Reasoning About Programs

Week 8 Tutorial - String Concatenation

Sophia Drossopoulou and Mark Wheelhouse

This exercise sheet has been adapted from a past exam question. Parts (a), (c), (e), (f) and (h) were in the 2013 Logic and Reasoning About Programs exam. A good revision exercise would be to try and prove the remaining proof obligations for partial correctness of the `concat` program that are not covered on this exercise sheet (e.g. that the initialisation code in each function establishes the invariant and that the invariants are maintained by their respective loops).

1st Question: String Concatenation

Consider the following Java function `concat` which constructs the concatenation of two input strings, denoted by `a++b`:

```
1   char[] concat( char[] a, char[] b )
2   //PRE: a ≠ null  ∧  b ≠ null
3   //POST: a ≈ a0  ∧  b ≈ b0  ∧  r ≈ a++b
4   {
5       int x = a.length + b.length;
6       // MID: M1
7       char[] c = new char[x];
8       // MID: M2
9       copy( c, a, 0 );
10      // MID: M3
11      copy( c, b, a.length );
12      // MID: M4
13      return c;
14  }
```

This function makes use of an auxiliary function `copy`, implemented as follows:

```
1   void copy( char[] a, char[] b, int x )
2   // PRE: a ≠ null  ∧  b ≠ null  ∧  0 ≤ x ≤ a.length − b.length
3   // POST: b ≈ b0  ∧  a ≈ a0[0..x)++b[0..b.length)++a0[x + b.length..a0.length)
4   {
5       int i = 0;
6       // INV: I
7       // VAR: V
8       while( i < b.length ) {
9           a[x+i] = b[i];
10          i++;
11      }
12  }
```

Consider also copyN, an alternative implementation of the copy function:

```
1  void copyN( char[] a, char[] b, int x ) {
2      int i = 0;
3      // INV: I_N
4      // VAR: V_N
5      while( i < b.length - (N-1) ) {
6          copyTuple( N, a, b, x+i, i );
7          i = i + N;
8      }
9      copyTuple( b.length-i, a, b, x+i, i );
10 }
```

The copyN function makes use of copyTuple which is a (fictional) library function that allows for N adjacent array cells to be copied in parallel (N is a variable set by the system). The implementation of copyTuple is not known, but the function has been shown to satisfy the following specification:

```
1  copyTuple( int n, char[] a, char[] b, int x, int y )
2  // PRE:  a ≠ null  ∧  b ≠ null  ∧  0 ≤ n ≤ N
3  //        ∧  0 ≤ x ≤ a.length − n  ∧  0 ≤ y ≤ b.length − n
4  // POST: b ≈ b_0  ∧  a ≈ a_0[0..x)++b[y..y+n)++a_0[x+n..a_0.length)
5  {
6      ...
7  }
```

It is claimed that the copyN function has the same overall behaviour as the copy function, and thus we can replace copy with copyN in the concat function to obtain a more efficient implementation of string concatenation.

(a) Write an invariant I for the loop in the body of the original copy function.
(You do not need to prove anything.)

A possible answer:

$$I \quad \longleftrightarrow \quad \texttt{b} \approx \texttt{b}_0 \ \wedge\ 0 \le \texttt{i} \le \texttt{b.length} \wedge\ \texttt{a} \approx \texttt{a}_0[0\,..\,\texttt{x})\texttt{++b}[0\,..\,\texttt{i})\texttt{++a}_0[\texttt{x}+\texttt{i}\,..\,\texttt{a}_0\texttt{.length})$$

Notes:

$\texttt{b} \approx \texttt{b}_0$ is required by the postcondition of copy so must be included in the invariant.

$0 \le \texttt{i} \le \texttt{b.length}$ tracks the value of the loop counter.

$\texttt{a}_0[0\,..\,\texttt{x})$ is the unmodified initial section of the array a (everything before offset x).

$\texttt{b}[0\,..\,\texttt{i})$ is the updated part of the array a. Before the loop $\texttt{i} = 0$ so the range is empty, and after the loop $\texttt{i} = \texttt{b.length}$ which matches the requirement of the postcondition of copy.

$\texttt{a}_0[\texttt{x}+\texttt{i}\,..\,\texttt{a}_0\texttt{.length})$ is the remaining part of the array a not yet (or possibly ever) modified. As above, before the loop $\texttt{i} = 0$ so none of the array a has been modified yet, and after the loop $\texttt{i} = \texttt{b.length}$ which again matches the requirement of the postcondition of copy.

(b) Write a variant V for the loop in the body of the original `copy` function.
(You do not need to prove anything.)

A possible answer:

$$V \quad = \quad \texttt{b.length} - \texttt{i}$$

(c) Prove that the postcondition of the original `copy` function holds immediately after the termination of the loop in its body. State clearly what is given and what you need to show.
(You do not need to show that the invariant is established before the loop or re-established by the loop.)

A possible answer:

Given:

(1)	$\texttt{b} \approx \texttt{b}_0$	from `INV`
(2)	$0 \leq \texttt{i} \leq \texttt{b.length}$	from `INV`
(3)	$\texttt{a} \approx \texttt{a}_0[0\,..\,\texttt{x})\texttt{++b}[0\,..\,\texttt{i})\texttt{++a}_0[\texttt{x} + \texttt{i}\,..\,\texttt{a}_0\texttt{.length}]$	from `INV`
(4)	$\texttt{i} \geq \texttt{b.length}$	from ¬loop condition

To show:

(α)	$\texttt{b} \approx \texttt{b}_0$
(β)	$\texttt{a} \approx \texttt{a}_0[0\,..\,\texttt{x})\texttt{++b}[0\,..\,\texttt{b.length})\texttt{++a}_0[\texttt{x} + \texttt{b.length}\,..\,\texttt{a}_0\texttt{.length}]$

Proof:

α	follows directly from (1)	
(5)	$\texttt{i} = \texttt{b.length}$	from (2) and (4)
β	follows from (3) and (5)	

(d) Prove that the variant V decreases with every iteration and has a lower bound.

A possible answer:

Given:

(1)	$\texttt{b} \approx \texttt{b}_0$	from INV
(2)	$0 \leq \texttt{i} \leq \texttt{b.length}$	from INV
(3)	$\texttt{a} \approx \texttt{a}_0[0\,..\,\texttt{x})\texttt{++b}[0\,..\,\texttt{i})\texttt{++a}_0[\texttt{x}+\texttt{i}\,..\,\texttt{a}_0\texttt{.length})$	from INV
(4)	$\texttt{i} < \texttt{b.length}$	from loop condition
(5)	$\texttt{i}' = \texttt{i} + 1$	from code line 10
(6)	$\texttt{b}' \approx \texttt{b}$	implicit from code

To show:

(α)	$\texttt{b.length} - \texttt{i} \geq 0$
(β)	$\texttt{b}'\texttt{.length} - \texttt{i}' < \texttt{b.length} - \texttt{i}$

Proof:

(7)	$\texttt{i} \leq \texttt{b.length}$	from (2)
(8)	$0 \leq \texttt{b.length} - \texttt{i}$	from (7) ($-\texttt{i}$ from each side)
α	follows from rearranging (8)	
(9)	$\texttt{i}' > \texttt{i}$	from (5)
(10)	$-\texttt{i}' < -\texttt{i}$	from (9) (mult each side by -1)
(11)	$\texttt{b.length} - \texttt{i}' < \texttt{b.length} - \texttt{i}$	from (6) ($+\texttt{b.length}$ to each side)
β	follows from (11) and (6)	

(e) Write mid-conditions M_1, M_2, M_3 and M_4 that hold at their respective points in the body of the `concat` function. You must ensure that your mid-conditions are strong enough to be used in a proof of partial correctness of the code, but you do not need to prove this partial correctness.

Note: You can assume that the `char[] c = new char[x]` command creates a new array of length `x`, with all of its contents initially set to `'null'` (the `null` character `'\u0000'`) and returns its address to the variable `c`.

A possible answer:

$$M_1 \longleftrightarrow \texttt{a} \approx \texttt{a}_0 \;\wedge\; \texttt{b} \approx \texttt{b}_0 \;\wedge\; \texttt{x} = \texttt{a.length} + \texttt{b.length}$$

$$M_2 \longleftrightarrow \texttt{a} \approx \texttt{a}_0 \;\wedge\; \texttt{b} \approx \texttt{b}_0 \;\wedge\; \texttt{c.length} = \texttt{a.length} + \texttt{b.length} \;\wedge\; \texttt{c}[0\,..\,\texttt{c.length}) = \texttt{'null'}$$

$$M_3 \longleftrightarrow \texttt{a} \approx \texttt{a}_0 \;\wedge\; \texttt{b} \approx \texttt{b}_0 \;\wedge\; \texttt{c.length} = \texttt{a.length} + \texttt{b.length} \;\wedge\; \texttt{c}[0\,..\,\texttt{a.length}) \approx \texttt{a}[0\,..\,\texttt{a.length}) \;\wedge\; \texttt{c}[\texttt{a.length}\,..\,\texttt{c.length}) = \texttt{'null'}$$

$$M_4 \longleftrightarrow \texttt{a} \approx \texttt{a}_0 \;\wedge\; \texttt{b} \approx \texttt{b}_0 \;\wedge\; \texttt{c} \approx \texttt{a++b}$$

Note: The properties describing the `'null'` parts of the array `c` are not necessary to prove partial correctness, but have been added above for completeness sake.

To satisfy the preconditions of the calls to `copy` you might expect to see $\texttt{c} \neq \texttt{null}$ in the midconditions M_2 and M_3. Adding this property would not be incorrect, but we can actually derive it from the midconditions and precondition of `concat`. In particular:

Given:

(1)	$\texttt{a}_0 \neq \texttt{null}$	from PRE
(2)	$\texttt{b}_0 \neq \texttt{null}$	from PRE
(3)	$\texttt{a} \approx \texttt{a}_0$	from MID
(4)	$\texttt{b} \approx \texttt{b}_0$	from MID
(5)	$\texttt{c.length} = \texttt{a.length} + \texttt{b.length}$	from MID

To show:

$$(\alpha) \quad \texttt{c} \neq \texttt{null}$$

Proof:

(6)	$\texttt{a} \neq \texttt{null}$	from (1) and (3)
(7)	$\texttt{b} \neq \texttt{null}$	from (2) and (4)
(8)	`a.length` well defined	from (6)
(9)	`b.length` well defined	from (7)
(10)	`c.length` well defined	from (8), (9) and (5)
α	follows from (10)	

There are a few common pitfalls to be wary of when writing these (or any) midconditions:

- It is essential that we track that `a` and `b` are not modified by the code at each midcondition, or else we won't be able to establish this in the postcondition.
- In M_2 it can seem tempting to write something along the lines of $\texttt{c.length} = \texttt{x}$. However, if you don't also state that $\texttt{x} = \texttt{a.length} + \texttt{b.length}$ in the *same* midcondition, then we don't actually know what the length of `c` is and we would be unable to establish that the precondition of `copy` holds on line 9.
- Following on from the previous point, we cannot try to capture the previous value of `x` by writing $\texttt{c.length} = \texttt{x}_0$. Remember that $\texttt{x}_0$ would refer to the value of `x` as passed into the `concat` method and there was no such `x`. We deliberately have no notational method for referring to values at arbitrary points in the code, as this would lead to a lot of potential confusion.
- Similarly, there should not be any reference to $\texttt{c}_0$ in any midcondition, as no such `c` was passed into the `concat` method. In particular, this means that we *cannot* write $\texttt{c} \approx \texttt{a++c}_0\texttt{[a.length..c.length]}$ in M_3.
- There should not be any reference to $\texttt{c}'$ in any midcondition. It would be *incorrect* to write something like $\texttt{c}'\texttt{[a.length .. c.length)} \approx \texttt{c[a.length .. c.length)}$ in M_3. Remember that each midcondition describes a snap-shot of the program state at that point in time. There is no variable $\texttt{c}'$ in our code, so it makes no sense to reason about it (in fact, I would recommend that you avoid using primed variable names at all for the duration of this course). The only place that we prime our variables is in the "To Show" sections of proofs where the code has been modified.

(f) Write an invariant I_N for the loop in the body of the `copyN` function.
(You do not need to prove anything.)

Note: You can assume that the `copyN` function has the same specification as the original `copy` function.

A possible answer:

$$\begin{aligned} I_N \quad \longleftrightarrow \quad & \texttt{b} \approx \texttt{b}_0 \ \wedge \ 0 \leq \texttt{i} \leq \texttt{b.length} \\ & \wedge \ 0 \leq \texttt{x} \leq \texttt{a}_0\texttt{.length} - \texttt{b.length} \\ & \wedge \ \texttt{a} \approx \texttt{a}_0[0\,..\,\texttt{x})\texttt{++b}[0\,..\,\texttt{i})\texttt{++a}_0[\texttt{x} + \texttt{i}\,..\,\texttt{a}_0\texttt{.length}) \end{aligned}$$

Note: The above invariant is much the same as that for `copy` with the addition that we choose to track the range of `x` as well. It might seem surprising that the upper bound on `i` does not depend on `N`, but this is due to how the loop modifies `i`. Consider the following:

$\texttt{i} < \texttt{b.length} - (\texttt{N} - 1)$	from `copyN cond`
$\texttt{i} < \texttt{b.length} + 1 - \texttt{N}$	by rearranging
$\texttt{i} \leq \texttt{b.length} - \texttt{N}$	shifting $<$ to $\leq$
$\texttt{i} \leq \texttt{b.length}$	since $\texttt{i}' = \texttt{i} + \texttt{N}$ in loop body

(g) Write a variant V_N for the loop in the body of the `copyN` function.
(You do not need to prove anything.)

A possible answer:

$$V_N \quad = \quad \texttt{b.length} - \texttt{i}$$

or

$$V_N \quad = \quad \texttt{b.length} - (\texttt{N} - 1) - \texttt{i}$$

Note that both of these decrease on each loop iteration and have a lower bound of 0. It does not matter that in the first case the program may terminate before the variant actually reaches this bound. It is enough that there is a finite limit on the number of iterations of the loop.

(h) Prove that the finalisation code in the `copyN` function establishes its postcondition. State clearly what is given and what you need to show.
(You do not need to show that the invariant is established before the loop or re-established by the loop.)

A possible answer:

Given:

$(P1)$	$\mathtt{a}_0 \neq \mathtt{null}$	from PRE
$(P2)$	$\mathtt{b}_0 \neq \mathtt{null}$	from PRE
(1)	$\mathtt{b} \approx \mathtt{b}_0$	from INV
(2)	$0 \leq \mathtt{i} \leq \mathtt{b.length}$	from INV
(3)	$0 \leq \mathtt{x} \leq \mathtt{a}_0\mathtt{.length} - \mathtt{b.length}$	from INV
(4)	$\mathtt{a} \approx \mathtt{a}_0[0\,..\,\mathtt{x})\mathtt{++b}[0\,..\,\mathtt{i})\mathtt{++a}_0[\mathtt{x}+\mathtt{i}\,..\,\mathtt{a}_0\mathtt{.length}]$	from INV
(5)	$\mathtt{i} \geq \mathtt{b.length} - (\mathtt{N}-1)$	from ¬loop condition
$(C6)$	$\mathtt{b}' \approx \mathtt{b}$	from code line 9
$(C7)$	$\mathtt{a}' \approx \mathtt{a}[0\,..\,\mathtt{x}+\mathtt{i})\mathtt{++b}[\mathtt{i}\,..\,\mathtt{i}+(\mathtt{b.length}-\mathtt{i}))$ $\mathtt{++a}[\mathtt{x}+\mathtt{i}+(\mathtt{b.length}-\mathtt{i})\,..\,\mathtt{a.length})$	from code line 9
(8)	$\mathtt{x}' = \mathtt{x}$	implicit from code

To show:

(α)	$\mathtt{b}' \approx \mathtt{b}_0$
(β)	$\mathtt{a}' \approx \mathtt{a}_0[0\,..\,\mathtt{x}')\mathtt{++b}'[0\,..\,\mathtt{b}'\mathtt{.length})\mathtt{++a}_0[\mathtt{x}'+\mathtt{b}'\mathtt{.length}\,..\,\mathtt{a}_0\mathtt{.length})$

Proof:

(9)	$\mathtt{b.length} - (\mathtt{N}-1) \leq \mathtt{i} \leq \mathtt{b.length}$	from (2) and (5)
(10)	$0 \leq \mathtt{b.length} - \mathtt{i} \leq \mathtt{N}-1$	from (9)
(11)	$0 \leq \mathtt{x}+\mathtt{i} \leq \mathtt{a.length} - (\mathtt{b.length}-\mathtt{i})$	from (3)
(12)	$0 \leq \mathtt{i} \leq \mathtt{b.length} - (\mathtt{b.length}-\mathtt{i})$	from (2)
	$(P1)$, $(P2)$, (1), (4), (10) (11) and (12) satisfy `copyTuple` PRE	
	This means that we can use $(C6)$ and $(C7)$ in the rest of our proof	
α	follows from (1) and $(C6)$	
(13)	$\mathtt{a}' \approx \mathtt{a}_0[0\,..\,\mathtt{x})\mathtt{++b}[0\,..\,\mathtt{i})\mathtt{++b}[\mathtt{i}\,..\,\mathtt{b.length})$ $\mathtt{++a}_0[\mathtt{x}+\mathtt{b.length}\,..\,\mathtt{a}_0\mathtt{.length})$	from $(C7)$ and (4)
(14)	$\mathtt{a}' \approx \mathtt{a}_0[0\,..\,\mathtt{x})\mathtt{++b}[0\,..\,\mathtt{b.length})\mathtt{++a}_0[\mathtt{x}+\mathtt{b.length}\,..\,\mathtt{a}_0\mathtt{.length})$	from (13)
β	follows from (14), $(C6)$ and (8)	

Note that it is essential that we verify the pre-condition of `copyTuple` to be able to use the information about the modification to `a` from its post-condition in the proof.

(i) Prove that the variant V_N decreases with every iteration and has a lower bound.

A possible answer:

We work with the first variant from part (g).

Given:

$(P1)$	$\texttt{a}_0 \neq \texttt{null}$	from PRE
$(P2)$	$\texttt{b}_0 \neq \texttt{null}$	from PRE
(1)	$\texttt{b} \approx \texttt{b}_0$	from INV
(2)	$0 \leq \texttt{i} \leq \texttt{b.length}$	from INV
(3)	$0 \leq \texttt{x} \leq \texttt{a}_0\texttt{.length} - \texttt{b.length}$	from INV
(4)	$\texttt{a} \approx \texttt{a}_0[0\,..\,\texttt{x}) : \texttt{b}[0\,..\,\texttt{i}) : \texttt{a}_0[\texttt{x}+\texttt{i}\,..\,\texttt{a}_0\texttt{.length})$	from INV
(5)	$\texttt{i} < \texttt{b.length} - (\texttt{N}-1)$	from loop condition
$(C6)$	$\texttt{b}' \approx \texttt{b}$	from code line 6
$(C7)$	$\texttt{a}' \approx \texttt{a}[0\,..\,\texttt{x}+\texttt{i})\texttt{++b}[\texttt{i}\,..\,\texttt{i}+\texttt{N})$ $\texttt{++a}[\texttt{x}+\texttt{i}+\texttt{N}\,..\,\texttt{a}_0\texttt{.length})$	from code line 6
(8)	$\texttt{i}' = \texttt{i} + \texttt{N}$	from code line 7
(9)	$\texttt{N}' = \texttt{N}$	implicit from code
(10)	$\texttt{x}' = \texttt{x}$	implicit from code

To show:

(α) $\texttt{b.length} - \texttt{i} \geq 0$

(β) $\texttt{b}'\texttt{.length} - \texttt{i}' < \texttt{b.length} - \texttt{i}$

Proof:

α	follows directly from (2)	
(11)	$0 \leq \texttt{i} \leq \texttt{b.length} - (\texttt{N})$	from (2) and (5)
(12)	$0 \leq \texttt{x} + \texttt{i} \leq \texttt{a}_0\texttt{.length} - (\texttt{b.length} - \texttt{i})$	from (3)
(13)	$0 \leq \texttt{x} + \texttt{i} \leq \texttt{a}_0\texttt{.length} - \texttt{N}$	from (11) and (12)
	$(P1)$, $(P2)$, (1), (4), (11) and (13) satisfy copyTuple PRE	
	This means that we can use $(C6)$ and $(C7)$ in our proof	
(14)	$\texttt{b}'\texttt{.length} = \texttt{b.length}$	from $(C6)$
(15)	$\texttt{i}' > \texttt{i}$	from $(C8)$
β	follows from (14) and (15)	

Note that it is essential that we verify the pre-condition of copyTuple to know that the array b is unmodified by the code. If the pre-condition of copyTuple were not satisfied then the method call could completely change the program state (but especially the array b) in any way that it wanted to.