Program Analysis for Software Security

Seminar 5

Assignment 5

- Please complete the following tasks until last 2 weeks of the semester.
- The Assignment 5 must be presented by all group members in the class.
- The code of the below problems is given in the Teams folder.

- Consider axiomatization of 2D points on the right. We added a function and an axiom for adding two points by adding their components.
- Try out different triggering patterns for the axiom on the right and test them for client below. Find patterns such that
  a) verification succeeds,
  b) verification fails, and
  c) verification does not terminate.

```
// file: examples/06-trigger-point.vpr
domain Point {
function cons(x: Int, y: Int): Point
  function first(p: Point): Int
  function second(p: Point): Int

  function add(p: Point, q: Point): Point

  axiom {
    forall p: Point, q: Point ::
     first(add(p,q)) == first(p) + first(q)
     && second(add(p,q)) == second(p) + second(q)
   }
   // ...
}
```

```
method client() {
  var x: Point := add( cons(17, 42), cons(3,8) )
  assert first(x) == 20
  assert second(x) == 50
}
```

- Use a lemma to verify the following client:

```
// file: 16-exercise.vpr
function foo(x: Int): Int {
  x <= 0 ? 1 : foo(x - 2) + 3
}

method client(r: Int) {
  var s: Int := foo(r)
  var t: Int := foo(s)

  assert 2 <= t - r
}
```

- Bonus: prove the following lemma (including termination):

```
// file: 17-commutativity.vpr
function X(n: Int, m: Int): Int
  requires n >= 0 && m >= 0 {
  m == 0 ? 0 : n + X(n, m-1)
}

method lemma_X_commutative (n: Int, m: Int)
  requires n >= 0 && m >= 0
  ensures X(n, m) == X(m, n) {
  // TODO: show commutativity of
  //       multiplication function X
}
```

Exercise: swapping the fields of two objects

➔ 04-swap.vpr

- Implement a swap method that exchanges the field values of two objects.
- Specify its functional behavior.
- Write a client method that creates two objects and calls swap on them. Include an assertion to check that swap's specification is strong enough.
- Change your client method such that it calls swap, passing the same reference twice.

```
field f: Int

method swap(a: Ref, b: Ref)
{ … }
```

- Reconsider the method on the right.
- Change the precondition such that we can call the method by passing both aliasing references and non-aliasing references to it as arguments without violating the precondition.
- Does the assertion still hold?
  Why (not)?

```
method alias(a: Ref, b: Ref)
  requires acc(a.f) && acc(b.f)
{
  a.f := 5
  b.f := 7
  assert a.f == 5
}
```

Exercise: working with permissions

➔ `07-account.vpr`

- Implement, specify, and verify a class for bank accounts with the following methods:
  - `create` returns a fresh account with initial balance 0
  - `deposit` deposits a non-negative amount to an account
  - `transfer` transfers a non-negative amount between two accounts
  - Account balances are integers.

- Verify the client program on the right.

```
method client()
{
  var x: Ref
  var y: Ref
  var z: Ref
  x := create()
  y := create()
  z := create()
  deposit(x, 100)
  deposit(y, 200)
  deposit(z, 300)
  transfer(x, y, 100)
  assert x.bal == 0
  assert y.bal == 300
  assert z.bal == 300
}
```

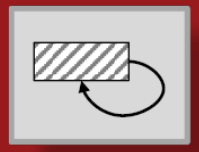
Exercise: cyclic lists

➔ `07-cyclic.vpr`

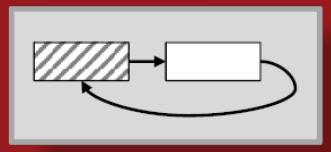
a. Write a predicate `list(this)` that represents cyclic lists
   Hint: use the `lseg` predicate

b. Implement and verify a method that creates an empty list

c. Implement and verify a method that inserts an element right after the sentinel node



empty list (sentinel only)



list with one element (plus sentinel)

```
predicate lseg(this: Ref, last: Ref) {
  this != last ==> acc(this.next) &&
                   lseg(this.next, last)
}
```

Exercise: sorted lists

- Write a user-defined predicate `list(this)` that represents sorted integer lists

➔ `10-list-sorted.vpr`

Exercise: sharing

➔ 05-flyweight.vpr

- Implement a simplified version of the Flyweight pattern with the following properties:
- A flyweight object has a single field `val`.
- The factory manages only one object.
- The factory's `get` method returns a flyweight object and provides read access to its `val` field.
- It obtains this flyweight object from a cache, and creates it if the cache is empty.