ETH zürich

Alexander J. Summers

Program Verification

Exercise Solutions 12: Verification of Rust Programs

Assignment 1 (Unreachable statements)

As seen in the last lecture, a Rust `unreachable!()` statement can be encoded as an `assert false` statement in Viper:

```
field int_val: Int

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

method correct(x: Ref, y: Ref) returns (r: Ref)
    // We need to encode that unsigned integers are non-negative, to avoid
    // divisions by zero later. See also the following assignment.
    requires U32(x) && unfolding U32(x) in x.int_val >= 0
    requires U32(y) && unfolding U32(y) in y.int_val >= 0
    ensures U32(r)
{
    unfold U32(x)
    unfold U32(y)

    y.int_val := y.int_val + 1

    if (x.int_val / y.int_val == 0) {
        x.int_val := x.int_val + 1
    }

    if (x.int_val == 0) {
        assert false // unreachable!()
    } else {
        r := new(int_val)
```

```
        r.int_val := y.int_val / x.int_val
    }

    fold U32(r)
}

method wrong(x: Ref, y: Ref) returns (r: Ref)
    requires U32(x) && unfolding U32(x) in x.int_val >= 0
    requires U32(y) && unfolding U32(y) in y.int_val >= 0
    // Additional precondition to avoid the unreachable!() statement:
    requires unfolding U32(y) in y.int_val > 0
    ensures U32(r)
{
    unfold U32(x)
    unfold U32(y)

    if (y.int_val == 0) {
        assert false // unreachable!()
    } else {
        r := new(int_val)
        r.int_val := x.int_val / y.int_val
    }

    fold U32(r)
}
```

Assignment 2 (Value ranges and overflow checks)

- None of the two techniques is strictly better than the other. By using technique (a) the encoding is more concise and readable. This is useful even in a tool that automatically translates Rust to Viper, because it makes debugging easier. Prusti, for example, uses this technique. However, the overflow check performed by `fold U32(x)` is not sufficient in a program that performs a sequence of mathematical operations, e.g. `(x - y) + y`, because the second operation might hide the overflow of the first one if the predicate is not folded between them. So, part of technique (b) still needs to be applied in these cases.

- The technique (a) can be adapted by assuming that the value range is valid before each `fold` statement, and by removing any other value range check. The technique (b) can be adapted by removing all value range checks.

  Note that in both cases the verifier can prove `false` whenever an overflow happens. This is fine, because we explicitly decided to ignore all program traces that contain an integer overflow.