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 a 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.

While working at the sequential exercises we did want to hold the permission stick on both ends. The problem however is that you can only release one end of the stick at a time, while you have to release both ends to make the stick float in the air. Only when the stick floats in the air you can adjust it.

This is a problem we encountered in our first try with the queue. We had a node last and first and the resource state required the next state of the node. This way we created a chain from the first till the last. So if you wanted to edit the last you where not allowed because last permission was also locked by the chain created by implied by the first state.

```
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

This was easily solved by just removing the last, and following the chain till the end to change the last element of the queue. An other possible option was creating all the nodes in a factory and watch the permissions in this factory. However because this concept had some troubles implementing at first and the other implementation was faster implemented, we decided to not use a factory but remove the last node. We still could argue what solution would be better because they both have there nice things and ugly things. Another reason why the factory was not added in the end is because Vercors can be a bit fragile in times. We did not want to burn our fingers when there is already something workable.