Exam Questions
Proof Methods for Concurrent Programs

10 March 2011

Instructions: only printed documents are authorised. You can admit the result of one question and move on. Leave optional questions until the end.

Our goal is to implement a *concurrent queue* and prove the implementation correct. Our design will have amortised $O(1)$ access time, and enable a `push` and a `pop` to be performed concurrently most of the time.

Exercise 1. (Basic data structures)

Remember that the specification of the *list* data structure is

$$\begin{aligned} \mathrm{ls}\ \epsilon\ (\mathtt{f}) &= \mathit{empty}\ \wedge\ \mathtt{f} = \mathtt{null} \\ \mathrm{ls}\ (\mathtt{a}\cdot\alpha)\ (\mathtt{f}) &= \exists \mathtt{j}.\, \mathtt{f} \mapsto \mathtt{a}, \mathtt{j}\ *\ \mathrm{ls}\ \alpha\ \mathtt{j} \end{aligned}$$

1. Implement a `add(f,a)` function that adds the element `a` in front of the list pointed by `f`. Prove that your implementation satisfies the specification:

   $$\{\ \mathrm{ls}\ \alpha\ \mathtt{f}\ \}\ \mathtt{add(f,a)}\ \{\ \mathrm{ls}\ (\mathtt{a}\cdot\alpha)\ \mathtt{f}\ \}$$

2. Implement a `remove(f)` function that satisfies the specification:

   $$\begin{aligned} \{\ \mathrm{ls}\ (\mathtt{a}\cdot\alpha)\ \mathtt{f}\ \} &\quad \mathtt{remove(f)} \quad \{\ \mathrm{ls}\ \alpha\ \mathtt{f}\ \wedge\ \mathtt{r} = \mathtt{a}\ \} \\ \{\ \mathrm{ls}\ \epsilon\ \mathtt{f}\ \} &\quad \mathtt{remove(f)} \quad \{\ \mathit{false}\ \} \end{aligned}$$

   and prove it correct. The variable `r` is used to pass the return value of all the functions. *Optional:* similarly, implement and prove correct the functions `is_empty(f)` and `del(f)` specified as

   $$\begin{aligned} \{\ \mathrm{ls}\ (\mathtt{a}\cdot\alpha)\ \mathtt{f}\ \} &\quad \mathtt{is\_empty(f)} \quad \{\ \mathrm{ls}\ (\mathtt{a}\cdot\alpha)\ \mathtt{f}\ \wedge\ \mathtt{r} = \mathtt{false}\ \} \\ \{\ \mathrm{ls}\ \epsilon\ \mathtt{f}\ \} &\quad \mathtt{is\_empty(f)} \quad \{\ \mathrm{ls}\ \epsilon\ \mathtt{f}\ \wedge\ \mathtt{r} = \mathtt{true}\ \} \end{aligned} \qquad \{\ \mathrm{ls}\ \alpha\ \mathtt{f}\ \}\ \mathtt{del(f)}\ \{\ \mathit{empty}\ \}$$

3. Implement a `reverse(f)` function that returns a pointer to a list that contains all the elements of the list pointed by `f` in reversed order, according to the specification:

   $$\{\ \mathrm{ls}\ \alpha\ \mathtt{f}\ \}\ \mathtt{reverse(f)}\ \{\ \mathrm{ls}\ \alpha\ \mathtt{f}\ *\ \mathrm{ls}\ \overline{\alpha}\ \mathtt{r}\ \}$$

   where $\overline{\epsilon} = \epsilon$ and $\overline{a\cdot\alpha} = \overline{\alpha}\cdot a$. Observe that the original list is untouched.[1]

Exercise 2. (A sequential queue)

We implement a *queue* using two lists, pointed to by (and called) `front` and `back`. The two lists are initially empty. The semantics of `push` and `pop` (which should not leak memory) is described below:

- the `push(a)` function always puts the element `a` in front of list `front`.
- if the list `back` is non-empty, then the `pop()` function removes an element from the front of the list `back`.
- if the list `back` is empty, then the `pop()` function performs the following actions: `back` is updated to point to a list containing the elements of `front` in reverse order; `front` is updated to point to the empty list; and the front element of the list pointed by `back` is removed from the list and returned.

[1] Remark: a clever programmer would implement in-place list reverse here. However, for the sake of the exam questions, we stick to this copy and reverse semantics specified above.

We define the predicate `queue` as:

$$\texttt{queue}\ \alpha \quad = \quad \exists\beta.\exists\gamma.\ \text{ls}\ \beta\ \texttt{front}\ *\ \text{ls}\ \gamma\ \texttt{back}\ \wedge\ \alpha = \beta\cdot\overline{\gamma}$$

4. Using the functions defined in Exercise 1., implement the `push(a)` function and prove that it satisfies the specification:

$$\{\ \texttt{queue}\ \alpha\ \}\ \texttt{push(a)}\ \{\ \texttt{queue}\ (\texttt{a}\cdot\alpha)\ \}$$

5. Using the functions defined in Exercise 1., implement the `pop()` function and prove that it satisfies the specification:

$$\{\ \texttt{queue}\ (\texttt{a}\cdot\alpha)\ \}\quad \texttt{pop()}\quad \{\ \texttt{queue}\ \alpha\ \wedge\ \texttt{r} = \texttt{a}\ \}$$
$$\{\ \texttt{queue}\ \epsilon\ \}\quad \texttt{pop()}\quad \{\ \texttt{queue}\ \epsilon\ \wedge\ \texttt{r} = \texttt{null}\ \}$$

Exercise 3. (A concurrent queue)

6. Show that the implementation of `push` and `pop` done in questions [4.] and [5.] is not thread safe.

7. By using two resources `f` and `b` that protect respectively the list pointed by `front` and `back`, implement a thread-safe version of `push` and `pop`. This implementation should allow simultaneous executions of `push` and `pop` (unless the list pointed by `back` is empty).

8. Which are the resource invariants $R_{\texttt{f}}$ and $R_{\texttt{b}}$ associated to `f` and `b`? Assuming $R_{\texttt{f}}$ and $R_{\texttt{b}}$ prove that:

$$\{\textit{empty}\}\ \texttt{push(a)}\ \{\textit{empty}\} \qquad \{\textit{empty}\}\ \texttt{pop()}\ \{\textit{empty}\}$$

9. Can a system that invokes `push` and `pop` deadlock? If yes, show how; if not, explain informally why it can not.