U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Master in Informatics and Computing Engineering (M.EIC)
M.EIC037 | Formal Methods for Critical Systems 2021/22

Exercises on Program Verification with Dafny (Resolution)

1. Fibonacci numbers [Selected]

Assume the following definition of Fibonacci numbers (encoded as a recursive side-effect free function in Dafny):

```
function fib(n : nat ) : nat
  decreases n
{
    if n < 2 then n else fib(n - 2) + fib(n - 1)
}
```

The computation of Fibonacci numbers with this function takes an exponential time because of the repeated work in recursive calls.

Using dynamic programming, we can derive the following optimized routine (encoded in Dafny in the imperative style) to calculate Fibonacci numbers in time O(n):

```
method computeFib (n : nat) returns (x : nat)
{
    var i := 0;
    x := 0;
    var y := 1;
    while  i < n
    {
        x, y := y, x + y; // multiple assignment
        i := i + 1;
    }
}
```

a) Identify adequate pre and post-conditions for this method, and encode them as "requires" and "ensures" clauses in Dafny. Note that you can use fib(n) in those expressions.
   requires n >= 0
   ensures x == fib(n)

b) Identify an adequate loop variant and loop invariant, and encode them as "decreases" and "invariant" clauses in Dafny.
   invariant 0 <= i <= n && x == fib(i) && y == fib(i+1)
   decreases n - i

c) Complete manually all the steps needed to prove the correctness of this methods, using the proof tableaux technique (see example of integer division in the slides). You may encode all the

assertions of the proof tableaux directly in Dafny. Notice that Dafny does all this automatically for us, but this exercise is important to understand what Dafny does “behind the scenes”.

Proof Tableau - 1º inserir pré e pós-condição

```
{n >= 0}
i := 0
x := 0
y := 1
while i < n do
        x, y := y, x + y
        i := i + 1
{x = fib(n)}
```

Proof Tableau - 2º inserir asserções geradas pelo invariante e variante do ciclo

```
{n >= 0}
i := 0
x := 0
y := 1
{0 <= i <= n ∧ x = fib(i) ∧ y = fib(i+1) ∧ 0 <= n - i} // I
while i < n do
        {0 <= i <= n ∧ x = fib(i) ∧ y = fib(i+1) ∧ i < n ∧ 0 <= n - i ∧ n - i = V0} // I ∧ C ∧
V=V0
        x, y := y, x + y
        i := i + 1
        {0 <= i <= n ∧ x = fib(i) ∧ y = fib(i+1) ∧ 0 <= n - i < V0} // I ∧ 0 <= V < V0
{0 <= i <= n ∧ x = fib(i) ∧ y = fib(i+1) ∧ i >= n} // I ∧ ! C
{x = fib(n)} // Prova 3
```

Proof Tableau - 3º calcular pré-condições mais fracas

```
{n >= 0} // Prova 1
{0 <= 0 <= n ∧ 0 = fib(0) ∧ 1 = fib(0+1) ∧ 0 <= n - 0} // Prova 1
i := 0
{0 <= i <= n ∧ 0 = fib(i) ∧ 1 = fib(i+1) ∧ 0 <= n - i}
x := 0
{0 <= i <= n ∧ x = fib(i) ∧ 1 = fib(i+1) ∧ 0 <= n - i}
y := 1
{0 <= i <= n ∧ x = fib(i) ∧ y = fib(i+1) ∧ 0 <= n - i}
while i < n do
        {0 <= i <= n ∧ x = fib(i) ∧ y = fib(i+1) ∧ i < n ∧ 0 <= n - i ∧ n - i = V0} // Prova 2
        {0 <= i + 1 <= n ∧ y = fib(i + 1) ∧ x + y = fib(i+2) ∧ 0 <= n - i - 1 < V0} // Prova 2
        x, y := y, x + y
        {0 <= i + 1 <= n ∧ x = fib(i + 1) ∧ y = fib(i+2) ∧ 0 <= n - i - 1 < V0}
        i := i + 1
        {0 <= i <= n ∧ x = fib(i) ∧ y = fib(i+1) ∧ 0 <= n - i < V0}
{0 <= i <= n ∧ x = fib(i) ∧ y = fib(i+1) ∧ i >= n} // Prova 3 (já efetuada acima)
{x = fib(n)} // Prova 3
```

Proof Tableau - 4º provar as implicações (asserções consecutivas)

Prova 1:

```
n >= 0 => 0 <= 0 <= n ∧ 0 = fib(0) ∧ 1 = fib(0+1) ∧ 0 <= n - 0
⇔  0 <= n => 0 <= n ∧ 0 = 0 ∧ 1 = 1 ∧ 0 <= n
⇔  0 <= n => 0 <= n ∧ True ∧ True
⇔ 0 <= n => 0 <= n
⇔  True
```

Prova 2:

0 <= i <= n ∧ x = fib(i) ∧ y = fib(i+1) ∧ i < n ∧ 0 <= n - i ∧ n - i = V0 => 0 <= i + 1 <= n ∧ y = fib(i + 1) ∧ x + y = fib(i+2) ∧ 0 <= n - i - 1 < V0
⇔ 0 <= i <= n ∧ x = fib(i) ∧ y = fib(i+1) ∧ i < n ∧ 0 <= n - i ∧ n - i = V0 => 0 <= i + 1 <= n ∧ y = fib(i + 1) ∧ x + y = fib(i+2) ∧ 0 <= n - i - 1 < V0
⇔ True

Prova 3:
0 <= i <= n ∧ x = fib(i) ∧ y = fib(i+1) ∧ i >= n => x = fib(n)
⇔ i = n ∧ x = fib(i) ∧ y = fib(i+1) => x = fib(n)
⇔ true

2. Factorial of a number (n!) [Selected]

Assume the following definition of n! (encoded as a recursive side-effect free function in Dafny):

```
function method fact(n: nat) : nat
  decreases n
{
    if n == 0 then 1 else n * fact(n-1)
}
```

The computation of n! with this function takes O(n) space (stack of recursive calls), and can be optimized by the following routine (encoded in Dafny in the imperative style), which takes only O(1) space:

```
method factIter(n: nat) returns (f : nat)
{
  f := 1;
  var i := 0;
  while i < n
  {
    i := i + 1;
    f := f * i;
  }
}
```

a) Identify adequate pre and post-conditions for this method, and encode them as “requires” and “ensures” clauses in Dafny.
requires n >= 0
ensures f == fact(n)

b) Identify an adequate loop variant and loop invariant, and encode them as “decreases” and “invariant” clauses in Dafny.
invariant 0 <= i <= n && f == fact(i)
decreases n - i

c) Complete manually all the steps needed to prove the correctness of this methods, using the proof tableau technique (see example of integer division in the slides). You may encode all the assertions

of the proof tableau directly in Dafny. Notice that Dafny does all this automatically for us, but this exercise is important to understand what Dafny does "behind the scenes".

Proof Tableau

{n >= 0} // Prova 1
{0 <= 0 <= n ∧ 1 = fact(0) ∧ 0 <= n - 0} // Prova 1
f := 1
{0 <= 0 <= n ∧ f = fact(0) ∧ 0 <= n - i}
i := 0
{0 <= i <= n ∧ f = fact(i) ∧ 0 <= n - i}
while i < n do
    {0 <= i <= n ∧ f = fact(i) ∧ i < n ∧ 0 <= n - i ∧ n - i = V0} // Prova 2
    {0 <= i + 1 <= n ∧ f * (i + 1) = fact(i + 1) ∧ 0 <= n - i - 1 < V0} // Prova 2
    i := i + 1
    {0 <= i <= n ∧ f * i = fact(i) ∧ 0 <= n - i < V0}
    f := f * i
    {0 <= i <= n ∧ f = fact(i) ∧ 0 <= n - i < V0}
{0 <= i <= n ∧ f = fact(i) ∧ i >= n} // Prova 3
{f = fact(n)} // Prova 3

Obrigações de Prova:

Prova 1:
n >= 0 => 0 <= 0 <= n ∧ 1 = fact(0) ∧ 0 <= n - 0
⇔ n >= 0 => 0 <= n ∧ 1 = 1 ∧ 0 <= n
⇔ 0 <= n => 0 <= n ∧ True
⇔ True

Prova 2:
0 <= i <= n ∧ f = fact(i) ∧ i < n ∧ 0 <= n - i ∧ n - i = V0 => 0 <= i + 1 <= n ∧ f * (i + 1) = fact(i + 1) ∧ 0 <= n - i - 1 < V0
⇔ 0 <= i < n ∧ f = fact(i) ∧ n - i = V0 => 0 <= i + 1 <= n ∧ fact(i) * (i + 1) = fact(i + 1) ∧ True ∧ n - i - 1 < V0
⇔ 0 <= i < n ∧ f = fact(i) ∧ n - i = V0 => 0 <= i + 1 <= n ∧ fact(i) * (i + 1) = fact(i) * (i + 1) ∧ n - i - 1 < V0
⇔ 0 <= i < n ∧ f = fact(i) ∧ n - i = V0 => 0 <= i + 1 <= n ∧ True ∧ n - i - 1 < V0
⇔ True

Prova 3:
0 <= i <= n ∧ f = fact(i) ∧ i >= n => f = fact(n)
⇔ i = n ∧ f = fact(i) => f = fact(n)
⇔ true

3. Generalize the example of integer division (Div.dfy) to also work with negative numbers (i.e., with variables of type int). Notice that the remainder should always be non-negative.
Hints: Create an auxiliary function method `abs` to obtain the absolute value of an integer. In the specification of the new version of `div`, you need to revise the postcondition for `r`. In the implementation of the new version of `div`, you may distinguish two cases: if `n` is positive, `r` starts with value `n` and needs to be decreased until reaching the desired interval; if `n` is negative, `r` starts with value `n` and needs to be increased until reaching the desired interval. In the loop invariant(s) you may need to add the range of values of `r`.

```
// Computes the quotient 'q' and remainder 'r' of the integer
// division of a dividend 'n' by a (non-zero) divisor 'd'.
```

```
method divInt(n: int, d: int) returns (q: int, r: int)
 requires d != 0
 ensures 0 <= r < abs(d)
 ensures q * d + r == n
{
    q := 0;
    r := n;
    if n > 0
    {
        while r >= abs(d)
            decreases r
            invariant r >= 0
            invariant q * d + r == n
        {
            r := r - abs(d);
            q := q + sign(d);
        }
    }
    else if n < 0
    {
        while r < 0
            decreases abs(d) - r
            invariant r < abs(d)
            invariant q * d + r == n
        {
            r := r + abs(d);
            q := q - sign(d);
        }
    }
}

// Auxiliary function to compute the absolute value of a number (integer)
function method abs(n: int) : nat
{
    if n >= 0 then n else -n
}

// Auxiliary function to compute the signal of a number (integer)
function method sign(n: int) : int
{
    if n >= 0 then 1 else -1
}
```

4. Generalize the example of computing the power of real numbers (Power.dfy) to also work with negative exponents (of type `int`).
Hints: Create a new method `powerInt`, with `n` of type `int`, that issues appropriate calls to the existing method `powerOpt` (with `n` of type `nat`). You may need to add a precondition to `powerInt` to avoid division by zero (e.g. 0^{-1}). You may also need to add a postcondition to the function `power`, telling when the result is guaranteed to be non-zero. Notice that in Dafny real literals are written as in `0.0` and `1.0`. Notice that in Dafny you cannot call a method inside an expression; you have to assign the method result to a variable and then use the variable in the expression.

```
// Exercise TP2.4.
// Computes the power of a real number to an integer exponent.
method powerInt(x: real, n: int) returns (p: real)
   requires n < 0 ==> x != 0.0
   ensures if n >= 0 then p == power(x, n) else p == 1.0 / power(x, -n)
{
```

```
    if n < 0 {
        p := powerOpt(x, -n);
        p := 1.0 / p;
    }
    else {
        p := powerOpt(x, n);
    }
}
```