Verification-Driven Program Development
Exercise Sheet, Week 5

Ștefan Ciobâcă

30.10.2024

References

1. `https://dafny.org/teaching-material/Lectures/2-1-Logic-Propositions.html`
2. `https://dafny.org/teaching-material/Lectures/2-2-Logic-Definitions.html`

Exercises

1. There are two styles to tell Dafny about a function:

   (a) axiomatize it:

   ```
   ghost function fact(n : int) : int

   lemma fact0()
     ensures fact(0) == 1

   lemma fact1(n: int)
     requires n > 0
     ensures fact(n) == n * fact(n - 1)
   ```

   (b) define it:

   ```
   function f(n : int) : int
     requires n >= 0
     decreases n
   {
     if n == 0 then
       1
     else
       n * f(n - 1)
   }
   ```

   • TODO: Teach Dafny about the Fibonacci function in the two styles.

2. Defining a function is much better than axiomatizing it, because you risk an inconsistency in your axioms, as in the following example:

   ```
   ghost function fact(n : int) : int

   lemma fact0()
     ensures fact(0) == 1
   ```

```
lemma fact1(n : int)
  ensures fact(n) == n * fact(n - 1)

lemma helloFalse()
{
  fact0();
  assert fact(0) == 1;
  fact1(0);
  assert fact(0) == 0 * fact(-1);
  assert 0 == 1;
}
```

When defining a function, Dafny proves its existance by relying on the `decreases` annotation (called a function `variant`).

For the examples above (factorial and Fibonacci), there is no need for an explicit `decreases` annotation, because Dafny infers it automatically.

- TODO: Come up with the right `decreases` annotation for the following function:

```
function sumBetween(a : int, b : int) : int
  requires a <= b
  decreases ...
{
  if a == b then
    0
  else
    a + sumBetween(a + 1, b)
}
```

3. When you axiomatize a function, you can convince Dafny (and yourself) that the axiomatization is right by proving it equivalent to a defined version of the same function, as in the following example:

```
ghost function fact(n : int) : int

lemma fact0()
  ensures fact(0) == 1

lemma fact1(n: int)
  requires n > 0
  ensures fact(n) == n * fact(n - 1)

function f(n : int) : int
  requires n >= 0
  ensures f(n) == fact(n)
{
  if n == 0 then
    fact0();
    1
  else
    fact1(n);
    n * f(n - 1)
}
```

- TODO: Prove that the two versions of Fibonacci above are equivalent.

4. Consider the following algebraic datatype:

```
datatype Nat = Zero | Succ(Nat)
```

Dafny translates such a datatype in the following axiomatization.

```
type Nat(0) // declare (not define) Nat to be non-empty

ghost const Zero : Nat // declare Zero to be some Nat

ghost function Succ(n : Nat) : Nat // declare a successor functoin

// the following axioms ensure that Zero and Succ act as constructors
lemma succ_injective(n : Nat, m : Nat)
  requires Succ(n) == Succ(m)
  ensures n == m // no need for the other direction: n == m ==> Succ(n) == Succ(m) for any func
                 // no need for injectivity for Zero, as it is nullary

lemma zero_succ(n : Nat)
  ensures Succ(n) != Zero // two different constructors, two different values

lemma total(n : Nat)
  ensures n == Zero || exists m :: n == Succ(m) // any nat starts with one of the two construct

// the following induction principle is helpful with proofs about Nats
lemma induction(P : Nat -> bool)
  requires P(Zero) // base case
  requires forall n : Nat :: P(n) ==> P(Succ(n)) // inductive case
  ensures forall n : Nat :: P(n) // induction principle gives us that P holds for all naturals
```

- TODO: Propose a set of axioms for binary search trees defined as follows:

```
datatype BST = Leaf | Node(x : int, l : BST, r : BST)
```

Note: no need to encode the search property (for each node x: all nodes to the left of x carry an information smaller than x and all nodes to the right of x carry an information larger than x), only that `Leaf` and `Node` are constructors.

- TODO (leave this exercise for last): Propose a set of axioms for the accessors (`x`, `l`, `r`) and testers (`Leaf?`, `Node?`).

5. Predicates are functions that return `bool`:

```
predicate even(x : int)
{
  x % 2 == 0
}
```

But predicates and functions can also contain non functional expressions (which are not compilable):

```
predicate even(x : int)
{
  exists y :: x == 2 * y
}
```

```
predicate P(x : int)

lemma P_satisfiable()
  ensures exists x : int :: P(x)

ghost function giveWitness() : int
{
  P_satisfiable();
  var x :| P(x); // Hilbert's epsilon operator
  x
}
```

• TODO: define a predicate to check whether a natural number `x` is the divisor of some natural number `y` in two styles: using a functional expression (compilable) and using a quantifier (not compilable).

6. Define a non-functional predicate (uses quantifiers) that checks whether a number is the greatest common divisor of two naturals.
7. Define a function (executable) to compute the gcd of two naturals.
8. Prove that the two definitions for gcd agree.