Problem Sheet 2: Auto-Active Verification

Chris Poskitt*
ETH Zürich

"Beware of bugs in the above code; I have only proved it correct, not tried it."
– Donald E. Knuth

Starred exercises (∗) are more challenging than the others.

1 Boogie

In this first set of exercises, we will work directly with Boogie [1], the intermediate verification language and automatic verification framework developed by Microsoft Research. For an introduction to the language and verifier, consult the following slides and manual:

- The lecture slides:
  http://se.inf.ethz.ch/courses/2015b_fall/sv/slides/05-AutoActiveVerification.pdf
- Boogie manual:
  http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf

1.1 Setting Up

Boogie can be interacted with through your web browser:

http://rise4fun.com/boogie

Should you prefer, you can also install it locally:

http://research.microsoft.com/en-us/projects/boogie/

*With input from Nadia Polikarpova and Julian Tschannen.

1.2 Exercises

i. Prove the following specification in Boogie:

$$\{\text{true}\}\ t := x;\ x := x + y;\ y := t\ \{x = x' + y' \land y = x'\}$$

where x', y' respectively denote the values of x and y in the pre-state.

ii. Consider the Boogie program `ArraySum` which is supposed to recursively compute the sum of array elements:

http://rise4fun.com/Boogie/2kR

Fix the program and then verify it in Boogie.

Hint: don't forget about loop invariants! Without an invariant, any loop in Boogie is treated as equivalent to assigning arbitrary values to program variables.

iii. Implement and verify the algorithm `FindZero` (its signature is given in http://rise4fun.com/Boogie/SciP), that linearly searches an array for the element 0:

Input: an integer array a, and its length N.   *Output:* an index $k \in \{0, \dots, N-1\}$ into the array a such that $a[k] = 0$; otherwise $k = -1$.

The specification should guarantee that if there exists an array element $a[i] = 0$ with $0 \leq i < N$, then `FindZero` will always return a k such that $k \geq 0$ and $a[k] = 0$.

iv. ($*$) Copy the procedure `FindZero` you wrote in part (iv), rename it to `FindZeroPro`, and add the following two preconditions:

```
requires (forall i: int :: 0 <= i && i < N ==> 0 <= a[i]);
requires (forall i: int :: 0 <= i-1 && i < N ==> a[i-1]-1 <= a[i]);
```

These additional preconditions require that along the array, values never decrease by more than one. Adapt your linear search algorithm such that after an iteration of its loop, instead of incrementing the current index k by 1, it now increments it by $a[k]$. Verify that the procedure still establishes the same postconditions as in (iv).

Hint: you will need to prove that all array values between $a[k]$ and $a[k+a[k]]$ are non-zero (i.e. that 0-values are not skipped over by the search) and use this property in the loop. For this you will need to write more than simply a loop invariant, e.g. some "ghost" (or "proof") code.

v. ($*$) Take a look at the Boogie program `BinarySearch` which is supposed to perform a binary search[1]:

http://rise4fun.com/Boogie/Ilf

Debug the implementation and add the missing loop invariants.

[1] See: http://en.wikipedia.org/wiki/Binary_search_algorithm

2 AutoProof

This second set of exercises is concerned with AutoProof [2], an auto-active verifier for programs written in (a subset of) the object-oriented language Eiffel. The tool takes an Eiffel program annotated with *contracts* (i.e. executable pre-/postconditions, class invariants, intermediate assertions), and translates it into a Boogie program for verification. Errors returned by the Boogie verifier are traced back to the relevant parts of the original Eiffel program (see Figure 1).

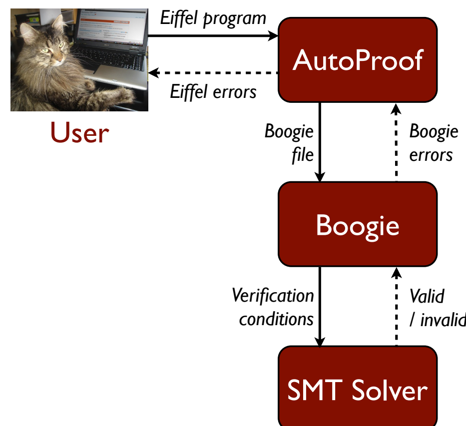

Figure 1: The AutoProof workflow

Extensive documentation (including a manual, tutorial, and software repository) is available online:

`http://se.inf.ethz.ch/research/autoproof/`

2.1 Exercises

The Eiffel programs for the following exercises are all available online in a web-based interface to AutoProof. Simply follow the given links for each exercise, edit the code and contracts in your browser, then hit the "Verify" button to run the tool. Note that the web interface to AutoProof does not permanently save your changes, so please make sure to save local copies of your solutions.

i. Consider the class `WRAPPING_COUNTER` in:

   `http://cloudstudio.ethz.ch/e4pubs/#sv-task1`

   The method **`increment`** increases its integer input by one, *except* if the input is 59, in which case it wraps it round to 0. Verify the class in AutoProof *without* changing the implementation, i.e. adding only the necessary preconditions. Strengthen the postcondition further as suggested in the comments, and check that the proof still goes through.

ii. In the axiomatic semantics problem sheet, we encountered several simple program specifications expressed as Hoare triples. Using the class `AXIOMATIC_SEMANTICS` in:

   `http://cloudstudio.ethz.ch/e4pubs/#sv-task2`

   write some simple contract-equipped methods and show the following in AutoProof:

(A) $\models$ $\{x = 21 \wedge y = 5\}$ `skip` $\{y = 5\}$
(B) $\models$ $\{x > 10\}$ $\texttt{x} := 2 * \texttt{x}$ $\{x > 21\}$
(C) $\models$ $\{x \geq 0 \wedge y > 1\}$ $\texttt{while x} < \texttt{y do x} := \texttt{x} * \texttt{x}$ $\{x \geq y\}$
(D) $\models$ $\{x = 5\}$ $\texttt{while x} > 0 \texttt{ do x} := \texttt{x} + 1$ $\{x < 0\}$
(E) $\models$ $\{x = a \wedge y = b\}$ $\texttt{t} := \texttt{x};\ \texttt{x} := \texttt{x} + \texttt{y};\ \texttt{y} := \texttt{t}$ $\{x = a + b \wedge y = a\}$
(F) $\models$ $\{in + m = 250\}$ $\texttt{while (i} > 0\texttt{) do m} := \texttt{m} + \texttt{n};\ \texttt{i} := \texttt{i} - 1$ $\{in + m = 250\}$

Hint: Eiffel does not offer a while construct. Try experimenting with from-until-loop instead, as well as if-then-else with recursion (note that recursive calls should be surrounded by `wrap` and `unwrap` so that the verifier checks the class invariant—see the code comments).

iii. Consider the class `MAX_IN_ARRAY` in:

http://cloudstudio.ethz.ch/e4pubs/#sv-task3

What does the `max_in_array` method do? Prove the class correct in AutoProof by determining a suitable precondition and loop invariant.

Hint: you might find Eiffel's across-as-all loop construct[2] helpful for expressing loop invariants.

iv. (∗) Consider the class `SUM_AND_MAX` in:

http://cloudstudio.ethz.ch/e4pubs/#sv-task4

What does the method `sum_and_max` do? What can you prove about it using AutoProof?

v. (∗∗) Consider the class `LCP` in:

http://cloudstudio.ethz.ch/e4pubs/#sv-task5

The method `lcp` implements a Longest Common Prefix (LCP) algorithm[3] with input and output as follows:

Input: an integer array a, and two indices x and y into this array. *Output:* length of the longest common prefix of the subarrays of a starting at x and y respectively.

What can you prove about the class in AutoProof?

References

[1] K. Rustan M. Leino. This is Boogie 2. Technical report, 2008. http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf.

[2] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. AutoProof: Auto-active functional verification of object-oriented programs. In *Proc. TACAS 2015*, volume 9035 of *LNCS*, pages 566–580. Springer, 2015. http://se.inf.ethz.ch/people/tschannen/publications/tfnp-tacas15.pdf.

[2] See: http://bertrandmeyer.com/2010/01/26/more-expressive-loops-for-eiffel/
[3] From the FM 2012 verification challenge.