Finding Loop Invariants and Creating Loops

- It is not always easy to create a loop that works correctly, and finding a good loop invariant is usually the most important part while creating a loop. Usually, after finding the loop invariant, the rest of the loop comes naturally.
- Like loop bound expressions, there is no algorithm to find a loop invariant; but there are some heuristics, and they don't work in all cases.

- While introducing loop invariants, we discussed some basic needs of a loop invariant. Let loop $W \equiv \{\textbf{inv}\ p\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}$ and if we need to show $\{p_0\}\ W\ \{q\}$, we need:
  - $p_0 \Rightarrow p$
  - $\{p \wedge B\}\ S\ \{p\}$ is provable.
  - $p \wedge \neg B \Rightarrow q$

  When we create a loop in a program, we don't have the loop body S and we don't have the loop condition B yet, all we have are conditions p_0 and q.

- To get a loop invariant, we ***usually start with q and try to weaken it***. Here are the reasons:
  1) q is the postcondition here, it is usually an expectation from the user.
  2) We observe that $p \wedge \neg B \Rightarrow q$, but p itself should not be stronger than q: or else, there is no need to have W and the postcondition is already satisfied.
  3) While weakening the postcondition q, we usually can get some insight on B at the same time.

- Here are some "tricks" we usually use to weaken a predicate:
  1) Replacing a constant or an expression with a variable: for example, we can weaken $2 + x = z$ to $y + x = z$ where y is a variable (whose domain includes 2).
  2) Adding a disjunct: for example, we can weaken $x = 2$ to $x \leq 2$ (because $x \leq 2 \Leftrightarrow x < 2 \vee x = 2$).
  3) Removing a conjunct: for example, we can weaken $1 < x \leq 3$ to $x \leq 3$ (because $1 < x \leq 3 \Leftrightarrow 1 < x \wedge x \leq 3$).

- Using these tricks won't guarantee a loop invariant, they can only find some candidate for loop invariants. We still need to try to create loops with these candidates to see whether they make sense.

1. Create a program (in a full proof outline under total correctness) that calculates the function $sum(0, n)$, which sums up consecutive natural numbers from 0 to n and stores the result $s = sum(0, n)$.

   1) First, Let's figure out the pre- and post- conditions from the question:
      - To calculated $sum(0, n)$, n should be a named constant defined in the precondition: $n \geq 0$
      - Postcondition is straightforward:

      This is what we get so far:

      $$\{n \geq 0\} \ldots \text{our program} \ldots \{s = sum(0, n)\}$$

   2) It looks like the program needs a loop to sum up numbers in a range. So secondly, let's try to find some possible loop invariants. Here, replacing a constant with a variable seems to be a good idea. There are two constants 0 and n in the postcondition, so we have two loop invariant candidates. We have already seen

the loop created with a loop invariant where n is substituted by a variable; so here let's try to substitute n by a variable.

If we replace 0 by a variable k, we get $s = sum(k, n)$. Since we need the sum of the first n positive integers, and k will be equal to 0 after the loop, so k should be in the range from 0 to n: we can initialize $k = n$ and decrease it in each iteration until $k = 0$. And we will get a loop that looks like this:

$\{n \geq 0\}$ … some code …
$\{\textbf{inv } s = sum(k, n) \wedge 0 \leq k \leq n\}\{\textbf{bd } k\}$
while $k \neq 0$ **do**
   … make k smaller and something else …
od
$\{s = sum(k, n) \wedge 0 \leq k \leq n \wedge k = 0\}$ $\# \, p \wedge \neg B$
$\{s = sum(0, n)\}$

- When we replace a constant c / an expression e with a variable k, we can consider the range of k (if we have enough information). We usually end the program with $k = c$ (or the value of e); if we can figure out the initial value of k before the program starts (say d), then c (or the value of e) and d is the range of variable k.

3) Next, let's consider how to initialize the loop. The program starts with $n \geq 0$ and it currently doesn't imply the loop invariant, so we need to initialize some variables before the loop.

   We are most likely starting the loop with $k = n$; which means we need $s = sum(k, n) \wedge 0 \leq k \leq n \wedge k = n$ before the first iteration, so $s = n$. Then:

   $\{n \geq 0\}\; k \coloneqq n;\; \{n \geq 0 \wedge k = n\}\; s \coloneqq n;$
   $\{n \geq 0 \wedge k = n \wedge s = n\}$
   $\{\textbf{inv } s = sum(k, n) \wedge 0 \leq k \leq n\}\{\textbf{bd } k\}$
   while $k \neq 0$ **do**
      … make k smaller and something else …
   od
   $\{s = sum(k, n) \wedge 0 \leq k \leq n \wedge k = 0\}$
   $\{s = sum(0, n)\}$

4) Then, let's consider the loop-body. Other than updating the variable k, we also need $\{p \wedge B\}\; S\; \{p\}$ being valid.
   k will be decreased by 1 after each iteration, and we need $s = sum(k, n)$ in the loop invariant. Thus, we can update $s \coloneqq s + sum(k - 1, n) - sum(k, n)$. Then:

   $\{n \geq 0\}\; k \coloneqq n;\; \{n \geq 0 \wedge k = n\}\; s \coloneqq n;$
   $\{n \geq 0 \wedge k = n \wedge s = n\}$
   $\{\textbf{inv } s = sum(k, n) \wedge 0 \leq k \leq n\}\{\textbf{bd } k\}$
   while $k \neq 0$ **do**
      $s \coloneqq s + k - 1;\; k \coloneqq k - 1$
   od
   $\{s = sum(k, n) \wedge 0 \leq k \leq n \wedge k = 0\}$
   $\{s = sum(0, n)\}$

   - The proof of the loop-body is omitted here.

- We can also try to create a loop invariant by adding some disjuncts or removing some conjuncts.

- Adding disjuncts can be very open-ended; but since we need $p \wedge \neg B \Rightarrow q$, we can try for different loop condition B and let $p \equiv q \vee B$, then we have both $p \wedge \neg B \Rightarrow q$ and $q \Rightarrow p$. The loop will look like:

  $\{\textbf{inv } q \vee B\}\{\textbf{bd} \ldots\}$
  $\textbf{while } B \textbf{ do}$
  $\quad \{(q \vee B) \wedge B\}$
  $\quad loop\ body$
  $\quad \{q \vee B\}$
  $\textbf{od}$
  $\{(q \vee B) \wedge \neg B\}$
  $\{q\}$

- Removing conjuncts can be used when the postcondition is a conjunction $q \equiv q_1 \wedge q_2 \wedge \ldots \wedge q_n$, where $n \geq 2$. It is natural to try to drop off some of q_k to get a loop invariant candidate: $p_k \equiv q_1 \wedge q_2 \wedge \ldots q_{k-1} \wedge q_{k+1} \wedge \ldots \wedge q_n$. Then the loop looks like:

  $\{\textbf{inv } p_k\}\{\textbf{bd} \ldots\}$
  $\textbf{while } \neg q_k \textbf{ do}$
  $\quad \{p_k \wedge \neg q_k\}$
  $\quad loop\ body$
  $\quad \{p_k\}$
  $\textbf{od}$
  $\{p_k \wedge q_k\}\{q\}$

2. Create a program which can linear-search for x in an array slice $b[0 \ldots n-1]$ (note that, in our language, we don't have the expression $b[0 \ldots n-1]$ to represent the first n indices of b). The precondition is that array b has at least n elements ($n \geq 0$); x may or may not appear in $b[0 \ldots n-1]$. The postcondition should be k equals to the index of the leftmost occurrence of x in $b[0 \ldots n-1]$; if x is not found then let $k = n$.

   1) Let us start with creating the pre- and post- conditions.

      Precondition can be: $size(b) \geq n \wedge n \geq 0$

      Let us define $x \notin b[0 \ldots n-1]$ with a predicate function $NotIn(x, b, n) \equiv \forall i. 0 \leq i < n \rightarrow x \neq b[i]$.

      We notice that no matter whether x is in $b[0 \ldots k-1]$ or not, we always have $NotIn(x, b, k)$ if k is in returned index. Thus, postcondition can be written as:

      $$0 \leq k \leq n \wedge NotIn(x, b, k) \wedge (k < n \rightarrow b[k] = x)$$
      $$\Leftrightarrow 0 \leq k \wedge k \leq n \wedge NotIn(x, b, k) \wedge (k < n \rightarrow b[k] = x)$$

   2) The postcondition is a conjunction, we can try to create a loop invariant by dropping some conjuncts. There are four conjuncts, and this means that we can try four candidates. Remember that, usually, the drop off conjunct will be considered as $\neg B$.

      a. If we drop off $0 \leq k$, then in the loop body we will have $k < 0$, and this is out of the bound of an array index. This is not a good idea.

      b. Similarly, if we drop of $k \leq n$, then in the loop body we will have $k > n$, which is not a guaranteed index in array b.

c. Dropping off $NotIn(x, b, k)$ means the loop condition becomes $x \in b[0 \ldots k-1]$ (note that this is not a legal expression). The problem is how do we start this loop?
   - If we start with $k = 0$, then we are check whether x is in an array slice of length 0, which is fine; but then we need to check with $k = 1$, and we need $x = b[0]$, but it is not guaranteed.
   - If we start with $k = n$, then we need $x \in b[0 \ldots n-1]$, which is also not guaranteed. So, this is not a good idea.

d. Dropping off $k < n \to b[k] = x$ could work. The loop condition will become $\neg(k < n \to b[k] = x) \Leftrightarrow k < n \wedge b[k] \neq x$. We can get a partial proof outline that looks like the following:

$\{size(b) \geq 0 \wedge n \geq 0\}$ … *some code* …
$\{\textbf{inv } 0 \leq k \leq n \wedge NotIn(x, b, k)\}$ # p
$\{\textbf{bd} \ldots\}$
while $k < n \wedge b[k] \neq x$ **do** # B
  $\{0 \leq k \leq n \wedge NotIn(x, b, k) \wedge k < n \wedge b[k] \neq x\}$ # $p \wedge B$
  loop body
  $\{0 \leq k \leq n \wedge NotIn(x, b, k)\}$ # p
od
$\{0 \leq k \leq n \wedge NotIn(x, b, k) \wedge (k < n \to b[k] = x)\}$ # $p \wedge \neg B \Leftrightarrow q$

3) We can start the loop with $k = 0$, so the precondition of the loop is $0 \leq k \leq n \wedge NotIn(x, b, k) \wedge k = 0$. In each iteration, we simply increase k.

$\{size(b) \geq 0 \wedge n \geq 0\}$ $k := 0;$
$\{size(b) \geq 0 \wedge n \geq 0 \wedge k = 0\}$ # forward assignment
$\{\textbf{inv } 0 \leq k \leq n \wedge NotIn(x, b, k)\}$
$\{\textbf{bd} \ldots\}$
while $k < n \wedge b[k] \neq x$ **do**
  $\{0 \leq k \leq n \wedge NotIn(x, b, k) \wedge k < n \wedge b[k] \neq x\}$
  $\{0 \leq k + 1 \leq n \wedge NotIn(x, b, k + 1)\}$ # backward assignment
  $k := k + 1$
  $\{0 \leq k \leq n \wedge NotIn(x, b, k)\}$
od
$\{0 \leq k \leq n \wedge NotIn(x, b, k) \wedge (k < n \to b[k] = x)\}$

4) $n - k$ is a good loop bound expression. Then we have a full proof outline of the program for linear search as follows:

$\{size(b) \geq 0 \wedge n \geq 0\}$ $k := 0;$
$\{size(b) \geq 0 \wedge n \geq 0 \wedge k = 0\}$
$\{\textbf{inv } 0 \leq k \leq n \wedge NotIn(x, b, k)\}$ $\{\textbf{bd } n - k\}$
while $k < n \wedge b[k] \neq n$ **do**
  $\{0 \leq k \leq n \wedge NotIn(x, b, k) \wedge k < n \wedge b[k] \neq n \wedge n - k = t_0\}$
  $\{0 \leq k + 1 \leq n \wedge NotIn(x, b, k + 1) \wedge n - (k + 1) < t_0\}$ # backward assignment
  $k := k + 1$
  $\{0 \leq k \leq n \wedge NotIn(x, b, k) \wedge n - k < t_0\}$
od
$\{0 \leq k \leq n \wedge NotIn(x, b, k) \wedge (k < n \to b[k] = n)\}$