ETH zürich

Alexander J. Summers

# Program Verification

## Exercise Solutions 10: Heap Reasoning and Permissions

## Assignment 1 (Framing and Monotonicity)

In a special case where $C$ is self-framing and does not mention any local variables modified by $s$, the validity of $\vDash \{A\}s\{B\}$ implies the validity of $\vDash \{A * C\}s\{B * C\}$. An example that illustrates that this does not hold if $C$ is not self-framing would be $A = B = \{\mathit{acc}(x.f)\}$, $C = \{x.f = 5\}$, and $s$:

```
x.f := 1
```

Similarly, the necessity of $C$ not mentioning modified variables can be illustrated with $A = B = \{\mathit{true}\}$, $C = \{x = 5\}$, and $s$:

```
x := 1
```

To see why the presented requirements are sufficient for the Hoare triple $\vDash \{A * C\}s\{B * C\}$ to be valid, we need to consider two questions. First, why can the program $s$ not fail? In the operational semantics defined on slide 212, all permission checks are lower bounds. Therefore, they cannot fail because $A * C$ has at least as much permission as $A$ (recall that reaching an inconsistent state, for example, having a permission amount 2 for $x.f$ in the permission mask would make the program trace to go to magic). Second, why can the program $s$ not violate $C$? Since the Hoare triple $\vDash \{A\}s\{B\}$ is valid, if the program $s$ writes to a memory location $x.f$, it must have full permission to that memory location solely from $A$ or *inhale* it during its execution. Therefore, having any non-zero permission for $x.f$ in $C$ would make the program $s$ reach a state with more than full permission to a memory location that would make the trace to go to magic (recall that we require $C$ to be self-framing).

This ability to extend the state with some additional state $C$ is called *monotonicity* and is important in verification for at least two reasons. First, it enables composition: one can verify that the program $s$ works correctly when started in a state that satisfies $A$ and then plug the program $s$ into a larger program where $s$ is started in a larger state $A * C$. Second, it solves the framing problem: if we know that $s$ depends only on $A$, we can soundly preserve all our additional knowledge expressed as $C$ during the execution of $s$.

In the soundness argument we presented above we mentioned that we rely on the fact that all permission checks that can lead to errors are lower bounds. However, if we extend implicit dynamic frames with an expression $\mathit{perm}(e.f)$, we can use it to bound the currently held permission amount from above. For example, the following program would succeed then started in a state $A = \mathit{true}$:

```
    assert perm(x.f) == none
```

but would fail when started in a state $A * C$ where $C = acc(x.f)$.

# Assignment 2 (Framing)

Encoding of `solve1` where all C++ variables are modelled as local Viper variables:

```
function fib(n: Int): Int {
    n <= 2 ? 1 : fib(n-1) + fib(n-2)
}

method solve1(n: Int) returns (res: Int)
    requires n >= 1
    ensures res == fib(n+1)
{
    var a: Int := 1;
    var b: Int := 1;
    var i: Int := 1;
    while (i < n)
        invariant i <= n
        invariant 1 <= i
        invariant a == fib(i)
        invariant b == fib(i + 1)
    {
        var temp: Int := a;
        a := b;
        b := a + temp;
        i := i + 1;
    }
    res := b;
}
method main() {
    var n: Int := 1;
    var tmp: Int;
    tmp := solve1(n);
    assert tmp == 1;
}
```

Encoding of `solve2` where all C++ variables are modelled as allocated on the Viper heap:

```
function fib(n: Int): Int {
    n <= 2 ? 1 : fib(n-1) + fib(n-2)
}

field int_val: Int
field ref_val: Ref
```

```
method solve2(nref: Ref) returns (res: Ref)
    requires acc(nref.ref_val, 1/2)
    requires acc(nref.ref_val.int_val, 1/2)
    requires nref.ref_val.int_val >= 1
    ensures acc(nref.ref_val, 1/2)
    ensures acc(nref.ref_val.int_val, 1/2) && acc(res.int_val)
    ensures res.int_val == fib(old(nref.ref_val.int_val)+1)
{
    var a: Ref
    a := new(int_val)   // Modelling memory allocation.
    a.int_val := 1;
    var b: Ref
    b := new(int_val)   // Modelling memory allocation.
    b.int_val := 1;
    var i: Ref
    i := new(int_val)   // Modelling memory allocation.
    i.int_val := 1;
    while (i.int_val < nref.ref_val.int_val)
        invariant acc(i.int_val) && acc(a.int_val) && acc(b.int_val)
        invariant acc(nref.ref_val, 1/4)
        invariant acc(nref.ref_val.int_val, 1/4)
        invariant i.int_val <= nref.ref_val.int_val
        invariant 1 <= i.int_val
        invariant a.int_val == fib(i.int_val)
        invariant b.int_val == fib(i.int_val + 1)
    {
        var temp: Ref
        temp := new(int_val)
        temp.int_val := a.int_val;
        a.int_val := b.int_val;
        b.int_val := a.int_val + temp.int_val;
        i.int_val := i.int_val + 1;
    }
    res := new(int_val)
    res.int_val := b.int_val
}

method main() {
    var n: Ref;
    n := new(int_val);    // Modelling memory allocation.
    n.int_val := 1;
    var tmp1: Ref;
    tmp1 := new(ref_val);
    tmp1.ref_val := n;
    var tmp2: Ref;
    tmp2 := solve2(tmp1);
    assert tmp2.int_val == 1
}
```

If the full permission amount to `n` is used in specifications, the surrounding context loses knowledge about its value. For example, if the method `solve2` took `acc(n.ref_val.int_val)` in its precondition, then after the call to `solve2` the value of `nvar.int_val` would be unknown. As a result, the verifier would not know which value was returned by the method call and would fail to prove the assertion. This happens because the caller, by giving away all permissions to `nvar.int_val`, allows the callee to modify the location and, therefore, it would be unsound to assume that the value was not changed. Similarly, if the loop invariant required the same permission amount as the precondition of the method `solve2`, the verifier would not know the value of `n.ref_val.int_val` after the loop. In this particular example, this would not cause a problem because the postcondition relates the result to the current value of `n.ref_val.int_val`. Instead of using fractional permissions, one could ensure that the value has not changed by using `old` expressions.