COMP4161 T3/2022
Advanced Topics in Software Verification

[illegible]signment 2

This assignment sta[illegible]tober 2022 and is due on Friday 4nd November 2022 6pm. We will accept[illegible]) files only.

The assignment is ta[illegible]NOT mean you can work in groups. Each submission is personal. For mor[illegible] plagiarism policy: https://student.unsw.edu.au/plagiarism

Submit using give o[illegible]

```
give cs4161 a2 files ...
```

For example:


```
give cs4161 a2 a2.thy a2_fmap.thy
```


For this assignment, all proof methods aand proof automation available in the standard Isabelle distribution is allowed. This includes, but is not limited to, `simp`, `auto`, `blast`, `force`, and `fastforce`.

However, if you're going for full marks, you shouldn't use "proof" methods that bypass the inference kernel, such as `sorry`. We *may* award partial marks for plausible proof sketches where some subgoals or lemmas are sorried.

If you use `sledgehammer`, it's important to understand that the proofs suggested by `sledgehammer` are just suggestions, and aren't guaranteed to work. Make sure that the proof suggested by `sledgehammer` actually terminates on your machine (assuming an average spec machine). If not, you can try to reconstruct the proof yourself based on the output, or apply a few manual steps to make the subgoal smaller before using `sledgehammer`.

Note: this document contains explanations of the problems and your assignment tasks. The full set of definitions can be found in the associated Isabelle theory files.

Hint: there are hints at the end of this document.

0 Introduction

The *garbage collector* is the most important runtime component for programming languages with automatic memory management. The role of the garbage collector is to detect when data allocated on the heap is no longer in use, and free it. This liberates the programmer from having to juggle error-prone `malloc`s and `free`s

In this assignment, we will verify (a model of) a mark-and-sweep garbage collector.

This assignment spans over two theory files, a2_fmap.thy and a2.thy. Both files contain questions, and, in particular, in a2_fmap.thy, the library development and questions are interleaved, so make sure you go through the file and attempt them all!

1 Finite map library (22 marks)

We will build our model of a garbage collector using a library of finite maps. A finite map is a partial map whose domain is finite. In Isabelle, a partial map (type $'a \rightharpoonup 'b$) is implemented using an option type: if f has type $'a \to 'b$ and x is in the domain of f (i.e., f is defined for x), there exists y such that $f\ x = Some\ y$. If f is not defined for x, we have $f\ x = None$.

In a2_fmap.thy, the type of finite map $('a, 'b)\ fmap$ is defined using *typedef*, and various operations on fmaps are defined and their properties are proved.

(a) Explain how th[illegible] as an example of what you learned in the lecture. Also give a brief desc[illegible]s Isabelle generates. What exactly is *lookup*? (4 marks)

(b) Prove the exte[illegible] *lookup* (3 marks)

$(\bigwedge x.\ lookup\ f\ x$ [illegible] $= g$

(c) *fmap-filter* $P\ f$ is a fmap whose domain is the domain of f restricted only to x such that $P\ x$ holds. Prove that the domain of *fmap-filter* $P\ f$ is equal to the domain of f restricted to P. (4 marks)

$fdom\ (fmap\text{-}filter\ P\ m) = Set.filter\ P\ (fdom\ m)$

(d) *fmap-of* converts a $('a \times 'b)\ list$ to a $('a, 'b)\ fmap$. Prove simplification rules for *fmap-of*. (3 marks)

$fmap\text{-}of\ [] = fempty$

$fmap\text{-}of\ ((k, v)\ \#\ kvs) = fupd\ k\ v\ (fmap\text{-}of\ kvs)$

(e) Prove a lemma about *fmap-keys* and *fpred*. (5 marks)

$fpred\ P\ (fmap\text{-}keys\ f\ m) = fpred\ (\lambda a\ b.\ P\ a\ (f\ a\ b))\ m$

(f) *fmmap* $f\ m$ takes a function $f{:}'b \Rightarrow 'c$ and an fmap $m{:}('a, 'b)\ fmap$ and returns an fmap of type $('a, 'c)\ fmap$. Prove the following equality about *lookup* and *fmmap*. (3 marks)

$lookup\ (fmmap\ f\ m)\ x = map\text{-}option\ f\ (lookup\ m\ x)$

2 Garbage collector specification (30 marks)

We will use natural numbers to represent memory addresses (politely ignoring the inconvenience that real memory is finite). A *block* is a flat piece of data that resides on the heap; for example, in a Java runtime, blocks might represent objects. At this level of abstraction, we don't particularly care how blocks are laid out in memory. All we need to know is that blocks contain a list of pointers to other blocks, and some (non-pointer) data:

type-synonym $'data\ block = nat\ list \times 'data$

We use a type variable to represent the non-pointer data, because we do not (yet) care about its structure. A *heap* is a finite map that associates memory addresses to *blocks*:

type-synonym $'data\ heap = (nat, 'data\ block)\ fmap$

The intuition is that $lookup\ h\ a = Some\ b$ holds if, in the heap h, if we dereference the pointer a, we'll find the block b. If $lookup\ h\ a = None$ holds, that means that there is no block at address a.

2.1 Reachability

The relation *reach* specifies the set of reachable addresses in a heap h from a given set of *roots*. Intuitively, the roots are memory addresses that can be reached directly from outside the heap; a real-world example is a heap pointer residing in a stack frame.

inductive-set *reach* :: *'data heap* ⇒ *nat list* ⇒ *nat set* **for** *h roots*
where
reach-root[*intro*]: $a \in set\ roots \Longrightarrow a \in reach\ h\ roots$
| *reach-step*[*intro*]: $[\![b \in reach\ h\ roots;\ lookup\ h\ b = Some(as,data);\ a \in set\ as]\!] \Longrightarrow a \in reach\ h\ roots$

The first rule states t[illegible]lves are always reachable. The second rule states that we can reach an add[illegible]pointer from within a reachable block.

Prove that the *reach*[illegible]following properties:

(a) If we have no r[illegible]able. (3 marks)

$a \in reach\ h\ []$ [illegible]

(b) Reachability is monotonic wrt. the roots. (3 marks)

$[\![a \in reach\ h\ roots;\ set\ roots \subseteq set\ roots']\!] \Longrightarrow a \in reach\ h\ roots'$

(c) Any address reachable from a reachable address is reachable. (3 marks)

$[\![a \in reach\ h\ roots;\ b \in reach\ h\ [a]]\!] \Longrightarrow b \in reach\ h\ roots$

(d) If the roots contain a dangling pointer x, this adds no reachable elements except the dangling pointer itself. (3 marks)

$[\![lookup\ h\ x = None;\ a \in reach\ h\ (x\ \#\ roots)]\!] \Longrightarrow x = a \lor a \in reach\ h\ roots$

(e) Reachable addresses can be found either in the roots or in a block. (4 marks)

$x \in reach\ heap\ roots \Longrightarrow x \in set\ roots \lor (\exists block.\ block \in fran\ heap \land x \in set\ (fst\ block))$

2.2 Collection

We can now specify the expected behaviour of a garbage collector as follows:

fun *collect* :: *'data heap* ⇒ *nat list* ⇒ *'data heap*
where
$collect\ h\ roots = frestrict\text{-}set\ (reach\ h\ roots)\ h$

The garbage collector restricts the heap h to the set of reachable addresses. Or, for a more operational intution, it removes all unreachable addresses from the heap.

(f) Consider the following example heap, which is illustrated in Figure 1:

$ex1\text{-}heap = fmap\text{-}of\ [(0,\ [1,\ 2],\ ()),\ (1,\ [0],\ ()),\ (2,\ [2],\ ()),\ (3,\ [0],\ ())]$

Define a set of addresses *ex1-reach*, which is the set of reachable addresses from the root *0*. (3 marks)

(g) Use your answer to the previous question to show that after garbage collection, the expected addresses are all reachable. (3 marks)

$ex1\text{-}reach \subseteq fdom\ (collect\ ex1\text{-}heap\ [0])$

(h) Show that garbage collection is sound, in the sense that it doesn't collect reachable blocks (preserves reachability). (3 marks)

$n \in reach\ h\ roots \Longrightarrow n \in reach\ (collect\ h\ roots)\ roots$

(i) Show that garbage collection is complete, in the sense that it collects all garbage. (3 marks)

$[\![n \in fdom\ h;\ n \notin reach\ h\ roots]\!] \Longrightarrow n \notin fdom\ (collect\ h\ roots)$

(j) Show that running the garbage collector again does nothing. (2 marks)

$collect\ (collect\ h\ roots)\ roots = collect\ h\ roots$

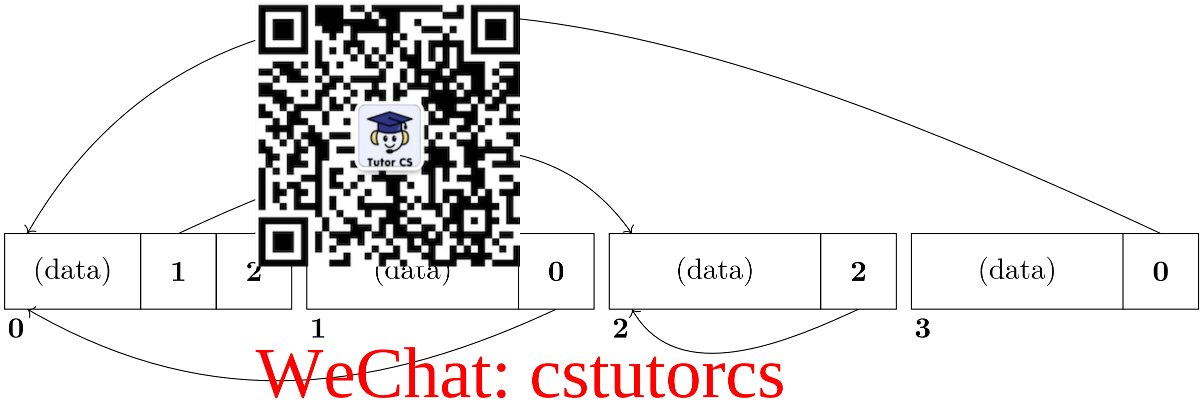

Figure 1: Illustration of the heap from Question 2(g). Blocks are represented as rectangles partitioned into data segments and pointers. The number below each block is its address in the heap. The arrows show where pointers point.

3 Garbage collector refinement (48 marks)

In this part, we will elaborate the garbage collector specification by introducing more implementation details, and prove our elaboration sound with respect to our specification. This process is called *refinement*. In a real-world application, we could perform more and more refinement steps, adding further implementation details until we're down to the machine code level. For the sake of your sanity, we'll only do this one step in this assignment.

In a *mark-and-sweep* collector, the first step is to traverse the heap and *mark* all reachable blocks. We assume all blocks have a special tag bit for this purpose, which is only used internally by the GC. When the marking phase finishes, a *sweep* is conducted: the entire heap is traversed top-to-bottom. In this process, any unmarked block is freed and marked blocks are kept and their markings are removed.

For the sweep process, we need blocks to additionally contain this tag bits. Here it is handy that we chose a type variable for the data, which we can now instantiate to $(bool \times 'data)$ *block*, on the understanding that the boolean contains the tag bit. The following auxiliary functions test whether a block is marked, mark it, and unmark it, respectively:

fun *marked* :: $(bool \times 'data)$ *block* $\Rightarrow$ *bool*
  where *marked*$(ptr,(tag,data))$ = *tag*

fun *mark-block* :: $(bool \times 'data)$ *block* $\Rightarrow$ $(bool \times 'data)$ *block*
  where *mark-block*$(ptr,(tag,data))$ = $(ptr,(True,data))$

fun *unmark-block* :: $(bool \times 'data)$ *block* $\Rightarrow$ $(bool \times 'data)$ *block*
  where *unmark-block*$(ptr,(tag,data))$ = $(ptr,(False,data))$

3.1 Mark

The following inductive relation defines the behaviour of the marking phase:

inductive *mark* **where**
  mark-done[*intro!*,*simp*]: *mark heap* [] *heap*

| *mark-dangling*[*elim*]:
  $[\![$*lookup heap root* = *None*; *mark heap roots new-heap*$]\!]$
  $\Longrightarrow$ *mark heap* (*root*#*roots*) *new-heap*
| *mark-marked*[*intro*]:
  $[\![$*lookup heap root* = [illegible]
  marked block;
  mark heap roots ne[illegible]
  $\Longrightarrow$ *mark heap* (*roo*[illegible]
| *mark-unmarked*[*intro*[illegible]
  $[\![$*lookup heap root* = [illegible]
  ¬*marked block*;
  mark (*fupd root* (*m*[illegible]) (*roots*@*fst block*) *new-heap*$]\!]$
  $\Longrightarrow$ *mark heap* (*root*#*roots*) *new-heap*

The intention is that *mark old-heap roots new-heap* is true if the result of marking *old-heap*, starting from *roots*, is *new-heap*. The new heap will contain the same data and blocks as the old heap, except reachable blocks become marked.

The argument *roots* maintains a list of memory addresses we need to visit in the future. When we visit an address, if it points to a marked block, we ignore it (on the understanding that it's already been visited). If it points to an unmarked block, we mark it, and add its children to the roots.

3.2 Unmarked reachability

To connect the GC specification *reach* with the marking implementation *mark*, the following definition turns out to be useful.

abbreviation *unmarked-root* **where**
  unmarked-root ≡ λ*h root*. *case lookup h root of None* ⇒ *True* | *Some block* ⇒ *Not*(*marked block*)

inductive-set *ureach*:: (*bool* × *data*) *heap* ⇒ *nat list* ⇒ *nat set* **for** *h roots*
  where
  ureach-root[*intro*]: $[\![$*a* ∈ *set roots*; *unmarked-root h a*$]\!]$ $\Longrightarrow$ *a* ∈ *ureach h roots*
| *ureach-step*[*intro*]: $[\![$*b* ∈ *ureach h roots*; *lookup h b* = *Some*(*as*,(*False*,*data*)); *a* ∈ *set as*;
  unmarked-root h a$]\!]$
  $\Longrightarrow$ *a* ∈ *ureach h roots*

Intuitively *ureach* denotes *reachability via unmarked blocks* (*u-reachability* for short). It's like *reach*, except you're not allowed to visit marked blocks.

The following properties of it will be handy, many of which are similar to those for *reach*:

(a) U-reachability is monotonic wrt. the roots. (3 marks)

$[\![$*a* ∈ *ureach h roots*; *set roots* ⊆ *set roots'*$]\!]$ $\Longrightarrow$ *a* ∈ *ureach h roots'*

(b) If the roots contain a dangling pointer *x*, this adds no u-reachable elements except the dangling pointer itself. (3 marks)

$[\![$*lookup h x* = *None*; *a* ∈ *ureach h* (*x* # *roots*)$]\!]$ $\Longrightarrow$ *a* = *x* ∨ *a* ∈ *ureach h roots*

(c) Removing the address of a marked block from the roots does not impact u-reachability. (3 marks)

$[\![$*lookup h x* = *Some* (*ptrs*, *True*, *tags*); *a* ∈ *ureach h* (*x* # *roots*)$]\!]$ $\Longrightarrow$ *a* ∈ *ureach h roots*

(d) If an address can be u-reached after marking an unmarked block and adding its children to the roots, it could be u-reached before too. (5 marks)

$[\![$*lookup heap root* $=$ *Some* $($*a*, *False*, *b*$)$; $x \in$ *ureach* $($*fupd root* $($*a*, *True*, *b*$)$ *heap*$)$ $($*roots* @ *a*$)]\!] \Longrightarrow x \in$ *ureach heap* $($*root* # *roots*$)$

(e) If an address can be u-reached from an unmarked root, marking this address and adding its children as [illegible] u-reachability of other addresses. (6 marks)

$[\![$*lookup heap r*[illegible], *b*$)$; $x \in$ *ureach heap* $($*root* # *roots*$)]\!] \Longrightarrow x \in$ *ureach* $($*fupd root* $($*a*, [illegible] @ *a*$)$ $\lor$ $x =$ *root*

(f) Running *mark* [illegible]s that are u-reachable through unmarked blocks: (7 marks)

mark heap roo[illegible]*eap* $=$ *fmap-keys* $(\lambda$*ptr* $($*ptrs*, *tag*, *data*$)$. $($*ptrs*, *tag* $\lor$ *ptr* $\in$ *ureach heap roots*, *data*$))$ *heap*

(g) U-eachability implies reachability. (3 marks)

$a \in$ *ureach h roots* $\Longrightarrow a \in$ *reach h roots*

(h) Reachability implies u-reachability, if all blocks are unmarked. (6 marks)

$[\![a \in$ *reach h roots*; *fpred* $(\lambda$*ptr block*. $\neg$ *marked block*$)$ *h*$]\!] \Longrightarrow a \in$ *ureach h roots* $\lor$ $a \in$ *set roots*

(i) *mark* marks all reachable blocks, if everything is initially unmarked. (6 marks)

$[\![$*mark heap roots newheap*; *fpred* $(\lambda$*ptr block*. $\neg$ *marked block*$)$ *heap*$]\!] \Longrightarrow$ *newheap* $=$ *fmap-keys* $(\lambda$*ptr* $($*ptrs*, *tag*, *data*$)$. $($*ptrs*, *ptr* $\in$ *reach heap roots*, *data*$))$ *heap*

3.3 Sweep

Our characterisation of the sweeping phase will be considerably more abstract than the marking; for example, we don't worry about characterising the step-by-step behaviour of the sweeping function.

A *sweep* removes all marked blocks (with *fmap-filter*) and then unmarks all blocks (with *fmmap*).

definition *sweep* :: $($*bool* $\times$ *'data*$)$ *heap* $\Rightarrow$ $($*bool* $\times$ *'data*$)$ *heap* **where**
sweep h $=$ *fmmap unmark-block* $($*fmap-filter* $($*Not o unmarked-root h*$)$ *h*$)$

This allows us to conclude our refinement story by showing that together, *mark* and *sweep* implements *collect*.

(j) Prove the following theorem statement. (6 marks)

$[\![$*mark h roots h'*; *fpred* $(\lambda$*ptr block*. $\neg$ *marked block*$)$ *h*$]\!] \Longrightarrow$ *collect h roots* $=$ *sweep h'*

4 Hints

- The lemma you prove for 1(a), called *lookup-ext*, is often useful in proving lemmas for the later questions. Applying it as an introduction rule tends to unlock a lot of simplification.
- For proving *mark-correct-aux* proof, the *ureach* lemmas are useful.
- Many proofs will require induction of one kind or the other. Other than inducting on datatypes directly, you may find it useful to do induction on inductively defined sets and relations such as *reach* and *mark*. The induction rules for these are automatically generated by Isabelle.

You can apply these induction rules as elimination rules, e.g. `apply(erule reach.induct)`, but a more convenient and flexible alternative is

```
apply(induct rule: reach.induct)
```

which allows y[illegible]ariables should not be all-eliminated using e.g.

```
[illegible]itrary: x y rule: reach.induct)
```

- Not everything [illegible]

- The assumptio[illegible]d in the `assumes-show` format are accessible via the fact named `assms`. For example, you can do `simp add: assms` or `rule assms(1)`. The assumptions can be added directly to the goal state by beginning your proof with the command `apply(insert assms)` or `using assms`; that is usually what you want to do before starting an induction.

- The equivalent of `spec` for the meta-logic universal quantifier, if you need it, is called `meta_spec`.

- For some exercises, you will likely need additional lemmas to make the proof go through. Part of the assignment is figuring out which lemmas are needed.

- Make use of the `find_theorems` command to find library theorems. You are allowed to use all theorems proved in the Isabelle distribution.

5 Acknowledgements

This assignment is inspired by Magnus O. Myreen's paper *Reusable verification of a copying collector*. The finite map library is based on a finite map library by Lars Hupel.