Program Verification

Assignment 2

Niek Haarman, Tanja de Jong, Tinus Pool

July 3, 2015

For this assignment we specified and verified a sequential and concurrent `Queue` implementation. Below is a report about our findings.

1 Sequential Queue implementation

For the sequential Queue, we decided to go with a `LinkedList` implementation, which can be seen as a FIFO queue where items are appended at the tail, and taken from the head of the queue. We took the Java implementation of `LinkedList` and stripped it down to contain only code relevant to the exercise, and created a PVL implementation based on that.

The Java implementation keeps a reference to the first and last `Node` of the list as fields. This allows for easy appending and extraction of values. We chose to create a `contents()` function which describes the current contents of the list as a `seq<int>`. That way we can easily ensure items are added and removed properly. To do that, we declared a recursive `contents()` function on `Node`. This approach requires that the first `Node` has (recursively) at least a read permission on the `val` and `next` fields of all subsequent `Nodes`, as seen in Listing 1. This immediately leads to a problem, since we cannot easily obtain a write permission to append an item to the queue: either we have full write permissions on the contents of the `LinkedList`'s `last` field, or we have recursive read permissions for all `Nodes`. To solve this, we dropped the `last` field as a whole, and gave full recursive *write* permission to the `Nodes`. Appending an item now traverses the entire list, starting from the head. The up-to-date and working version can be found in the attachments.

We had to specify the methods `peek`, `poll` and `offer`. The latter two delegate their work to the 'private' functions `unlinkFirst()` and `linkLast(int)` respectively. Since we used the `contents()` function in our specifications, we can easily verify and test that our implementation works as intended.

To verify the sequential implementation, run the following command:

```
vct Integer.pvl Node.pvl Queue.pvl Test.pvl --silver=silicon --inline
```

```
class Node {
  Node next;
  int val;

  resource state() = Value(val) ** Value(next) ** next->state();

  requires state();
  seq<int> contents() =
  unfolding state() in (
    next == null
     ? seq<int>{val}
     : seq<int>{val} + next.contents()
  );
}
```

Listing 1: Basic Node specification

2 Concurrent Queue implementation

For the concurrent Queue, the assignment specified to use the `LinkedBlockingQueue`. Again, we took the Java implementation and created a PVL implementation of the required methods.

A big difference between the implementation of `LinkedBlockedQueue` and `LinkedList`, which we used for the sequential queue, is that the constructor of the `LinkedBlockedQueue` constructs a '`null Node`', so that the queue is never actually empty, while the `LinkedList` simply starts with an empty list. At our first try, we did not understand why this approach was taken, so we decided to implement this part similarly to the `LinkedList`, by constructing the Queue without any nodes. However, during the process of verifying the `LinkedBlockedQueue`, we realized that this resulted in a problem for a concurrent queue if one thread attempts to put a value in the queue, while an another thread tries to read a value from that queue at the same time, when there is exactly one element in the queue.

The main problem we encountered while verifying the concurrent implementation was the following: How can you build in two locks in a queue that refer to two different parts of the queue? This problem is created because of the restriction that the queue can contain only one lock_invariant. At first, we tried to solve this problem by using kind of symbolic locks. This way, nothing was actually locked, but because the implementation would be consistent in the usage and basically have barrier against violating what we would actually wanting to lock, it was a good start and in our opinion valid for the moment. Using this approach, the `takeLock` prohibited parallel consuming of items, and `putLock` ensured no two threads could append at the same time.

In this way, we were able to write all the method implementations. However, for inserting and removing objects, we used the same method as we used in the sequential

```
class Node {
  Node next;
  Integer val;

  resource lock_invariant() = Perm(val, 1) ** val->state()
    ** Perm(next, 1);

  requires val != null ==> Value(val.val);
  Node(Integer val) {
    this.val = val;
    this.next = null;
  }
}
```

Listing 2: Node implementation using a `lock_invariant`

implementation, which required to have full permission over the queue and its node. So that implementation is like one stick or chain that is only allowed to be held by one hand. This is a problem for concurrent programs, where you have to be able to hold the same side of the stick/chain and be able to hold it with multiple hands. So, this implementation did not allow for concurrent access to the queue.

Instead, we dropped the recursive `state` predicate of `Node`, and exchanged it with a lock_invariant which releases full write permissions on its fields when lock upon, as seen in Listing 2. The lock_invariant will preserve the state, so you don't have to save that state by having to remember the next node's state. This enabled us to introduce the head and last node again. The queue became a thing where we only want to ensure that it has a value. We don't need write permission of the queue itself to change elements. We changed the symbolic locks to elements that do actually know the queue instance. One of the locks will lock the first element of the queue, which prevents other threads from removing elements from the queue. The other lock will lock the last element of the queue, which prevents other threads from adding elements to the queue.

3 Other problems encountered:

- We could not test our implementation with the Java compiler, since the `seq<int>` conversion didn't work correctly.
- Need a `lockAndUnlock()` action for some methods, to create a kind of atomic `get()`. In our atomic integer we where not allowed to use our `get()` in verifying our code, because a normal method might change code.
- Error messages for not catching return values and using reserved variables can be confusing and hard to discover if you write a lot of code in a flow.

- Assigning null to a return or variable was troubling.

It is possible that along the way some of the problems mentioned above where solved.

4 Possibilities for adding functional properties

In the current specification for the concurrent implementation, it is not possible to ensure the state of the queue after any method, because no assumptions can be made about other threads that might have changed the queue after the lock has been released. For example, after the `take()` method is called, it can not be ensured that the contents of the queue are equal to the contents of the queue, before the method was called, without the first element.

However, it is possible to use so-called *history* to be able to ensure more properties of the methods. By doing this, the history keeps track of all elements that have been added to or removed from the queue. For example, if we would add this specification to the `put()` and `take()` methods, we will know that a value has been respectively added to or removed from the queue. Then, if multiple threads are simultaneously calling the `put` method, we can use the history to ensure that both elements have been added to the queue. However, we cannot ensure the order in which this was done. We can draw the same conclusion when multiple threads call the `take()` method, but instead of ensuring that elements were added, we can ensure that these exact elements were taken from the queue.

So, by added information about the history of the queue, we can draw more conclusions about the contents of a concurrent queue after an element has been added to or removed from that queue. An interesting case which could benefit from this would be some sort of a job queue where multiple threads produce jobs to be executed, and consumer threads consume the jobs in the queue.

5 Work distribution

Because the bulk of the work revolved around figuring out a correct approach for the verification of the sequential and concurrent queues, the main activity for this assignment was discussing the possibilities. Therefore, nearly everything was done as a group and there was not actually much distribution of work. When not working together, we would each fiddle with the code trying out different ways, which would then be shared with the group to benefit from that.