COMP4161 S2/2015
Advanced Topics in Software Verification

Assignment 2

This assignment starts on Fri, 2015-10-09 and is due on Fri, 2015-10-30, 23:59h.
We will accept Isabelle .thy files only.
Submit using `give` on a CSE machine:

```
give cs4161 a3 files ...
```

For example:

```
give cs4161 a3 a3.thy
```

Hint: *the questions in this assignment are phrased so that things that you prove in earlier sub-questions may often be useful to you in later sub-questions. 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 Regular Expressions (21 marks)

Regular expressions are widely used. For instance, JFlex (http://jflex.de/) is a tool that generates *lexers* for Java, taking as input a set of regular expressions and corresponding actions, and generating a program of the lexer. Lexers usually are the first front-end step in e.g. compilers.

In the second part of week 6 in the lecture, we have seen a theory modelling a *subset* of regular expressions, such as $!((e1 \cdot e2 \mid <CHR\ ''c''>)\star)$, and we proved properties about the language generated by regular expressions.

More regular expression constructs can be defined, and derived from this subset, such as *e?*, *e+*, etc.

In this exercise, we will extend the regular expressions defined in week 6, and prove properties about when a regular expression may match the empty word, i.e. when the empty word is recognised by its language.

This *maybeempty* is a function in JFlex that is implemented here:

https://github.com/jflex-de/jflex/blob/master/jflex/src/main/java/jflex/SemCheck.java#L58-L111

First let us consider the definition of regular expressions from week 6:

datatype *regexp* =
  Atom char (<->)
| *Alt regexp regexp* (**infixl** | *50*)
| *Conc regexp regexp* (**infixl** · *60*)
| *Star regexp* (- ⋆ [*79*] *80*)
| *Neg regexp* (!- [*70*] *70*)

primrec
  $lang :: regexp \Rightarrow string\ set$
where
  $lang\ (Atom\ c) = \{[c]\}$
| $lang\ (Alt\ e1\ e2) = lang\ e1 \cup lang\ e2$
| $lang\ (Conc\ e1\ e2) = conc\ (lang\ e1)\ (lang\ e2)$
| $lang\ (Star\ e) = star\ (lang\ e)$
| $lang\ (Neg\ e) = -\ (lang\ e)$

Let U be the regular expression defining the universal language, i.e. the language of all words.

definition
  $U :: regexp$ **where** $U \equiv Alt\ (Atom\ CHR\ ''x'')\ \ (Neg\ (Atom\ CHR\ ''x''))$

Let N be the regular expression defining the empty language, i.e. the language containing no words.

definition $N :: regexp$ **where** $N \equiv Neg\ U$

Let E be the regular expression defining the language containing only the empty word.

definition $E :: regexp$ **where** $E \equiv Star\ N$

(a) Prove that the language recognised by U is indeed the universal set $UNIV$, that the language recognised by N is indeed the empty set, and that the language recognised by E is indeed the set containing only the empty word:
$lang\ U = UNIV$
$lang\ N = \emptyset$
$lang\ E = \{[]\}$ (4 marks)

(b) Define a function *String* (using primrec), that takes a list of characters (a string), and produces the regular expression recognising that string. If the list is empty, the regular expression should recognise only the empty word. Check (prove) that
$lang\ (String\ xs) = \{xs\}$. (2 marks)

(c) Define a function *Maybe*, taking a regular expression e, and producing the regular expression $e?$ recognising either e or empty word. Check (prove) that
$lang\ (e?) = \{[]\} \cup lang\ e$. (1 mark)

(d) Define a function *Plus*, taking a regular expression e, and producing the regular expression $e+$ recognising one or more concatenations of e. Check (prove) that
$[] \notin lang\ e \Longrightarrow lang\ (e+) = lang\ (e\ \star) - \{[]\}$. (3 marks)

(e) Define a function *CClass*, taking a list of characters, and producing the regular expression recognising any of these single characters, i.e. $CClass\ [c1,\ c2]$ recognises the two words $[c1]$ and $[c2]$. Check (prove) that $lang\ (CClass\ xs) = set\ (map\ (\lambda c.\ [c])\ xs)$. (2 marks)

(f) Consider the function *Tilde*, taking a regular expression e, and producing the regular expression recognising words made of any string not containing e followed by one e. This function is for instance useful to define comments. Note that *enum* enumerates all elements of a finite type:

$Tilde\ e \equiv\ !(AnyChar\ \star\ \cdot\ e\ \cdot\ AnyChar\ \star)\ \cdot\ e$
$AnyChar \equiv CClass\ enum$

Prove that $\{w\ @\ [c]\ |\ c \notin set\ w\} \subseteq lang\ (Tilde\ <c>)$. (1 mark)

(g) Define a function *maybeempty* (using primrec) taking a regular expression e of type *regexp* (i.e. one of the five type of regular expressions of week 6) and returning True if and only

if this e may contain the empty word. Check (prove) that $maybeempty\ e = ([] \in lang\ e)$. (3 marks)
Hint: you may look at how JFlex defines this function.

(h) Verify that the JFlex definitions for the five additional regular expression type are correct. I.e. prove that *maybeempty* of the five new types of regular expressions (*Maybe*, *Plus*, *CClass* and *Tilde*) is equal to what JFlex defines:
$maybeempty\ (e?) = True$
$maybeempty\ (e+) = maybeempty\ e$
$maybeempty\ (CClass\ xs) = False$
$maybeempty\ (String\ xs) = (length\ xs = 0)$
$maybeempty\ (Tilde\ c) = False$
(5 marks)

2 Termination (15 marks)

Let a function f be defined by

$f\ xs = (\textsf{if}\ xs = []\ \textsf{then}\ []\ \textsf{else}\ f\ (butlast\ (g\ xs))\ @\ [last\ (g\ xs)])$

where g is defined by

$g\ [] = []$
$g\ (x\ \#\ xs) = xs\ @\ [x]$

(a) Explain (in words) what the function f is computing (2 marks)

(b) Define g using primrec (3 marks)

(c) Define f using function (8 marks)

(d) Prove that f is equivalent to the pre-defined Isabelle function computing the same thing. (2 marks)

3 C Verification: Binary Search (64 marks)

In this question, we will verify the correctness of the binary search implementation we saw in the first week of the lecture. You will remember that the Java code shown in that lecture had a bug. We transcribed that Java code into C (see the file *binsearch.c*), preserving the bug. The task in this question is to find the preconditions under which the code works correctly and to prove that it does so.

The template uses the C parser and AutoCorres to convert the C code into a monadic specification in Isabelle. We will prove properties about this AutoCorres output.

The code operates on an array of signed integers (*int* []), and we will some functions to describe its properties. We start with a function that enumerates the addresses (pointers) that the array contains:

fun *array-addrs* :: *s-int ptr* ⇒ *nat* ⇒ *s-int ptr list* **where**
  $array\text{-}addrs\ p\ 0 = []\ |$
  $array\text{-}addrs\ p\ (Suc\ len) = p\ \#\ array\text{-}addrs\ (p +_p 1)\ len$

(a) We will have to deal with the fact that the C heap stores unsigned words, but the program mostly uses signed C ints. Signed and unsigned pointers can be converted into each other using the function *ptr-coerce*.

Use *array-addrs* to define a function that takes an unsigned int heap (a function from *u-int ptr* to *u-int*), the array base address (a signed int pointer), and a length (an Isabelle *int*), and returns the elements of the array as a list. (5 marks)

(b) Use *array-addrs* again to define a function that takes a heap validity predicate such as *is-valid-w32* in the AutoCorres output *binary-search′*, as well as an unsigned int heap, and asserts that each array address is a valid C pointer. The function should also place a condition on the values that the array stores. Find and add this condition. Hint: it is probably easier to explore the invariant and proof obligations of the program first before you add the additional condition. (6 marks)

(c) Prove the following lemmas

$$\mathit{length}\ (\mathit{array\text{-}addrs}\ a\ \mathit{len}) = \mathit{len}$$

(2 marks)

$$[\![0 \le x;\ \mathit{nat}\ x < \mathit{len}]\!] \Longrightarrow \mathit{array\text{-}addrs}\ a\ \mathit{len}\ !\ \mathit{nat}\ x = a +_p x$$

(5 marks)

$$\begin{aligned}&[\![0 \le x;\ x < \mathit{len}]\!]\\ &\Longrightarrow \mathit{uint}\ (\mathit{heap\text{-}w32}\ s\ (\mathit{ptr\text{-}coerce}\ a +_p x)) =\\ &\quad\ \mathit{array\text{-}list}\ (\mathit{heap\text{-}w32}\ s)\ a\ \mathit{len}\ !\ \mathit{nat}\ x\end{aligned}$$

(2 marks)

$$\begin{aligned}&[\![\mathit{key} < \mathit{xs}\ !\ \mathit{nat}\ \mathit{mid};\ \mathit{mid} - 1 < x;\ \mathit{sorted}\ \mathit{xs};\ 0 \le \mathit{mid};\ x < \mathit{int}\ (\mathit{length}\ \mathit{xs})]\!]\\ &\Longrightarrow \mathit{key} < \mathit{xs}\ !\ \mathit{nat}\ x\end{aligned}$$

(5 marks)

$$\begin{aligned}&[\![\mathit{xs}\ !\ \mathit{nat}\ \mathit{mid} < \mathit{key};\ 0 \le x;\ x \le \mathit{mid};\ \mathit{sorted}\ \mathit{xs};\ \mathit{mid} < \mathit{int}\ (\mathit{length}\ \mathit{xs})]\!]\\ &\Longrightarrow \mathit{xs}\ !\ \mathit{nat}\ x < \mathit{key}\end{aligned}$$

(5 marks)

(d) Main correctness theorem. The template *a3.thy* contains a partially filled in correctness theorem for binary search.

Strengthen the precondition with additional conditions such that the post-condition of *binary-search′* is provable. Think about which precondition is needed to avoid triggering the overflow bug that is present in the code. (5 marks)

(e) Adjust the invariant with any additional conditions you need to prove the lemma. (10 marks)

(f) Prove correctness of the binary search implementation. (15 marks)