ETH zürich

Alexander J. Summers

# Program Verification

## Exercise Sheet 10: Heap Reasoning and Permissions

### Assignment 1 (Pure Variables and Heap)

Let's consider the following competitive programming task[1]:

> A staircase has $n$ steps. John is going upstairs and can either make a single step or jump over two steps. Question: in how many different ways John can climb up the stairs?

A possible solution to this problem written in C++ would be:

```
int solve1(int n) {
    int a = 1;
    int b = 1;
    int i = 1;
    while (i < n) {
        int temp = a;
        a = b;
        b = a + temp;
        i = i + 1;
    }
    return b;
}
int main() {
    int n = 1;
    assert(solve1(n) == 1);
    return 0;
}
```

Encode this implementation in Viper and show that the returned value is equivalent to $(n+1)$-th Fibonacci number as modelled by this Viper function:

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

---

[1] We would like to thank Artūras Lapinskas for sharing this task.

Now imagine that the programmer decided to pass `n` not by value, but by reference:

```
int solve2(int* nref) {      // Modified line.
    int a = 1;
    int b = 1;
    int i = 1;
    while (i < *nref) {     // Modified line.
        int temp = a;
        a = b;
        b = a + temp;
        i = i + 1;
    }
    return b;
}
int main() {
    int n = 1;
    assert(solve2(&n) == 1);
    return 0;
}
```

We cannot model `n` in Viper as a local variable of type `Int` because we would have no way to model `&n`. One way to circumvent this problem would be to model `n` as a local variable of type `Ref` with a field of type `Int`. For example, the code snippet:

```
int x = 1;
int* y = &x;
```

could be encoded as:

```
field int_val: Int
field ref_val: Ref
// ...
var xaddr: Ref;
xaddr := new(int_val);          // Allocating a local variable
                                // on the stack.
xaddr.int_val := 1;
var yaddr: Ref
yaddr := new(ref_val);          // Allocating a local variable
                                // on the stack.
yaddr.ref_val := xaddr;
```

Here the local variable `xaddr` corresponds to the address of the variable `x` and `xaddr.int_val` corresponds to the value stored at that location. Similarly, `yaddr` and `yaddr.ref_val` corresponds to the address of the variable `y` and its value respectively.

Encode the `solve2` procedure in Viper by using this approach. What happens if you use the full permission amount in all specifications? Why? What are possible ways to fix the issue?

# Assignment 2 (Framing and Monotonicity)

Let's consider a valid Hoare triple $\vDash \{A\}s\{B\}$ where $A$ and $B$ are implicit dynamic frames assertions. Recall that the validity of this triple means that the program $s$ started in a state satisfying $A$ will not reach an error state and if it terminates the end state will satisfy $B$.

Now consider that we start the program $s$ in a state satisfying $A * C$ (recall that $*$ is a separating conjunction). Can we say anything about the end state? In other words, what can we write instead of $???$ so that $\vDash \{A * C\}s\{???\}$ is guaranteed to be a valid Hoare triple (for any statement $s$)?

Now imagine that we extend implicit dynamic frames with an expression *perm*$(r.f)$ that can be used to inspect the currently held permission amount to the field $f$ of the reference $r$. For example, by using this expression one could assert that we currently hold exactly $1/2$ permission to $x.f$:

```
assert perm(x.f) == 1/2
```

Does your conclusion about what you can write instead of $???$ in $\vDash \{A * C\}s\{???\}$ still hold in this extended logic?