ETH zürich

Alexander J. Summers

Program Verification

Exercise Sheet 12: Verification of Rust Programs

Assignment 1 (Unreachable statements)

Consider the following Rust program, which we are going to encode and verify in Viper:

```
// ‘u32‘ is the type of an unsigned integer
fn correct(mut x: u32, mut y: u32) -> u32 {
    y += 1;

    // ‘x / y‘ is an integer division
    if x / y == 0 {
        x += 1;
    }

    if x == 0 {
        unreachable!()  // crashes if executed
    } else {
        y / x  // this is the returned value
    }
}
```

Assume that due to the encoding of references (not used in this exercise) the type `u32` is encoded to the following Viper predicate.

```
field int_val: Int

predicate U32(self: Ref) {
    acc(self.int_val)
}
```

In this encoding, a Rust local variable `x: u32` is encoded as a Viper reference and an instance of the `U32` predicate.

- How would you encode the Rust function `correct`, such that Viper reports a verification error if the statement `unreachable!()` is reachable? Write the corresponding Viper program and check that it verifies.
- Using *the same* encoding technique as before, encode to Viper the following Rust function and check that it does not verify. What would be a reasonable precondition for this function, such that the Viper encoding verifies?

```
fn wrong(x: u32, y: u32) -> u32 {
    if y == 0 {
        unreachable!()
    } else {
        x / y
    }
}
```

Assignment 2 (Value ranges and overflow checks)

Consider the Rust functions listed below, which we are going to verify with respect to the following two properties:

(i) crash freedom (i.e. no `unreachable!()` statement can ever be executed), and

(ii) overflow freedom (i.e. no integer under/overflow happens at runtime).

```
fn wrong(x: u16, y: u16) -> u16 {
    x * y  // potential integer overflow
}

fn correct1(x: u16) {
    if x < 0 {
        unreachable!()  // crashes if executed
    }
}

fn correct2(x: u16, y: u16) -> u16 {
    if x <= 100 && y <= 100 {
        x * y
    } else {
        0
    }
}

fn correct3(x: u16, y: u16) -> u32 {
    let xx = x as u32;  // casting `u16` to `u32` preserves the value
    let yy = y as u32;
    xx * yy
}
```

In these examples it is important to encode the value ranges of the integer types `u16` and `u32`:

- `x: u16` $\Rightarrow x \in [0, 65535]$
- `x: u32` $\Rightarrow x \in [0, 4294967295]$

Two techniques to encode the value ranges are the following:

(a) the assertion that encodes the value range can be encoded as part of the predicate that encodes `u16` and `u32`;

(b) the assertion that encodes the value range can be assumed and checked, where needed, using the `assume` and `assert` statements.

- What are the advantages and disadvantages of the two techniques? Which one would you prefer to use in a tool that encodes Rust to Viper?
- Imagine now that we are only interested to verify the crash freedom property. That is, we are not interested in detecting potential overflows and we do not want to pay verification time for it. How would you change the encoding, such that all Rust functions above verify?