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.

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 idiomatic iterative IMP code to be written fairly easily:

```
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.

4 Takeaways