COMP4161 T3/2020
Advanced Topics in Software Verification

Assignment 3

This assignment starts on Wednesday, 2020-11-11 and is due on Sunday, 2020-11-22, 23:59h. We will accept Isabelle .thy files only. In addition to this pdf document, please refer to the provided Isabelle template for the definitions and lemma statements.

The assignment is take-home. This does NOT mean you can work in groups. Each submission is personal. For more information, see the plagiarism policy: https://student.unsw.edu.au/plagiarism

Submit using `give` on a CSE machine: `give cs4161 a3 a3.thy`

For all questions, you may prove your own helper lemmas, and you may use lemmas proved earlier in other questions. You can also use automated tool like sledghammer. If you can't finish an earlier proof, use *sorry* to assume that the result holds so that you can use it if you wish in a later proof. You won't be penalised in the later proof for using an earlier true result you were unable to prove, and you'll be awarded part marks for the earlier question in accordance with the progress you made on it.

1 C verification: doubly-linked list (55 marks)

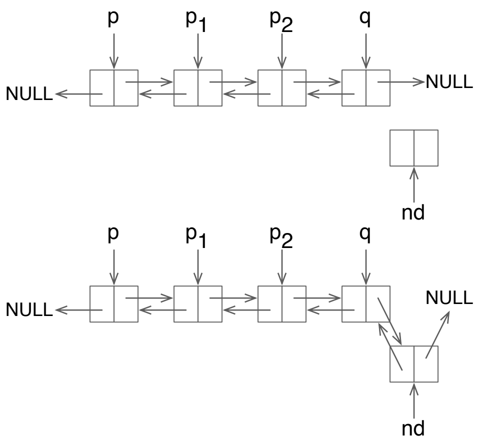
In this question, we look at the insertion operation for a doubly-linked list of nodes defined as follows:

```
struct node {
  struct node *prev;
  struct node *next;
};
```

Here, each node consists simply of prev and next links (no data stored in nodes). Now, we define our insertion function *insert-after* as follows:

```
/* insert node 'nd' after node 'ptr' */
void insert_after(struct node* nd, struct node* ptr) {
  nd->prev = ptr;
  nd->next = ptr->next;
  if (ptr->next != 0) {
    ptr->next->prev = nd;
  }
  ptr->next = nd;
}
```

The function *insert-after nd q* takes a new node *nd* and a pointer *q* to a node in a doubly-linked list. This function updates the linked list by inserting the new node *nd* after the node that *q* is pointing to. In this question, we only consider the case where *q* points to the end of the doubly-linked list. The diagram below shows how the links are changed by *insert-after nd q* in this case.



Our goal is to prove the correctness of *insert-after* for this case, which we state as a Hoare triple:

$$\{\!|\textit{insert-pre}\ p\ \textit{xs}\ q\ \textit{nd}|\!\}\ \textit{insert-after}'\ \textit{nd}\ q\ \{\!|\lambda\text{-}.\ \textit{insert-post}\ p\ \textit{xs}\ \textit{nd}|\!\}\quad (1)$$

for suitable precondition *insert-pre* and postcondition *insert-post*. And, here, *insert-after′* is the result that the *autocorres* tool produces by translating the C parser output of the function *insert-after* to make it nicer to reason about.

Before proving this statement, we first introduce several definitions that we need for the precondition and the postcondition.

Precondition. The precondition states that there exists a valid, non-empty doubly-linked list from a pointer *p* to the pointer *q*. It will also state that the pointer *nd* to the new node to be inserted is not NULL, points to a valid node, and does not point to a node already in the doubly-linked list:

$\textit{insert-pre}\ p\ \textit{xs}\ q\ \textit{nd}\ s =$
$(\textit{xs} \neq [] \wedge$
$\ \textit{is-dlist}\ (\textit{is-valid-node-C}\ s)\ (\textit{heap-node-C}\ s)\ p\ \textit{xs}\ q \wedge$
$\ \textit{nd} \neq \textit{NULL} \wedge \textit{is-valid-node-C}\ s\ \textit{nd} \wedge \textit{nd} \notin \textit{set}\ \textit{xs})$

Here, the functions *heap-node-C* and *is-valid-node-C* are given by AutoCorres and provides the heap content and pointer validity respectively. The notion of a valid doubly-linked list is defined with *is-dlist vld hp p xs q* stating that there is a doubly-linked list of *valid* (according to *vld*) nodes starting from *p* and finishing in *q*, and where *xs* is the list of all the pointers in this doubly-linked list. The function *is-dlist* is defined in terms of *path*: there is a path from p to the NULL pointer if we follow the *next* field and from q to NULL if we follow the *prev* field:

$\mathit{is\text{-}dlist}\ \mathit{vld}\ \mathit{hp}\ p\ \mathit{xs}\ q \equiv$
$\mathit{path}\ \mathit{vld}\ \mathit{hp}\ \mathit{next\text{-}C}\ p\ \mathit{xs}\ \mathit{NULL} \wedge \mathit{path}\ \mathit{vld}\ \mathit{hp}\ \mathit{prev\text{-}C}\ q\ (\mathit{rev}\ \mathit{xs})\ \mathit{NULL}$

$\mathit{path}\ \mathit{vld}\ \mathit{hp}\ \mathit{nxt}\ p\ []\ q = (p = q)$
$\mathit{path}\ \mathit{vld}\ \mathit{hp}\ \mathit{nxt}\ p\ (x \# \mathit{xs})\ q =$
$(p \neq q \wedge \mathit{vld}\ p \wedge p \neq \mathit{NULL} \wedge p = x \wedge \mathit{path}\ \mathit{vld}\ \mathit{hp}\ \mathit{nxt}\ (\mathit{nxt}\ (\mathit{hp}\ p))\ \mathit{xs}\ q)$

Postcondition. The postcondition states that there is still a valid doubly-linked list from a pointer p that now goes up to pointer nd and contains the initial list of pointers plus nd at its end:
$\mathit{insert\text{-}post}\ p\ \mathit{xs}\ \mathit{nd}\ s =$
$\mathit{is\text{-}dlist}\ (\mathit{is\text{-}valid\text{-}node\text{-}C}\ s)\ (\mathit{heap\text{-}node\text{-}C}\ s)\ p\ (\mathit{xs}\ @\ [\mathit{nd}])\ \mathit{nd}$

Proof. Here are a series of questions to guide you towards proving the correctness of the *insert-after* C function. For many of these, you will need to use induction, often with generalisation of some of the variables. Pay attention to which variable to use *arbitrary* on.

Note that if you manage to prove the correctness lemma (1) using your own helper lemmas (with none of them "sorried"), you will get full marks for this question. (Note that partial marks for progress towards solution will only be awarded if you follow the lemmas below).

(a) Start to prove (1) by unfolding definitions and applying wp. This leads to a large subgoal, with a lot of function updates (terms of the form $(f(x := y))\ z$). (6 marks)

(b) The goal contains terms of the form $\mathit{path}\ \mathit{vld}\ (\mathit{hp}(x := y))\ n\ p\ \mathit{xs}\ q$. Prove:

$$x \notin \mathit{set}\ \mathit{xs} \Longrightarrow \mathit{path}\ \mathit{vld}\ (\mathit{hp}(x := y))\ n\ p\ \mathit{xs}\ q = \mathit{path}\ \mathit{vld}\ \mathit{hp}\ n\ p\ \mathit{xs}\ q$$

. (6 marks)

(c) The goal also contains terms of the form $\mathit{path}\ \mathit{vld}\ \mathit{hp}\ n\ p\ (\mathit{xs}\ @\ \mathit{ys})\ q$. To simplify these, we prove a series of helper lemmas:

(c1) Prove that the start of a doubly-linked list is in the set on pointers:

$[\![\mathit{path}\ \mathit{vld}\ \mathit{hp}\ n\ p\ \mathit{xs}\ q;\ \mathit{xs} \neq []]\!] \Longrightarrow p \in \mathit{set}\ \mathit{xs}$

(5 marks)

(c2) Prove that a path is unique:

$\llbracket path\ vld\ hp\ n\ p\ xs\ q;\ path\ vld\ hp\ n\ p\ ys\ q\rrbracket \Longrightarrow xs = ys$

(7 marks)

(c3) Prove a destruction rule for a path of an append list:

$path\ vld\ hp\ n\ p\ (xs\ @\ ys)\ q \Longrightarrow$
$\exists\, r.\ path\ vld\ hp\ n\ p\ xs\ r \wedge path\ vld\ hp\ n\ r\ ys\ q$

(8 marks)

(c4) Using the lemma *in-set-conv-decomp*:$(x \in set\ xs) = (\exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs)$ from Isabelle, prove:

$path\ vld\ hp\ n\ (n\ (hp\ p))\ xs\ q \Longrightarrow p \notin set\ xs$

(7 marks)

(c5) Finally prove the value of a path of an append:

$path\ vld\ hp\ n\ p\ (xs\ @\ ys)\ q =$
$(path\ vld\ hp\ n\ p\ xs\ (if\ ys = []\ then\ q\ else\ hd\ ys) \wedge$
$path\ vld\ hp\ n\ (if\ ys = []\ then\ q\ else\ hd\ ys)\ ys\ q \wedge$
$set\ xs \cap set\ ys = \{\} \wedge q \notin set\ xs)$

(8 marks)

(d) Using the lemmas *path-append-last* and *path-upd* finish the proof of (1). **Hint**: try case distinction on "rev xs". (8 marks)

2 C verification: primality test (45 marks)

The function *is-prime-2* below tests whether a given number n is a prime or not.

```
unsigned int is_prime_2(unsigned int n)
{
  /* Numbers less than 2 are not prime. */
  if (n < 2) {
    return 0;
  }

  /* 2 is the only even number that is prime */
  if (n % 2 == 0) {
    return (n == 2);
  }

  /* Find the first non-trivial factor of 'n'. */
  unsigned int i = 3;
  while (n % i != 0) {
    i+=2;
  }

  /* If the first factor found is 'n', it is a prime */
  return (i == n);
}
```

We will verify that *is-prime-2* correctly computes the primality of *n*. In other words, we will show that *is-prime-2* returns *1* if and only if the input *n* is a prime and that the computation always terminates.

(a) Define a predicate *is-prime-inv* which states the invariant of the while loop in the *is-prime-2* function. (6 marks)

(b) Define a measure *is-prime-measure* which returns a natural number that strictly decreases each loop iteration. (5 marks)

(c) Using AutoCorres ("*apply wp*"), show *is-prime-2* is correct. To prove this, first introduce the following lemmas:

  (c1) *prime-2-or-odd* which states that a prime number *n* is either equal to *2* or an odd number. (4 marks)

  (c2) *is-prime-precond-implies-inv* which states that the invariant holds when you first enter the loop; (4 marks)

  (c3) *is-prime-body-obeys-inv* which states that the invariant holds between loop iterations; (6 marks)

  (c4) *is-prime-body-obeys-measure* which states that the measure decreases between loop iterations; (6 marks)

  (c5) *is-prime-body-implies-no-overflow* which states that the loop invariant implies there is no overflow (5 marks)

  (c6) *is-prime-inv-implies-postcondition* which states that the invariant implies the function's post-condition when the loop finally finishes. (5 marks)

(d) Finally, use these lemmas to complete the proof. (4 marks)

Hints

- You might find lemmas such as *prime-nat-iff* and *prime-nat-naiveI* useful for converting *prime* into something easier to handle.
- While C uses 32-bit words (of Isabelle type `word32`), AutoCorres uses a technique called *word abstraction*, allowing you to reason using `nat`'s instead. This comes at the price of being obliged to prove that arithmetic operations don't overflow `UINT_MAX` (i.e., $2^{32} - 1$).

- The tactic "`apply arith`" may help to solve 'obvious' proofs involving arithmetic. Additionally, you may find *sledgehammer* to also be effective at proofs that seem obvious.

- Feel free to modify any of the helper definitions or lemmas in the template. The goal is to prove the function is correct: if changing some of the provided lemmas helps you achieve this, don't be afraid to do it.