# Problem Set 1: Program Verification
## CS 173 Honors Section

Assigned: September 7, 2011 Due on: September 14, 2011

One of the applications of logic in computer science is its use in proving programs to be correct. The idea is to express precisely what the program does, and then use logical inference (along with an understanding of what each statement in the program does) to construct a proof that the program is correct. This approach is explained in the class textbook by Rosen. *Read section 5.5 and the examples therein.*

In this homework we will be using a tool called *VCC* (for Verified C Compiler), developed at Microsoft Research. This tool tries to prove C programs to be correct based on specifications of correctness, and helpful assertions like *loop invariants*. In this homework, we will use the web interface to VCC that is available at `http://rise4fun.com/Vcc`. All the information about this tool that you will need for this homework is described in this problem set. But if need a reference you can consult the tutorial available at `http://research.microsoft.com/en-us/um/people/moskal/pdf/vcc-tutorial-col2.pdf`.

## 1 An Example: Division

Consider the following program [1].

```
#include <vcc.h>

void main() {
  int a;
  int b;
  int q;
  int r;

  if ((a >= 0) && (b > 0)) {
    r = a;
    q = 0;
    while (r >= b)
    {
      q++;
      r = r - b;
    }
    _(assert ((a == q*b + r) && (r < b)))
  }
}
```

The first line in this program (`#include <vcc.h>`) includes a "header" file that contains the functions that VCC uses to reason about programs. The remaining lines (except the one `_(assert ...)`) are a C program

[1] All programs used in the problem set are downloadable from the CS 173 website.

that computes the quotient and remainder when a positive integer `a` is divided by a (non-zero) positive integer `b` [2]. The quotient is stored in the variable `q` and the remainder in `r`. They are computed in the `while`-loop, where the quotient is incremented in each iteration, and the remainder is decreased by `b`, until the remainder becomes less than `b`, the divisor.

The `_(assert ...)` statement towards the end states an *assertion* that we expect to hold whenever the program reaches that point in the program; this is what we want VCC to check about the program. In this example, it says that, (a) the remainder is less than the divisor, and (b) the dividend is equal to the quotient times the divisor plus the remainder.

## 1.1 Assertions in VCC

The logical formula following `_(assert` can be any Boolean expression in C. So you can use any arithmetic operations, comparison operations, and Boolean connectives in C. Here are some commonly used ones that you may find useful.

- **Arithmetic Operations:** Addition is `+`, subtraction is `-`, multiplication is `*`, quotient in integer division is `/`, and remainder in integer division is `%`.
- **Comparison Operations:** Equality is `==` (note, `=` is assignment and not equality), non-equal is `!=` greater than equal is `>=`, and less than is `<=`.
- **Boolean Operations:** Conjunction is `&&`, disjunction is `||`, and negation is `!`.

In addtion to the above, in VCC you can use implication (`==>`) and quantifiers. Universal quantification is written as

```
\forall <type> <var>; <exp>
```

So for example

```
\forall int x; (x >= 0) || (x < 0)
```

says that every `int x` is either greater than or equal to 0 or less than 0. Existential quantification is similarly

```
\exists <type> <var>; <exp>
```

## 1.2 Loop Invariants

Copy and paste the above code in the VCC window (at `http://rise4fun.com/Vcc`), and press the "ask vcc" button. If you did everything correctly, VCC will get back to you with the following two comments

[2] The program does not use any specific C features, and so anyone who has some experience with programming in Java/C should be able to follow the code.

1. "`q++` may overflow", which says that the incrementing of q may result in a number that is larger than the largest integer in C (resulting in an overflow) [3].

2. The assertion at the end could not be verified.

VCC could not, on its own, establish the correctness of the program. This is not just a weakness of VCC; you will learn in CS 373 that it is impossible to write a program (in any language, on any platform) that will automatically "verify" an arbitrary program. It is often the case that the main challenge for a program like VCC is to reason about the loops (and recursion, which we don't have in this example). It cannot figure out what happens in the loop, or how many times the loop runs (which leads to it suspecting that the increment may overflow). We can help VCC by providing it what is called a *loop invariant*.

A loop invariant is something that is true in every iteration of the loop. Look at the following changed program.

```
#include <vcc.h>

void main() {
  int a;
  int b;
  int q;
  int r;

  if ((a >= 0) && (b > 0)) {
    r = a;
    q = 0;
    while (r >= b)
    _(invariant a == q*b + r)
    {
      q++;
      r = r - b;
    }
    _(assert ((a == q*b + r) && (r < b)))
  }
}
```

We have added a new line `_(invariant ...)` which states the loop invariant. Once again, what follows after the keyword `invariant` is any VCC assertion of the kind that was described earlier. For this example, we assert that `a` is always `q` times `b` plus `r`. Notice the slight difference between the loop invariant and the assertion we are trying to verify. Also, notice where the invariant is placed — after the `while` but before the body starts.

Copy and paste the code with the loop invariant in the VCC window and press the "ask vcc" button. VCC should now successfully verify the program!

**What happened?** The loop invariant captured everything that one needs to know about the loop in order to verify the last assertion. VCC could very easily establish that the loop invariant holds the first time you

[3] Ensuring that variables do not overflow is very important not only for correctness but also to ensure that the program is secure. Some attacks rely on causing a variable to overflow and exploit that to breach security.

enter the loop, and if the loop invariant holds then it continues to hold in the next iteration. Finally, observe that on coming out of the loop, we know that the loop invariant holds and the condition in the `while`-loop is false (which is why we are out of the loop); these together are exactly the assertion we are trying to establish. Notice that even though we did not say how long the loop will run, using the loop invariant VCC was also able to establish that the quotient `q` would not overflow in the loop!

Other examples of programs and proving them correct can be found in Examples 4 and 5 on pages 375–376 in the class textbook by Rosen. Read them to get a better feel for them.

# 2 Computing Primes

Consider the following program that checks if a number is prime.

```
#include<vcc.h>

_(logic bool prime(int n, int k) = ... )

void main() {
  int p;
  int n;
  int i;

  i = 2;
  p = 1;
  if (n > 2) {
    while (i != n)
    {
      p = (p > 0) && (n%i != 0);
      i++;
    }
  }
  _(assert ((p > 0) ==> prime(n,n)) && (prime(n,n) ==> (p > 0)))
}
```

Ignore the VCC-related lines (i.e., those that start `_(...)`). The variable `p` is Boolean-valued and is initially set to **true** (i.e., a value $> 0$). Any time we find a factor of `n` (other than 1 and itself) we set this variable to 0. Thus, the value of `p` records whether `n` is prime — if it is $> 0$ it is prime and if it is $= 0$ it is not.

**Problem 1:** Write a logical formula prime$(n, k)$ in VCC syntax that is true if no number $\geq 2$ and $< k$ is a factor of $n$. Write this assertion instead of ... in the line that begins with `_(logic ...)`. Thus, a number $n$ is prime iff prime$(n, n)$ holds. Statements begining with `_(logic ...)` allow one to define (in VCC) a logical formulas that can be subsequently used in assertions and invariants. Thus, the correctness of the program (the line `_(assert ...)`) states that `p` is $> 0$ if and only if `n` is prime.

Copy and paste the program (after you write the formula for `prime`) and press the "ask vcc" button. VCC will not successfully prove your program to be correct.

**Problem 2:** Write a loop invariant for the while loop such that VCC successfully verifies the correctness of the program.

## 2.1 What and how to submit

Send an email containing your program (with the formula for `prime` and the loop invariant) that successfully passes in VCC to `vmahesh@cs.illinois.edu`.