t_0 \land m = (L+R)/2\} \\
\quad \text{if } b[m] = x \text{ then} \\
\quad \quad f := T ; L := m \\
\quad \text{else if } b[m] < x \text{ then} \\
\quad \quad L := m+1 \\
\quad \text{else } // b[m] > x \\
\quad \quad R := m-1 \\
\quad \text{fi fi} \\
\text{od} \\
\{q \land (f \lor L > R)\} \\
\{-1 \leq L-1 \leq R < n \land (f \rightarrow b[L] = x) \land (\neg f \rightarrow x \not\in b[0..n-1])\}
\]

**Example 4: Match across two lists**

- We have 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 should halt with \( i = n \lor j = m \).
  - We’ll use a bound function of \((n-i) + (m-j)\). We can initialize \( i \) and \( j \) to 0, increment at least one of them with each iteration and ensure that the invariant implies \( 0 \leq i \leq n \land 0 \leq j \leq m \).
  - We aren’t going to change \( b_1 \) or \( b_2 \), so we can specify \( \text{Sorted}(b_1, n) \land \text{Sorted}(b_2, m) \) in the initial precondition, but after that we can omit it as being implicit.
  - \( \text{Sorted}(b, n) = \forall 0 \leq k \leq n-2 . \ b[k] \leq b[k+1] \)
  - \( \text{NoMatch}(i, j) = \forall 0 \leq i' < i \leq n . \ \forall 0 \leq j' < j \leq m . \ b_1[i'] \neq b_2[j'] \)
  - Also, let \( \text{InRange}(i, j) = 0 \leq i \leq n \land 0 \leq j \leq m \), then our postcondition is
\[ q = \text{InRange}(i, j) \land \text{NoMatch}(i, j) \land (i < n \land j < m \rightarrow b_1[i] = b_2[j]) \]

- To get an invariant, we'll drop the third conjunct \((i < n \land j < m \rightarrow b_1[i] = b_2[j])\) (or, equivalently, add a disjunct of \((i < n \land j < m \rightarrow b_1[i] \neq b_2[j])\)).

\[
\{ \text{inv} \ p = \text{InRange}(i, j) \land \text{NoMatch}(i, j) \}
\]

\[
\text{while } \neg((i < n \land j < m \rightarrow b_1[i] = b_2[j]) \text{ do ... od}
\]

\[
\{ p \land (i < n \land j < m \rightarrow b_1[i] = b_2[j]) \} \{ q \}
\]

As in linear search (Example 1), we'll rewrite the test as \(B = (i < n \land j < m \land b_1[i] \neq b_2[j])\). As a conditional expression, this is \(\text{if } i < n \land j < m \text{ then } b_1[i] \neq b_2[j] \text{ else } fi\).

- Before writing the loop body, let's consider initialization. As we begin, NoMatch(0, 0) is all we know about the arrays, we can set \(i\) and \(j\) to zero.

\[
\{ n \geq 0 \land m \geq 0 \land \text{Sorted}(b, n) \land \text{Sorted}(b_2, m) \}
\]

\[
i := 0; j := 0 \{ \text{Range}(0, 0) \land \text{NoMatch}(0, 0) \}
\]

\[
\{ \text{inv} \ p = \text{InRange}(i, j) \land \text{NoMatch}(i, j) \} \{ bd \ (n-i) + (m-j) \}
\]

\[
\text{while } i < n \land j < m \land b_1[i] \neq b_2[j] \text{ do ... od}
\]

\[
\{ q = p \land B \} \quad // \text{where } B = i < n \land j < m \rightarrow b_1[i] = b_2[j]
\]

- For termination, we need the invariant to imply \((n-i) + (m-j) \geq 0\), which follows from InRange\((i, j)\).

- To get closer to termination, either \(i := i+1\) or \(j := j+1\) will do. So our loop body will include finding code taking us from the invariant and loop test to the wp of each progress statement

\[
\{ p \land \neg B \} \text{ ??? } \{ \text{InRange}(i+1, j) \land \text{NoMatch}(i+1, j) \} i := i+1 \{ p \}
\]

\[
\{ p \land \neg B \} \text{ ??? } \{ \text{InRange}(i, j+1) \land \text{NoMatch}(i, j+1) \} j := j+1 \{ p \}
\]

- (Recall \( p = \text{InRange}(i, j) \land \text{NoMatch}(i, j) \) and \( \neg B \Rightarrow i < n \land j < m \land b_1[i] \neq b_2[j]\).)

\[
\text{InRange}(i, j) \land ... i < n \land j < m ... \text{ implies } \text{InRange}(i+1, j) \land \text{InRange}(i, j+1).
\]

- So the question for \(i := i+1\) is how to get from NoMatch\((i, j) \land b_1[i] \neq b_2[j]\) to NoMatch\((i+1, j)\) If we assume \(p \land \neg B\), then \(b_1[i] > b_2[j]\) will ensure NoMatch\((i+1, j)\) because the elements \(b_2[j], b_2[j-1], ..., b_2[0]\) are nondecreasing and the loop test included \(b_1[i] \neq b_2[j]\).

- Altogether, we get \(p \land \neg B\) if \(b_1[i] > b_2[j] \rightarrow \{ p[i+1/i] \} i := i+1 \text{ fi } \{ p \}\) as one (nondeterministic) case for the loop body.

- Symmetrically, \(b_2[j] > b_1[i]\) will ensure NoMatch\((i, j+1)\). This gives us another loop body case:

\[
\{ p \land \neg B \} \text{ if } b_2[j] > b_1[i] \rightarrow \{ p[j+1/j]\} j := j+1 \text{ fi } \{ p \}
\]

- If we combining these two cases with nondeterministic if-fi, we get the (pleasingly?) symmetric

\[
\{ p \land \neg B \}
\]

\[
\text{if } b_1[i] > b_2[j] \rightarrow \{ p[i+1/i] \} i := i+1
\]

\[
\text{or } b_2[j] > b_1[i] \rightarrow \{ p[j+1/j]\} j := j+1
\]

\[
\text{fi } \{ p \}\]
• Since the loop test implies $b_1[i] \neq b_2[j]$, we've covered all the possible cases and also ensured that the if-fi won't cause a domain error (where none of the tests hold). This means the nondeterministic if-fi above can be used as the loop body. To rewrite the if-fi deterministically, since we know $b_1[i] \neq b_2[j]$, if $b_1[i] > b_2[j]$ is false, then $b_2[j] > b_1[i]$ must hold. This gives us

$$\{p \land \neg B\} \text{if } b_1[i] > b_2[j] \text{ then } i := i+1 \text{ else } j := j+1 \text{ fi } \{p\}$$

• Adding this to the loop framework (initialization and test), we get

$$\{n \geq 0 \land m \geq 0 \land \text{Sorted}(b, n) \land \text{Sorted}(b_2, m)\}$$

$$i := 0; j := 0$$

$$\{\text{inv } p = \text{InRange}(i, j) \land \text{NoMatch}(i, j) \land n-i + m-j \geq 0\} \{bd \ n-i + m-j\}$$

$$\text{while } \neg B \text{ do } \{p \land \neg B \land n-i + m-j = t_0\} \quad \text{// } \neg B \iff i < n \land j < m \land b_1[i] \neq b_2[j]$$

if $b_1[i] > b_2[j]$ then

$$\{p \land \neg B \land n-i + m-j = t_0 \land b_1[i] > b_2[j]\}$$

$$(p \land \neg B)[i+1/i] \land n-(i+1) + m-j < t_0 \} i := i+1 \{p \land n-i + m-j < t_0\}$$

else

$$\{p \land \neg B \land n-i + m-j = t_0 \land b_1[i] < b_2[j]\}$$

$$(p \land \neg B)[j+1/j] \land n-i + m-(j+1) < t_0 \} j := j+1 \{p \land n-i + m-j < t_0\}$$

fi

od $\{p \land B\} \{p \land (i < n \land j < m \to b_1[i] \neq b_2[j])\}$

• One interesting property of the nondeterministic solution is that it's easily extendable to more than two arrays. We can add a third array, $b_3$ with index $k$ and size $p$. The definitions of InRange and NoMatch get extended, as does the bound, which becomes $n-i + m-j + p-k \geq 0$

$$\{p \land \neg B\}$$

if $b_1[i] > b_2[j]$ then

$\{p[i+1/i]\} i := i+1$

$\circ b_2[j] > b_1[i] \to \{p[j+1/j]\} j := j+1$

$\circ b_3[k] > b_2[j] \to \{p[k+1/k]\} k := k+1$

fi

$p$

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

• Our specification is $\{x = x_0 \land y = y_0\} S \{z = x_0 \ast y_0\}$. ($x_0$ and $y_0$ are the initial values of $x$ and $y$)

• When the loop ends, we want $z = x_0 \ast y_0$.

• When the loop begins, we have $x_0 \ast y_0 = x \ast y$ because $x = x_0 \land y = y_0$.

• To get an invariant, we can reframe the definition of $z$ so that it covers both cases: $z = x_0 \ast y_0 - x \ast y$.

• When the loop begins, $x = x_0$ and $y = y_0$, so $x_0 \ast y_0 = x \ast y$, so we'll set $z := 0$.

• We can end the loop if $x$ or $y = 0$, because $z = x_0 \ast y_0 - x \ast y = x_0 \ast y_0 - 0$.

• If $x_0 \geq 0$ initially, then we can maintain $0 \leq x \leq x_0$, and we make progress by moving $x$ from $x_0$ toward 0. Let's use $x := x - 1$ as the progress step toward termination.

• Combining everything so far with $x \neq 0$ as the loop test gives us
• \{x = x_0 \geq 0 \land y = y_0\} z := 0;
• \{inv\ p = x \geq 0 \land z = x_0 \cdot y_0 - x \cdot y\} \{bd\ x\}
while x \neq 0
do
\{p \land x \neq 0\} code to write);
\{w\} x := x - 1 \{p\} // where w = wp(x := x - 1, p)
\{x = x_0 \land y = y_0\}
• Above, \(w = wp(x := x - 1, p) = p[x - 1/x] = (z = x_0 \cdot y_0 - (x - 1) \cdot y \land x - 1 \geq 0)\)
• The loop body precondition \(p \land x \neq 0 = (z = x_0 \cdot y_0 - x \cdot y \land x \geq 0) \land x \neq 0\)
• Note \(p\) implies \(z = x_0 \cdot y_0 - x \cdot y\), but \(w\) requires \(z = x_0 \cdot y_0 - (x - 1) \cdot y\).
  • So we don't have \(p \land x \neq 0 \rightarrow \lnot w\), 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 \((x_0 \cdot y_0 - x \cdot y) - (x_0 \cdot y_0 - (x - 1) \cdot y) = x \cdot y - (x - 1) \cdot y = y\)
  • So \(p \land x \neq 0\) z := z + y \{w\} x := x - 1 \{p\}
• Our program is
\{x = x_0 \geq 0 \land y = y_0\} z := 0;
• \{inv\ p = z = x_0 \cdot y_0 - x \cdot y \land x \geq 0\} \{bd\ x\}
while x \neq 0
do
\{p \land x \neq 0 \land x = t_0\} \{p[x - 1/x][z + y/z] \land x - 1 < t_0\}
\{z := z + y; \{p[x - 1/x] \land x - 1 < t_0\}
\{x := x - 1 \{p \land x < t_0\}
\{p \land x = 0\} z := x_0 \cdot 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.
• The invariant contains \(x \geq 0\) as a conjunct, so \(\text{invariant} \rightarrow \text{bound} \geq 0\) holds.
• The loop body decrements \(x\), so \(\{\text{invariant} \land \text{loop test} \land \text{bound} = t_0\} \text{ loop body} \{\text{bound exp} < t_0\}\) holds.

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

Progress Step Governs Runtime
• The program just finished to multiply integers has a runtime linear in \(x_0\). We can get a faster multiplication program if we make progress toward \(x = 0\) more quickly.
• What if we try \(x := x + 2\)?
  • We can still use \(x\) as the bound expression: The invariant still implies \(x \geq 0\), and if \(x \neq 0\), then \(x := x + 2\) brings us strictly closer to 0.
• Instead of a loop body of

\{ p \land x \neq 0 \land x = t_0 \} \ z := z+y; \ x := x-1 \{ p \land x < t_0 \}

we have

\{ p \land x \neq 0 \land x = t_0 \} ??? \{ w_1 \} \ x := x+2 \{ p \land x < t_0 \}

where \( w_1 = wp(x := x+2, p \land x < t_0) \)

\( = (p \land x < t_0)[x+2/x] \)

\( = p[x+2/x] \land x+2 < t_0 \)

\( = (z = x_0*y_0 - (x+2)*y) \land x+2 \geq 0 \land x+2 < t_0 \)

• The missing statement has to take us from \( p \land x \neq 0 \land x = t_0 \) to \( w_1 \).

• We're already ensured that the \( x+2 \geq 0 \) and \( x+2 < t_0 \) clauses of \( w_1 \) hold:

  • \( p \) implies \( x \geq 0 \), so we know \( x+2 \geq 0 \).
  • \( x = t_0 \) and \( x \geq 0 \land x \neq 0 \) implies \( x+2 < t_0 \).

• We need code to go from \( (z = x_0*y_0 - x*y) \) in \( p \) to \( (z = x_0*y_0 - (x+2)*y) \) in \( w_1 \).

  • If \( x \) is even, then \( (x+2)*(2*y) = x*y \).
  • So \( (p \land even(x)) \) \( y := 2*y; \{ w_1 \} \ x := x+2 \{ p \} \)

• But we don't know that \( x \) is even. We could check for it:

  \begin{verbatim}
  if even(x)
    then ... code above (requires x to be even) ... {w_1}
  else
    {p \land x \neq 0 \land odd(x)} ??? {w_1}
  fi
  \end{verbatim}

• Or we could force \( x \) to be even:

  \begin{verbatim}
  {p} if odd(x) then ??? ; x := x-1 fi; {p \land even(x)}
  ... above code ... {w_1}
  \end{verbatim}

• 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 \land y = y_0 \land x_0 \geq 0 \}
  z := 0;
  \{ inv p = z = x_0*y_0 - x*y \land x \geq 0 \} \{ bd x \}

  while \( x \neq 0 \) do
    if odd(x) then \( z := z+y; \ x := x-1 \) fi; \{ p \land even(x) \}
  y := 2*y; \ x := x+2
  od
  \{ p \land 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: 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). The basic loop was

\[
\{ \text{inv } x^2 \leq n < (x+y)^2 \land 1 \leq y \} \{ \text{bd } y \}
\]

\[
\text{while } y \neq 1 \text{ do ... od}
\]

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

- To make progress, we need to decrease \( y \). Two obvious techniques are \( y := y - 1 \) and \( y := y \div 2 \).

Let's use \( y := y \div 2 \), in 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:

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

\[
\text{while } y \neq 1 \text{ do}
\]

\[
\text{if } (x+y \div 2)^2 > n \text{ then}
\]

\[
\{ 0 \leq x^2 \leq n < (x+y \div 2)^2 \land y+2 < t_0 \}
\]

\[
y := y+2
\]

\[
\text{else} \quad \text{// } (x+y \div 2)^2 \leq n
\]

\[
\{ 0 \leq (x+y \div 2)^2 \leq n < (x+y \div 2 + (y-y\div2))^2 \land (y-y\div2) < t_0 \}
\]

\[
x := x+y \div 2; \quad y := y - y\div2
\]

\[
\text{fi; } \{ 0 \leq x^2 \leq n < (x+y)^2 \land y < t_0 \}
\]

\[
\text{od}
\]

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

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

- Notes: The invariant implies \( y \geq 1 \); that with \( y \neq 1 \) implies \( y \geq 2 \). That in turn implies \( y \div 2 \) and \( y-y\div2 \) are both \(< y \), which ensures progress whether the if test succeeds or fails.