CS245 Review Notes Midterm Edition

Preamble

Welcome to my unofficial course notes for the final exam of CS245(Summer 2014)

The course material that has influenced the contents of these course notes are as follows:

- Course Textbook.
- Course slides.
- Posts on the CS245 Piazza.
- Student perspectives.

To preserve the chronological order that the material was presented during lecture, these review notes will follow a similar structure. However, some material in these review notes may be introduced earlier or later. This will only occur if we feel as if material flows better in a separate section than it was originally introduced in.

Disclaimer

I ask that you respect all of the guidelines I set forth with respect to these review notes. If you are unable to respect any of the guidelines mentioned in the proceeding few paragraphs, please delete this file or close the hard copy of the review notes.

1. I alone reserve the right to alter and distribute these review notes. If you see a mistake or feel as if important content is missing, contact me (information provided below) so that I can make the appropriate correction(s).
2. I will be supporting the accuracy of these review notes for the remainder of the S14 semester. After such a time these review notes should be considered obsolete.
3. These review notes are provided absolutely free of charge. If you paid for a hard copy or e-copy of these review notes then you are obligated to contact us so that we can take the appropriate action(s) towards the distributor.

Keep in mind that these notes are in no way guaranteed to be accurate. There may be mistakes, outdated information or tangents that will not be directly related to some of the material presented in the course. These notes have been developed by me, a student during my undergraduate studies taking CS245. No professor, TA or anyone involved in the administration at the University of Waterloo has endorsed these notes.

If you feel a need to contact me at any time for clarification, issues with the notes whatever, you can reach me at my contact information below:
Jacob Willemsma, author (**wjwillem@uwaterloo.ca**).

Acknowledgements

Some further acknowledgments to the CS245 material and other influencing figures.

- Design inspired by "Linear Algebra Course Notes" by Dan Wolczuk (with permission).
- CS136 Course Notes by Jacob Pollack and Jacob Willemsma

Table of Contents

1 Program verification

1.1 Introduction

DEFINITION
Program correctness

The act of checking if a given program satisfies its specification, i.e. does it do what it should?

There are a few ways to check for program correctness.

- inspection.
- testing (white box and black box).
- formal verification.

You guessed it, for this class, we'll do it the formal way.

DEFINITION
Program Specification

Rules by which the program must obey. Preconditions and Postconditions.

DEFINITION
Formal verification

Formal verification tests a programs correctness by using formal program states and by proving the a program satisfies the specification for **all** valid inputs.

We're obviously not going to do this with a whole complicated language like C++, instead we'll focus on a smaller subset of a language like C/C++ or Java. Our *Core Programming Language* contains some familiar concepts:

- Integer and Boolean Expressions.
- Assignment.
- Sequence.
- If - Then - Else.
- While/For loops.
- Arrays.
- Functions and Procedures.

DEFINITION
State

The *state* of a program is the values of the variable at a **particular time** in the execution of the program. The reason for a state being only circumstance of a program at a particular time is that expressions often evaluate relative to the current state of the program (i.e. we can go inside the program and see what's going on). Commands change the state of the program.

Say we had the following function (calculate the factorial).

```
y = 1;
z = 0;
while (z != x) {
    z = z + 1;
    y = y * z;
}
```

A few program states would then be:

- Initial state s0: z=0, y=1.
- State s1: z=1, y=1.
- State s2: z=2, y=2.
- State s3: z=3, y=6.
- State s4: z=4, y=24.
- ...

In order to verify program correctness, we need to create a formal of which to present our data.

DEFINITION
Hoare triple

A specification of a program C is a Hoare triple with C as the second component: $\{P\}$ C $\{Q\}$
In this format, P represents the preconditions, C the program code and Q the post-conditions. P and Q collectively make up the program specifications.

EXAMPLE 1

If the input x is a positive integer, compute a number whose square is less than x.

```
{x > 0} C { y * y < x}
```

1.2 Correctness

Given that we now have the notion of a Hoare triple, we want to develop a notion of a proof which will allow us to prove that a program C satisfies the given specification P and Q.

DEFINITION **Partial correctness**

A triple $\{P\}\ C\ \{Q\}$ is **satisfied under partial correctness** if and only if while executing the program on every possible **state** the resulting state, *s'*, if the program were to terminate, *s'* would also satisfy $\{Q\}$.
We write $\models_{par} \{P\}C\{Q\}$.

Important to note triples that are partial satisfied do *not* necessary terminate. All that matters is that if it were to terminate, the current state would satisfy the postcondition $\{Q\}$.

EXAMPLE 2

In fact, here's a example of a partially correct triple that does **not** terminate.

```
{true}
while true {x = 0;}
{x = 0;}
```

Partial correctness seems like an awfully weak way to classify programs as correct. Any function that does not terminate would actually be partially correct under this definition (provided the current state satisfies $\{Q\}$).

What if we want to make sure the program works on all inputs **and** terminates?

DEFINITION **Total correctness**

A triple $\{P\}\ C\ \{Q\}$ is satisfied under total correctness if and only if while executing the program on every possible **state** the resulting state, *s'* also satisfies $\{Q\}$. That is to say, it is partially correct **and** terminates!
We write $\models_{tot} \{P\}C\{Q\}$.

EXAMPLE 2

The following is an example of total correctness.

```
{x >= 0;}
y = 1;
z = 0;
while (z != x) {
    z = z + 1;
    y = y * z;
}
{y = x!;}
```

EXAMPLE 2 The following is an example of neither, as the input is consumed.

```
{x >= 0;}
y = 1;
while (x != 0) {
    y = y * x;
    x = x - 1;
}
{y = x!;}
```

In this example, we would continuously, independently from the precondition be comparing y = 0!, since we have no notion that x in was the same as the x in $\{P\}$.

To prove correctness, we are interested in being satisfied by *total correctness*. However, in order to find out if a triple is *satisfied under total correctness* we will first go about proving that it is *satisfied under partial correctness* and then prove termination separately.

- Proving partial correctness will be done by introducing a *sound* and *complete* set of **inference rules**. Similar to what we did in natural deduction.
- Proving termination is a separate idea, but is often pretty trivia.

Occasionally in our specifications we will need additional variable that do not appear in the program.

DEFINITION **Logical variables** Logicals serve as placeholders to help differentiate variables used in code from the ones in the specifications. They do not appear in the code and are themselves not quantified (though they can follow rules for the variables in the specification.)

EXAMPLE 2 For example, in the example where x was previously consumed.

```
{x_0 >= 0;}
y = 1;
while (x != 0) {
    y = y * x;
    x = x - 1;
}
{y = x_0!;}
```

Similar to the way we did substitutions back in the glory days of CS136. Oh yeah... you remember. You wish you didn't, but you do.

1.3 Inference Rules