CSC-1S003-EP Introduction to Algorithms

TD 3: Program Correctness

February 20th, 2025

Exercise 1: Hoare triples, in English

Recall that a Hoare triple $\{P\}$ `<program>` $\{Q\}$ means "if P is true, then when `<program>` terminates, Q will be true". Observe that the triple does *not* state that the execution of `<program>` will terminate, it is just making a claim under the hypothesis that it does.

For example, the meaning of the triple

$$\{\mathtt{x} = n\}\ \texttt{<program>}\ \{\mathtt{y} \geq 2 \land \forall z.(z \text{ divides } \mathtt{y}) \Rightarrow (z = 1 \lor z = \mathtt{y})\}$$

may be expressed in English by the sentence "if the variable `x` contains n, then when `<program>` terminates, the variable `y` will contain a prime number".

Express in English, with as few explicit mathematical formulas as you can, the meaning of the following Hoare triples:

1. $\{\mathtt{x} = n\}$ `<program>` $\{\mathtt{x} = n + 3\}$
2. $\{\mathtt{x} \leq \mathtt{y}\}$ `<program>` $\{\mathtt{y} \leq \mathtt{x}\}$
3. $\{\mathtt{True}\}$ `<program>` $\{\mathtt{x} = 5\}$
4. $\{\mathtt{True}\}$ `<program>` $\{\mathtt{False}\}$
5. $\{\mathtt{n} \geq 0\}$ `<program>` $\{\mathtt{r}^2 \leq \mathtt{n} \land (\mathtt{r} + 1)^2 > \mathtt{n}\}$

Exercise 2: valid and invalid Hoare triples

Which of the following Hoare triples is *valid*, that is, the claimed relation between precondition, program and postcondition always holds?

1. $\{\mathtt{True}\}$ `x = 4` $\{\mathtt{x} = 4\}$
2. $\{\mathtt{x} = 3\}$ `x = x + 1` $\{\mathtt{x} = 4\}$
3. $\{\mathtt{True}\}$
   ```
   x = 3
   y = 1
   ```
   $\{\mathtt{x} = 3\}$
4. $\{\mathtt{x} = 0 \land \mathtt{x} = 1\}$ `x = 5` $\{\mathtt{x} = 42\}$
5. $\{\mathtt{x} = 42\}$ `pass` $\{\mathtt{x} = 41\}$
6. $\{\mathtt{x} = 42\}$ `pass` $\{\mathtt{x} \leq 100\}$
7. $\{\mathtt{True}\}$
   ```
   while True:
       pass
   ```
   $\{\mathtt{False}\}$
8. $\{\mathtt{x} = 0\}$
   ```
   while x == 0:
       x = x + 1
   ```
   $\{\mathtt{x} = 1\}$
9. $\{\mathtt{x} = 1\}$
   ```
   while x != 0:
       x = x + 1
   ```
   $\{\mathtt{x} = 100\}$

10. $\{x = 1\}$
```
while x != 0:
    <something>
```
$\{x = 0\}$

Exercise 3: Preconditions and postconditions

Complete the following Hoare triples so that they are valid. Try to make the pre/post-conditions as precise as possible, in the following sense.

When you are asked to complete a triple of the form $\{???\}$ `<something>` $\{Q\}$, the question you need to ask yourself is: what is *the minimum amount of information* I need to know before executing `<something>` such that, when `<something>` terminates, I can be sure that Q is true? The closer you get to the minimum, the more points you obtain.

Similarly, when you are asked to complete a triple of the form $\{P\}$ `<something>` $\{???\}$, the question you need to ask yourself is: if I know that P holds before `<something>` is executed, and if `<something>` terminates, what is the *maximum amount of information* I may infer? The closer you get to the maximum, the more points you obtain.

In this perspective, `True` is the minimum amount of information possible: it tells us nothing at all, because it is always true! On the other hand, `False` is the maximum amount of information possible, so much information that it includes inconsistent data, *i.e.*, P and $\neg P$ for any statement P.

For the above reason, `False` is discouraged as a precondition: we are trying to minimize information, and `False` is usually so far from the minimum that it is likely to give you zero points (not always though: there are some cases in which `False` is the only possible precondition for making a triple valid). Otherwise stated, using `False` as a precondition is like a "wildcard": it always works, so the exercise would become trivial if we admitted it without warning.

Dually, `True` is discouraged as a postcondition: a triple of the form $\{P\}$ `<whatever>` $\{$`True`$\}$ is *always* valid, that is, `True` is the "postcondition wildcard". Since, in this case, we are trying to maximize the information, usually `True` will give you zero points (again, not always: in some cases, we really have no information at the end and `True` is the only thing we may infer).

1. $\{???\}$ `x = x - 1` $\{x < 0\}$

2. $\{???\}$
```
if x == 3:
    y = 4
else:
    y = 2
```
$\{\text{y is a perfect square}\}$

3. $\{???\}$
```
while i < 10:
    x = x + 1
    i = i + 1
```
$\{x = i\}$

4. $\{x = 3\}$ `x = x - y` $\{???\}$

5. $\{x = 1\}$
```
if x % 2 == 0:
    y = 0
else:
    y = 47
```
$\{???\}$

6. $\{x = 6\}$

```
if y < 0:
    y = -y
else:
    y = y + 1
x = x + y
```

$\{???\}$

Exercise 4: the assignment rule

Replace the question marks below with an assertion making the statement a valid instance of the assignment rule of Floyd-Hoare logic. If there is more than one possibility, list them all; if there is none, justify why.

1.
```
#! ???
x = 2 * x
#! x ≤ 10
```

2.
```
#! ???
x = 3
#! 0 ≤ x ∧ x ≤ 5
```

3.
```
#! y > 0 ∧ y = 7
x = y
#! ???
```

4.
```
#! x + y > 0 ∧ y = 7
x = x + y
#! ???
```

5.
```
#! 1 > 0 ∧ x = 7
x = 1
#! ???
```

Exercise 5: a wrong assignment rule

Let <expr> be an arbitrary arithmetic expression of mini-Python. Although it seems intuitively correct, the triple

$$\{\texttt{True}\}\ \texttt{x = <expr>}\ \{x = \texttt{<expr>}\}$$

is *not* valid in general. Can you find an example showing why this is the case? (*Hint: if* <expr> *is constant, the triple is valid, but if it's not constant...*).

Exercise 6: the while rule

Knowing that the triples

$$\{x \leq 100 \land x < 100\}\ \texttt{x = x + 1}\ \{x \leq 100\} \qquad \text{and} \qquad \{x > 100 \land x > 100\}\ \texttt{x = x + 1}\ \{x > 100\}$$

are both valid (if you do not see why, pause one moment and convince yourself that they are!), replace the question marks below with an assertion making the statement a correct instance of the while rule of Floyd-Hoare logic:

1.
```
#! x ≤ 100
while x < 100:
    #! x ≤ 100 ∧ x < 100
    x = x + 1
    #! x ≤ 100
#! ???
```

2.
```
#! x > 100
while x > 100:
    #! x > 100 ∧ x > 100
    x = x + 1
    #! x > 100
#! ???
```

Let `loop1` and `loop2` denote the first and second loop, respectively. Can you prove that the Hoare triples

$$\{\mathtt{x} \leq 100\}\ \texttt{loop1}\ \{\mathtt{x} = 100\} \qquad \text{and} \qquad \{\mathtt{x} > 100\}\ \texttt{loop2}\ \{\texttt{False}\}$$

are valid?

Exercise 7: The Truth Will Always Be

What does the following triple mean?

$$\{\texttt{True}\}\ \texttt{<program>}\ \{\texttt{True}\}$$

Show, by induction on the size of `<program>`, that it is always valid. That is, check that you may always write

```
#! True
<something>
#! True
```

no matter whether `<something>` is a `pass` instruction, an assignment `<var> = <expr>`, an `if`, or a `while` loop. For the latter two, use the induction hypothesis to propagate `#! True` through the blocks of code in the two branches of the `if` or in the body of the `while` loop.

Exercise 8: proving Exercise 2

For each triple of Exercise 2 which you claimed valid, give a proof of validity using the rules of Floyd-Hoare logic. That is, for each such triple

$$\{P\}\ \texttt{<program>}\ \{Q\},$$

start with the precondition #! P and show how it propagates through the instruction(s) of `<program>` via the rules of Floyd-Hoare logic (specifically, the assignment, while and consequence rules) until you obtain the postcondition #! Q.

Exercise 9: proving correctness of a simple program

Given a non-negative integer `n`, the following code obviously writes in `s` the number `2*n`. Prove that this is the case by completing the assertions in the code so that they obey the rules of Floyd-Hoare logic. (NB: a line of the form #! ??? may be replaced by more than one assertion, using the consequence rule).

```
#! n ≥ 0
s = 0
#! ???
i = 0
#! s = 2i ∧ i ≤ n
while i < n:
    #! ???
    s = s + 2
    #! ???
    i = i + 1
    #! ???
#! ???
#! s = 2n
```

Exercise 10: another simple program

Given a non-negative integer n, the following code obviously writes in s the sum of the first n numbers. Prove that this is the case by completing the assertions in the code so that they obey the rules of Floyd-Hoare logic, like the previous exercise.

```
#! n ≥ 0
#! 0 = Σ_{k=0}^{0−1} k  ∧  0 ≤ n+1
s = 0
#! ???
i = 0
#! s = Σ_{k=0}^{i−1} k  ∧  i ≤ n+1
while i <= n:
    #! ???
    s = s + i
    #! ???
    i = i + 1
    #! ???
#! ???
#! s = Σ_{k=0}^{n} k
```

Exercise 11: another square root algorithm

The following is another algorithm, coded in mini-Python, for computing integer square root, different from the one we saw in class:

```
def sqrt(n):
    r = 0
    while r * r < n:
        r = r + 1
    if r * r != n:
        r = r - 1
    else:
        pass
    return r
```

Prove that the above algorithm is correct by showing that the Hoare triple

$$\{0 \leq \mathtt{n}\}\ \mathtt{sqrt(n)}\ \{\mathtt{r}^2 \leq \mathtt{n} \wedge (\mathtt{r}+1)^2 > \mathtt{n}\}$$

is valid. That is, start with the assertion #! $0 \leq \texttt{n}$ just before the instruction `r = 0` and show how it may be propagated down, using the rules of Floyd-Hoare logic, until you arrive at the assertion #! $\texttt{r}^2 \leq \texttt{n} \wedge (\texttt{r}+1)^2 > \texttt{n}$ just before the instruction `return r`.

Hint: a possible loop invariant is $\mathrm{pred}(\texttt{r})^2 \leq \texttt{n}$, *where* pred *is* truncated subtraction, *defined by* $\mathrm{pred}(x) = x - 1$ *for all* $x > 0$, *and* $\mathrm{pred}(x) = 0$ *for all* $x \leq 0$.

Exercise 12: Nico was correct

Here is an implementation of Lomuto's partitioning scheme in mini-Python (with one-line swap instructions):

```
def partition(l,b,e):
    pivot = l[e]
    p = b
    i = b
    while i < e:
        if l[i] < pivot:
            l[i],l[p] = l[p],l[i]
            p = p + 1
        else:
            pass
        i = i + 1
    l[e],l[p] = l[p],l[e]
    return p
```

Prove the correctness of Lomuto's partitioning scheme by showing that the Hoare triple

$$\{\texttt{b} \leq \texttt{e}\}\ \texttt{partition(l,b,e)}\ \{(\forall j.\, \texttt{b} \leq j < \texttt{p} \Rightarrow \texttt{l}[j] \leq \texttt{l}[\texttt{p}]) \wedge (\forall j.\, \texttt{p} < j \leq \texttt{e} \Rightarrow \texttt{l}[j] \geq \texttt{l}[\texttt{p}])\}$$

is valid, in the sense that, if the precondition holds before executing the first instruction of the `partition` function, then the postcondition will hold just before the `return p` statement.

For the proof, you may ignore "index out of bounds" errors, that is, you may assume that whenever an expression of the form $\texttt{l}[v]$ is evaluated, the value of v always falls within the length of `l` (which is in fact always the case as long as `b` and `e` are valid indices of `l`, but we will not bother proving it).

For the one-line swap instruction, you may use the rule

```
#! P[a,b]
a,b = b,a
#! P[b,a]
```

which allows to exchange the role of the swapped expressions in an arbitrary statement P containing them (for example, if $P[\texttt{a},\texttt{b}]$ is $\texttt{a} \leq \texttt{b}$, then $P[\texttt{b},\texttt{a}]$ is $\texttt{b} \leq \texttt{a}$).

Hint: the loop inviariant is the one discussed on the slides of lecture 1, namely

$$(\forall j.\, \texttt{b} \leq j < \texttt{p} \Rightarrow \texttt{l}[j] < \texttt{pivot}) \wedge (\forall j.\, \texttt{p} \leq j < \texttt{i} \Rightarrow \texttt{l}[j] \geq \texttt{pivot}) \wedge \texttt{l}[\texttt{e}] = \texttt{pivot} \wedge \texttt{i} \leq \texttt{e}.$$