IMP Arrays

CMSC 631 Final Project

Cameron Bernhardt

Fall 2019

1 Proposal

For this project, I proposed adding arrays to the IMP language developed in class, with functionality for accessing and mutating elements of the list. After this addition, I intended to implement a simple sort (insertion sort) in IMP with this array construct and prove its correctness using Hoare triples.

2 Source Code

This project was implemented with Coq and the source code can be found here. The additions made for this project are found in the Imp.v file.

3 Implementation

I ended up implementing the array for this language as a single global array for simplicity, as planned. The array can be accessed using the following IMP syntax:

- `HD`: retrieve the element at the front of the array
- `IND a`: retrieve the element at index `a` in the array (`a` is evaluated into a `nat` before retrieval)
- `LEN`: returns the current length of the array

To mutate the array, the following commands were added to IMP as well:

- `CONS a`: add an element `a` of type `aexp` to the front of the array (`a` is evaluated into a `nat` before being added)
- `SET a1 TO a2`: sets the element at index `n1` to the value `n2`, where `n1` and `n2` are the results of evaluating `aexp`s `a1` and `a2`, respectively.

The global array was introduced by modifying the state of IMP programs to include an additional component. Evaluation of a command `c` now takes the form `m =[c]=> m'`, where `m` and `m'` are of type `mem`:

```
Definition array := list nat.
Definition mem : Type := state * array.
```

If the evaluation of the command modifies the variables or array of the program's state, it can pull out the appropriate component of the `m` pair. Initially I proposed only allowing `ANum`s to be added to the program's array, but it turned out to be easier to generalize the array commands to operate on `aexp`s and evaluate them to `nat`s. The array access expressions (`HD`, `IND`, and `LEN`) are also implemented as `aexp`s and can be used anywhere that an `aexp` is expected

in IMP. This allows for these definitions to be used conveniently within IMP code to create iterative procedures:

```
Definition repeat : com :=
    WHILE ~(Z = 0) DO
        CONS X;;
        Z ::= Z - 1
    END

Definition decrement : com :=
    I = 0;;
    WHILE (I < LEN) DO
        SET I TO ((IND I) - 1);;
        I ::= I + 1
    END
```

There are examples of each of the new syntax pieces included throughout the Coq file that briefly demonstrate their behavior (with proofs).

Additionally, a Hoare rule was added for the `SET` command along the lines of the other rules for IMP commands (particularly `hoare_asgn`, which is its state-based counterpart). Similarly, the `set_sub` function mirrors `assn_sub`.

4 Differences from Proposal

The goal of this project was to implement insertion sort using the enhanced IMP described above, but unfortunately this ended up not being a practical scope for the project. Two implementations of insertion sort are included in the Coq file, as well as an unfinished proof of one of them. The definition that this proof is based on is a standard imperative insertion sort program, which while simple to write in IMP is difficult to prove without more machinery around the inductive definition of a sorted list used for the Hoare triples.

I came up with a reworked definition based on inserting into a sorted list and preserving the sorted property, but it needed several lemmas to prove that I didn't have time to implement. The file also contains a program to find the minimum element in an array, but this one proved too complex under my existing definition to complete in time.

I settled on a swap program to implement and prove instead, and this one ended up being the most practical. The proof of this program is called `swap_correct`, and it uses a Hoare triple to describe how the `swap` program will swap two values at indices `I` and `J` in the program's global array. Because I didn't design the IMP array definitions for this use case, I ran into problems near the end of the proof when I tried to prove that the result of looking up the new indices on the array with the swapped values would actually evaluate to the expected values.

The remaining subgoals of the proof made intuitive sense, but due to the

method I used to update values in the array for the evaluation of the `SET` command, it couldn't be reduced directly and instead needed two additional reduction lemmas (`repl_and_retrieve` and `redundant_add`). These lemmas make sense informally: replacing an element at an index in an array and immediately retrieving the value at that index should return the same value as was set, and retrieving a value at an index from an array that was updated at a different index should return the same value that it was set to. However, I left them admitted because I believe proving them formally would be difficult under the Fixpoint definition I used for updating values of the array (defined as `repl`).

5 Takeaways

Doing this project again, I would definitely reconsider the way I implemented the `SET` command and potentially the entire array state itself, as this presented difficulties in the insertion sort and swap proofs. In particular, a definition closer to the functional one of the state would have been helpful with reducing some of the more complex expressions in the proofs, and would have negated the need for the auxiliary lemmas seen towards the end of `swap_correct` and discussed above.

I would also have gone with the insert-based definition of insertion sort first, because if I'd had time to implement the requisite lemmas, it would have been a much more straightforward proof. If I went down this path, I would have written a lemma to demonstrate that splitting a sorted list in two resulted in two sorted lists, which would have been useful combined with my insert procedure, which inserted an element by shifting elements one-by-one until it found the correct location in the sorted list to place the new element. Once I proved that insertion maintained the sorted property, proving insertion sort itself would have been relatively simple (as seen in the Software Foundations chapter).

I would have also spent a lot more time on the Hoare triple-related definitions, making sure that the pieces I set up would hold up in more complex situations, as the trivial examples I used to test them weren't a good representation of the way that I actually used them in my proofs.