COMP4161 T3/2022
Advanced Topics in Software Verification

Assignment 3

This assignment star[illegible]22 and is due on Fri, 18 Nov, 20:00h. We will accept Isabelle .thy files on[illegible]s PDF document, please refer to the provided Isabelle template for the defi[illegible]atements.

The assignment is ta[illegible]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 tools like sledgehammer. 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 partial marks for the earlier question in accordance with the progress you made on it.

1 General recursive function (22 marks)

In this assignment, we continue with the theme of garbage collector that we explored in assignment 2. In Question 1, we look into *markF*, a general recursive function version of the marking function which was defined as an inductive relation in the assignment 2. Here we need to use Isabelle's **function** command as below and manually prove that the computation of *markF* does terminate. To prove termination, we need to define an appropriate measure that decreases with the each step of the computation.

Recall that, in the assignment 2, we used the following two types, parametrised by a type variable *'data*:

type-synonym *'data block = nat list × 'data*
type-synonym *'data heap = (nat, 'data block) fmap*

For the marking phase, we introduced a flag for marking by instantiating the type variable with *bool × 'data*. The block also contains a list of pointers, *nat list*, pointing to other blocks (see Figure 1 in the assignment 2). The function *markF* is defined using the two functions *marked* and *mark-block* as below:

```
fun marked :: (bool × 'data) block ⇒ bool where
  marked (ptr, (tag,data)) = tag

fun mark-block :: (bool × 'data) block ⇒ (bool × 'data) block  where
  mark-block (ptr, (tag,data)) = (ptr, (True,data))

function markF where
  markF heap [] = heap
| markF heap (root#roots) =
    (case fmap.lookup heap root of
       None ⇒ markF heap roots
     | Some blk ⇒ if marked blk then markF heap roots
                    else markF (fupd root (mark-block blk) heap) (roots@fst blk))
```

(a) Complete the definition of *markF* and prove its termination. For the termination proof, think which values will change in each loop step. (8 marks)

(b) Prove the *mark-correct-aux* lemma for *markF*. (7 marks)

markF heap root
fmap-keys (λ*ptr* , *tag* $\vee$ *ptr* $\in$ *ureach heap roots*, *data*)) *heap*

(c) Prove the corre marks)

fpred (λ*ptr block* $\Longrightarrow$
markF heap root
fmap-keys (λ*ptr* (*ptrs*, *tag*, *data*). (*ptrs*, *ptr* $\in$ *reach heap roots*, *data*)) *heap*

Note that the *ureach* and *reach* relations defined in the a3 template are identical to the ones in assignment 2.

2 C verification (78 marks)

In Question 2, we will consider a C version of the marking function, which is defined in the file *gc.c*. To keep the assignment manageable, we will concentrate on the properties leading up to the correctness of *mark*, but only state, not prove, the final lemma.

2.1 Linked lists

The C code uses the following struct `block` to model a block (a flat piece of data) in the heap. For the sake of simplicity, we assume that the actual data it carries is of type `int`:


```
struct block {
  struct blist *nexts;
  int flag;
  int data;
  struct block *m_nxt;
};
```

Here the `nexts` field corresponds to the list of pointers to other blocks that the block is linked to. The block struct also contains a `flag` field for marking, as well as the `m_nxt` ("marked next") field which is used for collecting marked blocks in the *mark* function.

In assignment 2, the list of `nexts` pointers in the block and the roots for the reachability relation were modelled using Isabelle's list datatype. In C, we use the following struct `blist` for these:

```
struct blist {
  struct block *this;
  struct blist *next;
};
```

which gives a linked list of blocks.

In order to further investigate the behaviour of the *mark* function, we first need to establish a correspondence between a `blist` pointer and an abstract representation of a list in Isabelle/HOL. We do this by using the following predicate *list*:

With the semantics given by AutoCorres, The `blist` pointer is given as a *blist-C ptr*: a pointer to a *blist-C* record, which models the `blist struct`. This record has two fields, corresponding to the two fields of the C struct: *this-C* and *next-C*. For instance, the selector function *this-C* has type *blist-C* $\Rightarrow$ *block-C ptr*, i.e. given a `blist` it returns the *block-C* pointer `this` field. Similarly, the *next-C* selector functions have type *blist-C* $\Rightarrow$ *blist-C ptr*.

The state type that the AutoCorres-generated functions operate on is called *lifted-globals*, AutoCorres also provides a heap for each type, i.e. here, *blist-C* heap, *heap-blist-C*, and *block-C* heap, *heap-block-C*, a[illegible]ctions, *is-valid-blist-C* and *is-valid-block-C*.

The following functio[illegible]*lobals* state, tests if a *blist-C ptr* points to a list whose pointer-structure cor[illegible]n *blist-C ptr list*. A *blist-C ptr list* is an abstract list whose elements are *b*[illegible]tructure gives us an *abstract* way of talking about the pointer-structure of a[illegible]heap.

primrec *list* :: *lifted-g*[illegible] *blist-C ptr list* $\Rightarrow$ *bool* **where**
  $list\ s\ p\ [] = (p = NULL)$
| $list\ s\ p\ (x\#xs) =$
    $(p = x \wedge p \neq NULL \wedge is\text{-}valid\text{-}blist\text{-}C\ s\ p \wedge list\ s\ (next\text{-}C\ (heap\text{-}blist\text{-}C\ s\ p))\ xs)$

Prove the following statements:

(a) `NULL` corresponds to an empty list. (3 marks)

$list\ s\ NULL\ xs = (xs = [])$

(b) If p is not `NULL`, p is an element of the list it points to. (3 marks)

$[\![list\ s\ p\ xs;\ p \neq NULL]\!] \Longrightarrow p \in set\ xs$

(c) When p is not `NULL`, the list it points to can be split into head and tail. (3 marks)

$p \neq NULL \Longrightarrow$
$list\ s\ p\ xs =$
$(\exists\ ys.\ xs = p\ \#\ ys \wedge is\text{-}valid\text{-}blist\text{-}C\ s\ p \wedge list\ s\ (next\text{-}C\ (heap\text{-}blist\text{-}C\ s\ p))\ ys)$

(d) p points to a unique list. (4 marks)

$[\![list\ s\ p\ xs;\ list\ s\ p\ ys]\!] \Longrightarrow xs = ys$

(e) The elements in a `blist` list are distinct. (6 marks)

$list\ s\ p\ xs \Longrightarrow distinct\ xs$

(f) Updating a pointer that is not in a `blist` list does not affect the list. (4 marks)

$q \notin set\ xs \Longrightarrow$
$list\ (heap\text{-}blist\text{-}C\text{-}update\ (\lambda h.\ h(q := next\text{-}C\text{-}update\ (\lambda\text{-}.\ v)\ (h\ q)))\ s)\ p\ xs = list\ s\ p\ xs$

Next, we define a function *the-list* which, given a state s and a pointer p for which $list\ s\ p\ xs$ holds for a `blist` list xs, returns the actual list xs.

definition *the-list* :: *lifted-globals* $\Rightarrow$ *blist-C ptr* $\Rightarrow$ *blist-C ptr list* **where**
  $the\text{-}list\ s\ p = (THE\ xs.\ list\ s\ p\ xs)$

(g) Prove that updating a pointer that is not in the list does not affect the list in terms of *the-list*. (5 marks)

$[\![q \notin set\ xs;\ list\ s\ p\ xs]\!]$
$\Longrightarrow the\text{-}list\ (heap\text{-}blist\text{-}C\text{-}update\ (\lambda h.\ h(q := next\text{-}C\text{-}update\ (\lambda\text{-}.\ v)\ (h\ q)))\ s)\ p =$
$\quad the\text{-}list\ s\ p$

2.2 Correctness of *append′*

AutoCorres generates a monadic version of *mark*, which is named *mark′*. Similarly, functions *append′* and *append-step′*, which *mark′* depends on, are also generated. We now prove the correctness of *append-step′* and *append′*, which correspond to the following C functions:

```
struct blist *append_step (struct blist *l1, struct blist *l2)
{
  struct blist *tmp = l1->next;
  l1->next = l2;
  l2 = l1;
  l1 = tmp;
  return l1;
}

struct blist *append (struct blist *l1, struct blist *l2)
{
  while (l1) {
    struct blist *tmp = append_step (l1, l2);
    l2 = l1;
    l1 = tmp;
  }
  return l2;
}
```

append takes two `blist` pointers representing two disjoint block lists, say *l1* and *l2*, and returns a pointer that represents a list *rev l1* @ *l2* (concatenation of the two lists but the first one reversed). *append-step* implements each step of *append*, where the head of the first list is moved to the head of the second list.

(h) Complete the proof of the correctness of *append-step′*. (8 marks)

(i) Provide the precondition for the correctness statement for *append′*. (4 marks)

(j) Prove the correctness of *append′* by providing appropriate invariants and a measure and completing the rest of the proof. (12 marks)

2.3 Marked list

The function *mark* is defined so that, as it marks the blocks, it collects all the marked block as another linked list, using the `m_nxt` field of the block. When it finishes marking, it returns the pointer that corresponds to the list of all marked blocks. This list is similar to the linked lists `nexts` and *roots* that we already saw, except that this one consists directly of blocks using the `m_nxt` field, rather than being wrapped as `blist`.

We can use this list of marked blocks to retrieve the set of all marked blocks. And together with the reachability notion on blocks, we will be able to state the conditions we expect to hold for the function *mark*.

For this, we first need to define an equivalent of *list* predicate for the marked list.

primrec *mkd-list* :: *lifted-globals* ⇒ *block-C ptr* ⇒ *block-C ptr list* ⇒ *bool*

(k) Complete the definition of *mkd-list* based on the definition of *list*. (5 marks)

definition *the-mkd-list* :: *lifted-globals* ⇒ *block-C ptr* ⇒ *block-C ptr list* **where**
the-mkd-list s p = (*THE xs. mkd-list s p xs*)

(l) Prove the uniqueness of *mkd-list*. (7 marks)

$[\![\textit{mkd-list}\ s\ p\ xs;\ \textit{mkd-list}\ s\ p\ ys]\!] \Longrightarrow xs = ys$

Next, we define reachability on blocks. The relation *b-reach* will specify the set of reachable `blocks` (the set of b[illegible]st) from a given set of roots in state s. To define it, you can use the pred[illegible] b which tests if the `block` b is in the linked list at the `blist` pointer p und[illegible]

(m) Define *b-reach* [illegible] 5 marks)

(n) Prove *b-reach-*[illegible]

$x \in \textit{b-reach}\ s\ rts \Longrightarrow$
$\textit{block-in-list}\ s\ rts\ x \lor (\exists \textit{block}.\ \textit{block-in-list}\ s\ (\textit{nexts-C}\ (\textit{heap-block-C}\ s\ \textit{block}))\ x)$

(o) State the correctness property we expect to hold after calling $\textit{mark}'\ \textit{roots}$ as the post-condition of the Hoare triple in the assignment template. (5 mark)

You do not have to prove this post-condition, only state it.