VSComp 2014 Problem Set

Ernie Cohen, Marcelo Frias, Peter Müller, Natarajan Shankar

June 14, 2014

The competition takes place over 48 hours during the weekend of 14/15 June starting with the publication of the challenge problems at 0900 GMT of Sat June 14 and ending with the submission deadline for the solutions at 0900 GMT of Mon June 16. Teams should contain no more than three individuals, and you must register to participate at `http://vscomp.org`. The competition consists of the five verification challenges given below. Teams have to solve the challenge problems using mechanized tools. You are allowed to use multiple tools to tackle the challenges. You can upload a *separate* solution to each of the problems you have tackled to `http://vscomp.org` prior to the deadline. Please indicate by email the three problem solutions you wish to have considered for evaluation. It is fine to submit partial solutions. An FAQ page will be maintained at the website to clarify questions that might come up before or during the competition. After the competition, solutions will be made publicly available for peer review by the other teams. Teams should be available to answer questions during the peer review process, and if necessary, be prepared to make their verification tools available for experimentation. Based on the results of the peer review, the organizers will evaluate the solutions for correctness, completeness, and clarity.

The winners will be given short speaking slots at VSTTE 2014 which takes place in Vienna during July 17, 18, 2014. The competition will run on an honor system to ensure that above guidelines are being followed by each of the teams. Please provide helpful and extensive documentation with your solution. Indicate the exact claims that have been successfully verified and list all assumptions (and it's okay to make such assumptions as long as they are explicit).

Questions or comments about the contest should be sent to vscomp2014@gmail.com.

1 Problem 1: Patience Solitaire

Patience Solitaire is played by taking cards one-by-one from a deck of cards and arranging them face up in a sequence of stacks arranged from left to right as follows. The very first card from the deck is kept face up to form a singleton stack. Each subsequent card is placed on the leftmost stack where its card value is no greater than the topmost card on that stack. If there is no such stack, then a new stack is started to right of the other stacks. We can do this with positive numbers instead of cards. If the input sequence is 9, 7, 10, 9, 5, 4, and 10, then the stacks develop as

$\langle[9]\rangle$
$\langle[7,9]\rangle$
$\langle[7,9],[10]\rangle$
$\langle[7,9],[9,10]\rangle$
$\langle[5,7,9],[9,10]\rangle$
$\langle[4,5,7,9],[9,10]\rangle$
$\langle[4,5,7,9],[9,10],\langle 10\rangle\rangle$

Verify the claim is that the number of stacks at the end of the game is the length of the longest (strictly) increasing subsequence in the input sequence. In the above example, the subsequence is $7, 9, 10$, i.e., the second, fourth, and last elements of the sequence. The informal argument is as follows. Let σ be the input sequence, and n be the number of stacks constructed by the Patience solitaire algorithm. One element $\sigma(i)$ precedes another $\sigma(j)$ in the sequence σ if $i < j$ and $\sigma(i) < \sigma(j)$. An increasing subsequence is given by a sequence of indices $i_1, \ldots, i_n$ where $\sigma(i_j)$ precedes $\sigma(i_{j+1})$.

1. To each element in stack $i+1$, there is an element in stack i that precedes it, so that by chaining these elements, we can construct an increasing subsequence of length n.

2. If there is a longer increasing subsequence, by the Pigeonhole Principle, it must contain two sequence elements $\sigma(i)$ and $\sigma(j)$ from the same stack, but such elements cannot precede one another.

2 Problem 2: Partition Refinement

A set S can be represented by an array A of distinct elements. A partition P of S splits S into a disjoint subsets $S_0, \ldots, S_{m-1}$. By rearranging the array elements in A, we can maintain each S_i as a set of contiguous elements of the array A, where each subset S_i is represented by a structure p_i that maintains *first* and *last* index in A for S_i. Now, given a set X, we want to refine the partition $S_0, \ldots, S_{m-1}$ so that we refine each S_i into two sets $S_i \cap X$ and $S_i \setminus X$. Your program should take the set S represented by A, the partition $S_0, \ldots, S_{m-1}$ represented as indicated below, and the set X, and return a representation of a new partition of S that refines each S_i into one or two nonempty subsets representing $S_i \cap X$ and $S_i \setminus X$.

The program can also be written with pointers, and participants are welcome to try to program it with pointers. The state of the algorithm consists of

1. The array A of size N represents a set S of positive integers. The elements in A can be rearranged as long as the set of integers is unchanged.

2. The input partitioning set X is given as a list of positive integers.

3. The partial map D represents an "inverse" of A so that $D(A(i)) = i$ for $0 \leq i < N$. It is needed to determine the position in array A of a given element of the S.

4. The partition P is an aggregate of intervals, e.g., a sequence, where each interval p contains fields *first* and *last* representing a contiguous segment of the array A. Each such partition p represents the subset $[\![p]\!]_A$ defined as $\{x \in S | \exists i : p.first \leq i \leq p.last : A(i) = x\}$. The algorithm also adds a field *p.count* (with a default value of 0) that is used in implementing the refinement algorithm.

5. A map F from the indices of A to partitions in P capturing the relationship that index i is in partition $p = P(F(i))$, i.e., $p.first \leq i \leq p.last$.

The algorithm refines the partition P by processing each element x of X. A putative design could be given in terms of procedures that operate on the data structures A, D, P, and F.

1. *refine*(A, D, P, F, X): In the first phase, successively invoke *refineOne*(A, D, P, F, x) over each x in X. In the second phase, apply *makeNewPartitions*(P, F) to the structures P and F returned by the first phase to spawn new intervals from the ones that have been refined.

2. *refineOne*(A, D, P, F, x): Let i be $D(x)$ and p be the interval $F(i)$. If $i < p.first + p.count$ then x has already been processed so the state is left unchanged. Otherwise, swap $A(i)$ and $A(p.first + p.count)$, set $D(A(i))$ to i and $D(A(p.first + p.count))$ to $p.first + p.count$, and increment *p.count* by one.

3. *makeNewPartitions*(P, F): For each p, j such that $p = P(j)$ and $0 < p.count < p.last - p.first + 1$, set *p.last* to $p.first + p.count - 1$, and add a new interval p' to P with $p'.first = p.count$ and $p'.last = p.last$, and set *p.count* and $p'.count$ both to 0. For each i from $p'.first$ to $p'.last$, set $F(i)$ to the index of p' in P.

For example, if

1. A is $\langle 0 \mapsto 22, 1 \mapsto 33, 2 \mapsto 100, 3 \mapsto 55, 4 \mapsto 44, 5 \mapsto 11 \rangle$

2. D is the inverse of A and is $\langle 100 \mapsto 2, 11 \mapsto 5, 22 \mapsto 0, 33 \mapsto 1, 44 \mapsto 4, 55 \mapsto 3 \rangle$

3. P is $\langle 0 \mapsto \langle \mathit{first} \mapsto 0, \mathit{last} \mapsto 2 \rangle, 1 \mapsto \langle \mathit{first} \mapsto 3, \mathit{last} \mapsto 5 \rangle \rangle$

4. F is $\langle 0 \mapsto 0, 1 \mapsto 0, 2 \mapsto 0, 3 \mapsto 1, 4 \mapsto 1, 5 \mapsto 1 \rangle$

5. X, the input, is $\{11, 22, 44\}$.

Processing each element of the input X yields a state where A is

$$\langle 0 \mapsto 2, 1 \mapsto 1, 2 \mapsto 5, 3 \mapsto 3, 4 \mapsto 0, 5 \mapsto 4 \rangle,$$

D is

$$\langle 100 \mapsto 4, 11 \mapsto 1, 22 \mapsto 0, 33 \mapsto 3, 44 \mapsto 5, 55 \mapsto 2 \rangle,$$

P is

$$\left\langle \begin{array}{l} 0 \mapsto \langle \mathit{first} \mapsto 0, \mathit{last} \mapsto 0 \rangle, \\ 1 \mapsto \langle \mathit{first} \mapsto 1, \mathit{last} \mapsto 2 \rangle, \\ 2 \mapsto \langle \mathit{first} \mapsto 3, \mathit{last} \mapsto 4 \rangle, \\ 3 \mapsto \langle \mathit{first} \mapsto 5, \mathit{last} \mapsto 5 \rangle \end{array} \right\rangle,$$

and F is

$$\langle 0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 2, 4 \mapsto 2, 5 \mapsto 3 \rangle.$$

Verification Tasks. The tasks are to show that

1. The algorithm always terminates returning an output state A', D', P', F'.
2. The output partition given by A', D', P', F' is a refinement of the input partition A, D, P, F so that
   (a) For any $i < N$, $P'(F(i))$ is an interval containing i.
   (b) For any p' in P', there exists a p in P such that $[\![p']\!]_{A'} = [\![p]\!]_A \cap X$ or $[\![p']\!]_{A'} = [\![p]\!]_A \setminus X$

3 Problem 3: A Lock-free Log-based Set Algorithm

Consider a system with `ThreadCount` threads, implementing a lock-free atomic set of ints as follows. The shared state of the system is given by the following declarations, where you should think of `size` as infinite:

```
atomic<int> log[size]; // log of operations on the set; initially all zeroes

atomic<size_t> gc, tl, hd; // initially all zero

atomic<size_t> ht[threadCnt]; // initially all size
```

The meaning of the state is approximately as follows:

- The `log` represents the history of operations on a set. A zero entry represents an unused entry. A positive entry represents the addition of its value to the set. A negative entry represents removal of its negation from the set. The latest entry for a particular (positive) int gives its current set membership.
- Log entries at or above `hd` are not yet "visible" to readers of the set.
- Log entries with indices below `gc` have been freed, so reading or writing these locations is considered an error.
- Log entries with indices below `tl` are redundant, i.e. each of them is followed by a later entry of the same value or its negation.
- Thread `t` writes `idx` into `ht[t]` to prevent the garbage collector from moving beyond `idx`.

Threads can call the functions update, lookup, and collect below. In the code, you can either assume `hd < size` or specifically prevent `hd` from changing when `hd + 1 == size`.

```
#define doReturn(v) { ht[me()] = size; return v; }

#define advance(local,var) { local = var; ht[me()] = local;}

#define grab(local, var) { advance(local,var); advance(local,var); }


void update(int val) { // atomically add or remove abs(val) from the set

size_t h;

grab(h, hd)

while (true) {

bool success = !cmpXchg<int>(log+h, 0, val);

cmpXchg<size_t>(&hd, h, h+1);

if (success) doReturn();
```

```
advance(h,hd);
}
}

bool lookup(int val) { // atomically check if positive int val is in the set
size_t t,i;
int x;
grab(t,tl);
for (i = hd; i != t && abs(x = h[i-1]) != val; i--);
doReturn(i != t && 0 < x);
}

void collect() { // try to collect garbage; semantic no-op
size_t t = tl;
for (size_t i = 0; i < threadCnt; i++) t = min(t,ht[i]);
size_t g = gc;
if (g < t) cmpXchg<size_t>(&gc,g,t);
}
```

In addition, `tl` can be updated atomically (by the environment) with the following atomic action:

```
<if (exists size_t i: tl < i < hd && (abs(log[tl]) == abs(log[i]))) tl++ >
```

Notes:

- The form of the code is motivated by brevity; feel free to restructure/rewrite it as long as you don't change its essential behavior.
- Reading or writing an `atomic<T>` is considered an atomic action.
- You can assume sequential consistency (i.e., the usual interleaving model of concurrency).
- The function `me()` returns the id of the executing thread (`0 <= me() < threadCnt`).
- The function `cmpXchg<T>(T* loc, T old, T new)` behaves as follows, but executes atomically:

```
T cmpXchg<T>(T* loc, T old, T new) {

if (*loc == old) { *loc = new; return old; }

return *loc;

}
```

A complete solution should do at least the following:

- Prove the program is memory-safe, i.e., no thread accesses `log` entries below `gc`.
- Define, for every state, the abstract state of the set in that state. (You can do this using ghost variables or using an abstraction map, or any other reasonable method.) This abstract value should be consistent with the obvious intentions of the problem.
- Prove that lookup is linearizable, i.e. that if it returns true/false, `val` was/(was not) in the abstract set at some point between when the call started and when it returned.

Notes:

- If you are using a system that provides concurrency only through locking, you can treat each access to an `atomic<T>` as protected by a lock. Alternatively, you may treat the entire shared state as protected by a single global lock.
- You can also choose to treat the program as a transition system and reformulate the requirements appropriately.
- If your system requires functions to terminate, you can modify `update()` to give up after some number of unsuccessful attempts and return an error indication.
- You can formulate linearizability in any reasonable way; ideally, you should use a method that allows a linearizable function call to be used in a larger proof in the same way that you would use an atomic action.

In addition, for your own amusement (but not graded as part of the contest), you can consider the following additional challenges (perhaps after the contest ends):

- Prove that update is linearizable, i.e. that its effect on the abstract value of the set happens atomically at some point between when the call starts and when it returns. Note that you should also prove (perhaps indirectly) that all updates to the abstract value of the set can be mapped bijectively to calls of update.
- Write a function that repeatedly compresses the log by reading a batch of entries from the beginning of the log, adding to the end of the log those entries from the batch that do not occur later in the log, and advancing tl past the batch. (This function should be a semantic no-op.)
- Modify the program to implement a mapping from nonzero int to nonzero int by having each log entry contain two atomic ints, a key and a def, only one of which can be read, written, or cmpXchg'd in a single atomic action. Your solution should still be lock-free, i.e. if some threads stop working, the remaining threads should continue to make progress (except that garbage collection might stall). You are not allowed to allocate memory on the heap, but you are allowed to have additional shared data structures (of size proportionate to the number of threads). (This is a fun but not too hard problem if you like concurrent programming.) (Hint: for each thread, introduce a shared data structure used by the thread to announce his intention to fill in a particular slot in a particular way.) And, of course, verify the resulting program.

- (Only for those into C/C++11 weak memory) Try to weaken the atomic operations appropriately (to acquire/release/consume/relaxed) wherever possible.

4 Problem 4: Graph Cloning

Consider the following representation of a graph G. The nodes of G are heap-allocated objects such as instances of a class or struct. Each node stores a natural number. The edges of G are labeled with integer values, where all edges with the same source node have distinct labels. Each node contains a map from labels to references (pointers) to nodes. G contains an edge from n_1 to n_2 with label l if and only if n_1's map maps l to n_2. In Java, this graph representation could be implemented as follows:

```
public class Node {
    Map<Integer, Node> edges;
    int value;  // a natural number

    public Node(int value) {
        edges = new HashMap<Integer, Node>();
        this.value = value;
    }

    ...
}
```

Note that the problem focuses on the verification of heap-manipulating programs. Therefore, it is essential that your implementation represents nodes as heap-allocated objects and edges as references (pointers).

Task 1: Implement a copy method that takes a node and copies the graph structure reachable from that node. A possible implementation in Java looks as follows (in class Node):

```
public Node copy() {
    return doCopy(new HashMap<Node, Node>()).c;
}

Result doCopy(Map<Node, Node> m) {
    if (m.containsKey(this)) {
        return new Result(m.get(this), m);
    } else {
        Node tmp = new Node(value);
        m.put(this, tmp);
        Set<Integer> ids = edges.keySet();
        for(Integer id: ids) {
            Result r = edges.get(id).doCopy(m);
            tmp.edges.put(id, r.c);
            m = r.m;
        }
        return new Result(tmp, m);
    }
}
```

where Result is an auxiliary class that allows method doCopy to return two results:

```
class Result {
    Node c;
    Map<Node, Node> m;

    Result(Node c, Map<Node, Node> m) {
        this.c = c;
```

```
        this.m = m;
    }
}
```

You should prove the following properties of your copy method:

1. Provided that the argument node is non-null, the method will not cause a run-time error except possibly errors that are not under the control of your program (such as memory errors or virtual machine errors). The exact set of run-time errors that must be shown to not occur depends on your programming language and implementation. In the above Java code, it would include null-pointer dereferencing and precondition violation for any of the called methods.
2. The method terminates for all non-null inputs.
3. The result of the method and all objects reachable from it are fresh, that is, are allocated and have not been allocated in the pre-state of the method.
4. The method has no observable side effects, that is, does not modify or de-allocate any object that was allocated in the pre-state. If the verifier has no direct support for specifying and verifying effects, one could prove that any well-defined function that traverses the heap starting from an allocated object yields the same result if applied in the heaps before and after a call to the copy method.
5. For any valid call to n.copy(), the subgraph reachable from the n before the call is isomorphic to the subgraph reachable from the result after the call.

Task 2: Implement an eval method that takes a non-null node and a finite sequence of integers (labels) and returns an integer. From the given node, the method traverses the graph by following edges according to the labels in the sequence. If there are suitable edges for all labels in the sequence, eval returns the natural number stored in the final node; otherwise, it returns -1. A possible implementation in Java is as follows (in class Node):

```
public int eval(List<Integer> path) {
    Node ptr = this;
    for(Integer id : path) {
        ptr = ptr.edges.get(id);
        if(ptr == null) {
            return -1;
        }
    }
    return ptr.value;
}
```

You should prove the following properties of your eval method:

1. Provided that the arguments are non-null, the method will not cause a run-time error (as defined in the previous task).
2. The method terminates for all non-null inputs (assuming the label sequence is finite).
3. The method satisfies the functional specification given above.
4. For all non-null nodes n and sequences path, n.eval(path) and n.copy().eval(path) yield the same result.

5 Problem 5: Binomial Heaps

We present a compilable excerpt from classes `BinomialHeap` and `BinomialHeapNode` modeling binomial heaps [1]. Method `extractMin` has a fault that allows for the class invariant to be violated. Your goals towards solving this problem are:

1. Find and describe the above-mentioned fault.
2. Provide an input binomial heap that exercises this fault, and describe the technique used to find the input (automated techniques are preferred to human inspection).
3. Provide a correct version of method `extractMin`.
4. Provide a suitable class invariant.
5. Verify that method `extractMin` indeed preserves the provided class invariant or at least a meaningful subset of properties from the class invariant.

Hints can be obtained from the organizers in exchange for penalities in the final score.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*, MIT-Press.