Programming Methodology

Lab 13: Final Review

Wednesday, May 17th, 2023

Weekly Lab Agenda

- Go over reminders/goals
- Review past material
- Work in groups of 2-3 to solve a few exercises
    - Lab leaders will assign new groups this week
- Discussion leaders will walk around and answer questions
- Solutions to exercises will be reviewed as a class
- Attendance taken at the end

Reminders

- Great job this semester everyone, you should be proud of your hard work!
- Max is hosting a final exam review session Sunday December 10 from 7-9pm room TBA, we'll talk about last semester's final exam!
- Office hours will continue to happen as scheduled.
  - Exceptions will be announced on campuswire.
- Exam Logistics:
  - Friday Dec. 15th 3:30pm - 5:50pm in Totman Gym.
  - Make sure to check SPIRE on exam day in case there is a last minute location change!
- Please fill out the SRTI course survey, this really helps make the class better!

Today's Goals

- Practice working with program correctness
- Practice working with asynchronous programming

Exercise: Program Correctness

The following code should partition the given array in-place such that all odd numbers come before all even numbers.

First, write the invariants which satisfy the high-level algorithm.

Then, fill in the code to satisfy the invariants.

```
function partition_even_odd(arr) {
  if (arr.length === 0) { return; }
  let low = ???;
  let high = ???;
  // low/high form a window, the outside of which is partitioned;
  // the window shrinks iteratively until everything is partitioned
  while (???) {
    if (???) {
      // swap arr[low] and arr[high]
      ???
    }
    if (???) {
      // update low
      ???
    }
    if (???) {
      // update high
      ???
    }
  }
}
```

```
function partition_even_odd(arr) {
  if (arr.length === 0) { return; }
  let low = ???;
  let high = ???;
  while (???) {
    // low/high form a window, the outside of which is partitioned
    // => (everything before low is odd) and (everything above high is even)
    // => (forall i: (i < low) => (arr[i] is odd)) and (forall i: (i > high) => (arr[i] is even))
    if (???) {
      // swap arr[low] and arr[high]
      ???
    }
    if (???) {
      // update low
      ???
    }
    if (???) {
      // update high
      ???
    }
  }
}
```

write a loop invariant; we can take our intuitive understanding the algorithm, write it down in plain english, then iteratively refine it until it is more mathematically formal

```
function partition_even_odd(arr) {
  if (arr.length === 0) { return; }
  let low = ???;
  let high = ???;
  // let P1 be (forall i: (i < low) => (arr[i] is odd)) and (forall i: (i > high) => (arr[i] is even))
  // P1
  while (???) {
    // P1
    if (???) {
      // P1
      // swap arr[low] and arr[high]
      ???
      // P1
    }
    // P1
    if (???) {
      // P1
      // update low
      ???
      // P1
    }
    // P1
    if (???) {
      // P1
      // update high
      ???
      // P1
    }
    // P1
  }
  // P1
}
```

fill in all invariants that are directly implied via control flow

```
function partition_even_odd(arr) {
  if (arr.length === 0) { return; }
  let low = 0;
  let high = arr.length - 1;
  // let P1 be (forall i: (i < low) => (arr[i] is odd)) and (forall i: (i > high) => (arr[i] is even))
  // P1
  while (???) {
    // P1
    if (???) {
      // P1
      let t = arr[low]; arr[low] = arr[high]; arr[high] = t;
      // P1
    }
    // P1
    if (arr[low] % 2 === 1) {
      // P1 and (arr[low] is odd)
      // note that (forall i: (i < low) => (arr[i] is odd)) and (arr[low] is odd)
      //           => (forall i: (i < low+1) => (arr[i] is odd))
      low += 1;
      // P1
    }
    // P1
    if (arr[high] % 2 === 0) {
      // P1 and (arr[high] is even)
      // note that (forall i: (i > high) => (arr[i] is even)) and (arr[high] is even)
      //           => (forall i: (i > high-1) => (arr[i] is even))
      high -= 1;
      // P1
    }
    // P1
  }
  // P1
}
```

we now have enough information to fill in initialization; there is only one option which satisfies the invariant

we know how to swap two elements; this does not violate the invariants

we have an idea of an update, but it would violate the invariant; we can modify the invariant inside the “if body” to match, and that informs the “if condition”

```
function partition_even_odd(arr) {
  if (arr.length === 0) { return; }
  let low = 0;
  let high = arr.length - 1;
  // let P1 be (forall i: (i < low) => (arr[i] is odd)) and (forall i: (i > high) => (arr[i] is even))
  // P1
  while (???) {
    // P1
    if (???) {
      // P1
      let t = arr[low]; arr[low] = arr[high]; arr[high] = t;
      // P1
    }
    // P1
    if (arr[low] % 2 === 1) {
      // P1 and (arr[low] is odd)
      low += 1;
      // P1
    }
    // P1
    if (arr[high] % 2 === 0) {
      // P1 and (arr[high] is even)
      high -= 1;
      // P1
    }
    // P1
  }
  // P1 and (???)
  // => (exists i: (forall j: (j <= i) => (arr[j] is odd)) and (forall j: (i < j) => (arr[j] is even)))
}
```

in order to know what the “while condition” should be, we need to know what state we want when the “while loop” finishes; we want the array to be fully partitioned at the end

```
function partition_even_odd(arr) {
  if (arr.length === 0) { return; }
  let low = 0;
  let high = arr.length - 1;
  // let P1 be (forall i: (i < low) => (arr[i] is odd)) and (forall i: (i > high) => (arr[i] is even))
  // P1
  while (low <= high) {
    // P1
    if (???) {
      // P1
      let t = arr[low]; arr[low] = arr[high]; arr[high] = t;
      // P1
    }
    // P1
    if (arr[low] % 2 === 1) {
      // P1 and (arr[low] is odd)
      low += 1;
      // P1
    }
    // P1
    if (arr[high] % 2 === 0) {
      // P1 and (arr[high] is even)
      high -= 1;
      // P1
    }
    // P1
  }
  // P1 and (low > high)
  // => (exists i: (forall j: (j <= i) => (arr[j] is odd)) and (forall: j (i < j) => (arr[j] is even)))
}
```

once we have figured out what needs to hold after the “while loop” is over, then the “while condition” will just be the negation of that

```
function partition_even_odd(arr) {
  if (arr.length === 0) { return; }
  let low = 0;
  let high = arr.length - 1;
  // let P1 be (forall i: (i < low) => (arr[i] is odd)) and (forall i: (i > high) => (arr[i] is even))
  // P1  while (low <= high) {

    // P1
    if (???) {
      // P1
      let t = arr[low]; arr[low] = arr[high]; arr[high] = t;
      // P1
    }
    // P1 and (???)
    if (arr[low] % 2 === 1) {
      // P1 and (arr[low] is odd)
      low += 1;
      // P1 and (high-low decreases)
    }
    // P1 and (???) and ((arr[old low] was odd) => (high-low decreases))
    if (arr[high] % 2 === 0) {
      // P1 and (arr[high] is even)
      high -= 1;
      // P1 and (high-low decreases)
    }
    // P1 and (???) and ((arr[old low] was odd) => (high-low decreases))) and ((arr[old high] was even) => (high-low
decreases))
    // => P1 and (???) and (((arr[old low] was odd) or (arr[old high] was even)) => (high-low decreases)))
    // => P1 and (high-low decreases))
  }
  // P1 and (low > high)
  // => (exists i: (forall j: (j <= i) => (arr[j] is odd)) and (forall: j (i < j) => (arr[j] is even)))
}
```

Are we making progress in each iteration, i.e., is high-low decreasing? Let's look at the 4 cases.

4 cases

Value at idx low	Value at idx high	Progress (high-low)
odd	odd	decreases by 1
even	even	decreases by 1
odd	even	decreases by 2
even	odd	What should ??? be to guarantee progress?

```
function partition_even_odd(arr) {
  if (arr.length === 0) { return; }
  let low = 0;
  let high = arr.length - 1;
  // let P1 be (forall i: (i < low) => (arr[i] is odd)) and (forall i: (i > high) => (arr[i] is even))
  // P1
  while (low <= high) {
    // P1
    if (???) {
      // P1
      let t = arr[low]; arr[low] = arr[high]; arr[high] = t;
      // P1
    }
    // P1 and (???)
    if (arr[low] % 2 === 1) {
      // P1 and (arr[low] is odd)
      low += 1;
      // P1 and (high-low decreases)
    }
    // P1 and (???) and ((arr[old low] was odd) => (high-low decreases))
    if (arr[high] % 2 === 0) {
      // P1 and (arr[high] is even)
      high -= 1;
      // P1 and (high-low decreases)
    }
    // P1 and (???) and ((arr[old low] was odd) => (high-low decreases))) and ((arr[old high] was even) => (high-low
decreases))
    // => P1 and (???) and (((arr[old low] was odd) or (arr[old high] was even)) => (high-low decreases)))
    // => P1 and (high-low decreases))
  }
  // P1 and (low > high)
  // => (exists i: (forall j: (j <= i) => (arr[j] is odd)) and (forall: j (i < j) => (arr[j] is even)))
}
```

we are still missing one more “if condition”, and no clear violations of the invariants; however, there is a hidden assumption that we want the loop to “make progress” on every iteration; from the “while condition” we see that our definition of “make progress” is either increasing “low” or decreasing “high”. This implies decreasing value of high-low compared to prev iteration and a progress towards termination.

```
function partition_even_odd(arr) {
  if (arr.length === 0) { return; }
  let low = 0;
  let high = arr.length - 1;
  // let P1 be (forall i: (i < low) => (arr[i] is odd)) and (forall i: (i > high) => (arr[i] is even))
  // P1
  while (low <= high) {
    // P1
    if (???) {
      // P1
      let t = arr[low]; arr[low] = arr[high]; arr[high] = t;
      // P1
    }
    // P1 and ((arr[low] is odd) or (arr[high] is even))
    if (arr[low] % 2 === 1) {
      // P1 and (arr[low] is odd)
      low += 1;
      // P1 and (high-low decreases)
    }
    // P1 and ((arr[old low] was odd) or (arr[high] is even)) and ((arr[old low] was odd) => (high-low decreases))
    // => P1 and ((arr[high] is even) or (high-low decreases))
    if (arr[high] % 2 === 0) {
      // P1 and (arr[high] is even)
      high -= 1;
      // P1 and (high-low decreases)
    }
    // P1 and ((arr[old high] was even) or (high-low decreases)) and ((arr[old high] is even) => (high-low decreases))
    // => P1 and (high-low decreases)
  }
  // P1 and (low > high)
  // => (exists i: (forall j: (j <= i) => (arr[j] is odd)) and (forall: j (i < j) => (arr[j] is even)))
}
```

so we figure out that the condition
`((arr[low] is odd) or (arr[high] is even))`
must hold after our incomplete “if statement”

```
function partition_even_odd(arr) {
  if (arr.length === 0) { return; }
  let low = 0;
  let high = arr.length - 1;
  // let P1 be (forall i: (i < low) => (arr[i] is odd)) and (forall i: (i > high) => (arr[i] is even))
  // P1
  while (low <= high) {
    // P1
    if (arr[low] % 2 === 0 && arr[high] % 2 === 1) {
      // P1 and (arr[low] is even) and (arr[high] is odd)
      let t = arr[low]; arr[low] = arr[high]; arr[high] = t;
      // P1 and (arr[low] is odd) and (arr[high] is even)
      // => P1 and ((arr[low] is odd) or (arr[high] is even))
    } else {
      // P1 and ((arr[low] is odd) or (arr[high] is even))
    }
    // P1 and ((arr[low] is odd) or (arr[high] is even))
    if (arr[low] % 2 === 1) {
      // P1 and (arr[low] is odd)
      low += 1;
      // P1 and (high-low decreases)
    }
    // P1 and ((arr[high] is even) or (high-low decreases))
    if (arr[high] % 2 === 0) {
      // P1 and (arr[high] is even)
      high -= 1;
      // P1 and (high-low decreases)
    }
    // P1 and (high-low decreases)
  }
  // P1 and (low > high)
  // => (exists i: (forall j: (j <= i) => (arr[j] is odd)) and (forall: j (i < j) => (arr[j] is even)))
}
```

we can finally finish the last bit of code; all invariants hold at every step, so the code is guaranteed to be correct

```
function partition_even_odd(arr) {
  if (arr.length === 0) { return; }
  let low = 0;
  let high = arr.length - 1;
  // let P1 be (forall i: (i < low) => (arr[i] is odd)) and (forall i: (i > high) => (arr[i] is even))
  // P1
  while (low <= high) {
    // P1
    if (arr[low] % 2 === 0 && arr[high] % 2 === 1) {
      // P1 and (arr[low] is even) and (arr[high] is odd)
      let t = arr[low]; arr[low] = arr[high]; arr[high] = t;
      // P1 and (arr[low] is odd) and (arr[high] is even)
    }
    // P1 and ((arr[low] is odd) or (arr[high] is even))
    if (arr[low] % 2 === 1) {
      // P1 and (arr[low] is odd)
      low += 1;
      // P1 and (high-low decreases)
    }
    // P1 and ((arr[high] is even) or (high-low decreases))
    if (arr[high] % 2 === 0) {
      // P1 and (arr[high] is even)
      high -= 1;
      // P1 and (high-low decreases)
    }
    // P1 and (high-low decreases)
  }
  // P1 and (low > high)
  // => (exists i: (forall j: (j <= i) => (arr[j] is odd)) and (forall: j (i < j) => (arr[j] is even)))
}
```

Exercise 2: Async

Write a function asyncPosMap(*arr*: T[], *f*: T => Promise(number): Promise<T[]>. This function takes a generic array, *arr* and an asynchronous function *f*, and returns a new Promise. That promise should be fulfilled with a new array containing the elements of *arr* that for which f resolved to a positive number. The promise should reject if at any point *f* rejects. Ensure that calls to f occur asynchronously, by using Promise.all.

Fall 2022 Midterm 2 Makeup - 20pts

Exercise 2: Solution


```
function asyncPosMap(arr: T[], f: T => Promise<number>): Promise<T[]> {

    // The first thing we have to do is get the result of applying f to every element in arr.



}
```

Exercise 2: Solution


```
function asyncPosMap(arr: T[], f: T => Promise<number>): Promise<T[]> {
        return Promise.all(                );

        // To do this we'll use Promise.all. Why do we need to do this?




}
```

Exercise 2: Solution


```
function asyncPosMap(arr: T[], f: T => Promise<number>): Promise<T[]> {
    return Promise.all(arr.map(f))

    // Let's think about what arr.map(f) will return. It returns an array of promises that will
    // resolve to a number, i.e., Promise<number>[].
    // It would be a lot more convenient if we had a Promise that resolved to a number[]
    // when all the asynchronous functions return.
    // This is what Promise.all will do.
    // It will transform our Promise<number>[] into a Promise<number[]>.
}
```

Exercise 2: Solution


```
function asyncPosMap(arr: T[], f: T => Promise<number>): Promise<T[]> {
    return Promise.all(arr.map(f)).then(

        // Now we'll use a .then to filter and retain the elements for which f resolved
        // to a positive number.
        // Notice that we can use .then because Promise.all gave us a single promise
        // instead of an array of promises. (Yay)


    );
}
```

Exercise 2: Solution


```
function asyncPosMap(arr: T[], f: T => Promise<number>): Promise<T[]> {
        return Promise.all(arr.map(f)).then(
                (nums: number[]) => {

                        // Since our promise resolves to a number[], let's use it now to filter.


                }
        );
}
```

Exercise 2: Solution


```
function asyncPosMap(arr: T[], f: T => Promise<number>): Promise<T[]> {
    return Promise.all(arr.map(f)).then(
        (nums: number[]) => {
            // need to return elements of arr, not nums
            // Return type is Promise<T[]>, not Promise<number[]>
            return arr.filter(...);
        }
    );
}
```

Exercise 2: Solution


```
function asyncPosMap(arr: T[], f: T => Promise<number>): Promise<T[]> {
    return Promise.all(arr.map(f)).then(
        (nums: number[]) => {
            // .filter (and other array HoF) can supply index of the current element
            // we don't need the element (first parameter) so we use "_"
            // what condition replaces …?
            return arr.filter((_, i) => … );
        }
    );
}
```

Exercise 2: Solution


```
function asyncPosMap(arr: T[], f: T => Promise<number>): Promise<T[]> {
        return Promise.all(arr.map(f)).then(
                (nums: number[]) => {
                        return return arr.filter((_, i) => nums[i] > 0 );
                }
        );
}

// If any call to f rejects, Promise.all will return a promise that rejects. This will bubble
// though our .then. Thus, if f rejects, so does our returned promise.
```