Assignment 6
Logic's Labyrinth: Navigating Decisions

15-414: Bug Catching: Automated Program Verification

Due 23:59pm, Friday, March 29, 2024
75 pts

This assignment is due on the above date and it must be submitted electronically on Gradescope. Please carefully read the policies on collaboration and credit on the course web pages at `http://www.cs.cmu.edu/~15414/assignments.html`.

What To Hand In

You should hand in the following files on Gradescope:

- Submit the file `asst6.zip` to Assignment 6 (Code). You can generate this file by running `make handin`. This will include your solution `ubarray.mlw` and the proof session in `ubarray/`.
- Submit a PDF containing your answers to the written questions to Assignment 6 (Written). You may use the file `asst6-sol.tex` as a template and submit `asst6-sol.pdf`. You can generate this file by running `make sol` (assuming you have pdflatex in your system).

  Make sure your session directories and your PDF solution files are up to date before you create the handin file.

Using LaTeX

We prefer the answer to your written questions to be typeset in LaTeX, but as long as you hand in a readable PDF with your solutions it is not a requirement. We package the assignment source `asst6.tex` and a solution template `asst6-sol.tex` in the handout to get you started on this.

1 Propagations and Conflicts (20 pts)

Considering the following propositional formula in CNF:

$$\begin{array}{ll}
\neg x_1 \lor x_2 \lor \neg x_3 & C_1 \\
\neg x_1 \lor \neg x_2 \lor \neg x_3 & C_2 \\
x_1 \lor \neg x_2 \lor \neg x_3 & C_3 \\
x_1 \lor x_2 \lor \neg x_3 & C_4 \\
\neg x_1 \lor x_2 \lor x_3 & C_5 \\
x_1 \lor x_2 \lor x_3 & C_6 \\
\neg x_1 \lor \neg x_2 \lor x_3 & C_7 \\
x_1 \lor \neg x_2 \lor x_3 & C_8
\end{array}$$

Task 1 (20 pts). Assume that the DPLL algorithm already started and that we know that the formula does not have a solution with x_1 assigned to *true*. Continue the DPLL algorithm with clause learning, starting with the decision that x_1 is assigned *false*, i.e. "Decide $\neg x_1$". The DPLL algorithm should terminate with either *satisfiable* or *unsatisfiable*. Note that if you need to backtrack to x_1 then you would show that the formula is unsatisfiable. You should write down the steps of your evaluation in the following form:

(1) Decide p

(2) Unit propagate q from clause C_2

(3) Decide $\neg r$

(4) Unit propagate s from clause C_1

(5) Conflicted clause C_1

(6) Backtrack to r

(7) Learn conflict clause $C_9 = \neg p \lor r$

(8) Unit propagate r from clause C_9

(9) ...

When reaching a conflict, you must learn a clause and show how you derive it from this conflict by either showing a sequence of resolution steps or drawing the implication graph and a possible separating cut. Please refer to Lecture 16 on ways to generate conflict clauses.

2 Congruence closure (15 pts)

Task 2 (15 pts). Use the congruence closure algorithm for T_E to determine the satisfiability of the following Σ_E-formulae. Please refer to Lecture 17 (page 7) for details on the algorithm.

1. $f(g(x)) = g(f(x)) \land f(g(f(y))) = x \land f(y) = x \land g(f(x)) \neq x$
2. $f(f(f(a))) = f(a) \land f(f(a)) = a \land f(a) \neq a$

3 Unbounded Arrays (40 pts)

In this problem we explore the use of verification to certify resource bounds using amortized analysis. We also revisit data structure invariants, ghosts, and illustrate how to handle data that admit a null value.

An *unbounded array* has constant time set and get operations, just like ordinary arrays, but we can also extend their domain by *adding* elements at the end or shrink their domain by *removing* elements from the end. Moreover, we specify that add and remove should have *constant-time amortized cost*. We fix the cost model by specifying

A write operation to an array requires one unit of work; all other operations are free.

You can find a description of unbounded arrays and their amortized analysis at 11-ubarrays.pdf from *15-122 Principles of Imperative Computation*.

In WhyML we represent an unbounded array with

```
type uba 'a = { mutable size : int ;
                mutable limit : int ;
                mutable data : array (option 'a) ;
                mutable ghost potential : int }
```

The `data` array contains elements `Some x` in its domain but may also have some unused elements. The `size` field represents the current domain of the unbounded array. We can apply `get` and `set` to any index $0 \leq i < \mathsf{size}$. `limit` is the actual length of the underlying `data` array that has been allocated, with elements $\mathsf{size} \leq i < \mathsf{limit}$ being reserved for expansion of the domain.

The ghost field `potential` represents the available potential (or number of "tokens") in the data structure, which must always remain nonnegative. According to our cost model, each token permits one array write operation. Each operation that entails a write, namely `set`, `add`, and `rem` is given a certain amount of potential. Any potential not immediately used is stored in the ghost field for later use. The stored potential is necessary to copy the array content whenever a new underlying data array is allocated, which happens when the domain becomes too large or too small. Further specifics are given with the operations below.

Implement the operations on unbounded arrays according to the following specs.

(i) `new (size : int) (default : 'a) : uba 'a`
Allocate and return a new unbounded array with initial domain size `size` and limit `2*size` (or `1` when `size` is `0`). The array should be initialized with the given default element. Allocation incurs no cost in our model.

(ii) `len (u : uba 'a) : int`
Return the current size of the domain of u.

(iii) `get (u : uba 'a) (i : int) : 'a`
Get the element at index i of u.

(iv) `set (u : uba 'a) (i : int) (x : 'a) (ghost p : int)`
Set the element at index i of u to be x. This requires 1 token which should be passed as p.

(v) `add (u : uba 'a) (x : 'a) (ghost p : int)`
Add a new element at the end of u, expanding it domain. This requires 3 tokens: 1 to perform

the write, and 2 to save. If adding an element fills the array completely, double the length of the array and copy the elements in the domain to the new array.

(vi) `rem (u : uba 'a) (ghost p : int) : 'a`
Remove an element from the end of u, shrinking its domain. This costs 2 tokens: 1 to overwrite the element to be `None` and 1 to save. When the domain is only one fourth of the length of the array or smaller, cut the length of the array in half and copy the elements in the domain to the new array. A precondition should require that `rem` cannot be called on an unbounded array with empty domain.

In order to implement the cost model, every array write operation is followed by a decrement of the potential. Also, each function that receives tokens adds them to the potential of the unbounded array. As an example, we show the (incompletely specified) `set` function.

```
let set (u : uba 'a) (i : int) (x : 'a) (ghost p : int) =
requires { 0 <= i < u.size }                (* i must be in domain *)
requires { p = 1 }                          (* amortized cost of 'set' is 1 *)
...                                         (* more contracts as needed *)
ghost (u.potential <- u.potential + p) ;    (* store received potential *)
u.data[i] <- Some x ;
ghost (u.potential <- u.potential - 1)      (* pay for write *)
```

The data structure invariants are simpler if the array can only be expanded but not shrunk. We therefore recommend the following sequence of tasks, even if you should hand in just one file `ubarray.mlw` at the end.

Task 3 (5 pts). Specify the data structure invariants of unbounded arrays, including the requirement that potential remain nonnegative. For this task, you may restrict attention to domain extension with the `add` operation and ignore `rem`.

Task 4 (25 pts). Implement and verify the `new`, `len`, `get`, `set`, and `add` operations (5 pts each). Verification of contracts should guarantee the intended meaning of each operation and, together with the data structure invariants, the correct amortized cost of each operation.

Task 5 (10 pts). Implement the `rem` operation and update the data structure invariants to guarantee the correctness of the amortized analysis.