Proving Total Correctness of Imperative and Recursive Programs

Costel Anghel and Mădălina Erașcu

Objectives

- Proving total correctness of imperative algorithms using Dafny.
- Proving total correctness of recursive algorithms using Dafny.

1 Total correctness of imperative algorithms using Dafny

Arrays are fundamental data structures in computer science, used to store collections of elements (values or variables), typically of the same data type, in a contiguous block of memory. They provide efficient access to elements using an index.

Arrays are an integral part of Dafny programming language, defined by the type `array⟨T⟩`, where `T` is another type. The companion type `array?⟨T⟩` represents possibly-null arrays, encompassing references to one-dimensional arrays of type `T` (i.e., `array⟨T⟩`) and the null reference. For now, we will only consider arrays of integers, which have the type `array⟨int⟩`.

Arrays have a built-in length field, `a.Length`. Elements are accessed using bracket syntax with zero-based indexing, so `a[3]` follows `a[0]`, `a[1]`, and `a[2]`, in that order. All array accesses must be proven to be within bounds, ensuring Dafny's no-runtime-errors safety guarantee. Since bounds checks are verified during compilation, no runtime checks are necessary. New arrays are created with the new keyword, but we will focus on methods that take pre-allocated arrays as arguments.

Consider a program which checks if all elements of an array are identical and returns $True$ in this case, and $False$ otherwise. This can be expressed as follows:

```
method identic6(x: array<int>, n: int) returns (rez: bool)
ensures rez == (forall i :: 0 <= i < n ==> x[i] == x[i + 1])
{
//Write the code which satisfies the postcondition
}
```

Writing assertions about array requires specifying properties of the entire array or parts of the array, besides individual elements. In these case, the usage of universal and/or existential quantifiers is required.

For example, the condition `forall i :: 0 <= i < n ==> x[i] == x[i + 1]` specifies that each two consecutive array elements are equal.

Note that because a has type `array⟨T⟩`, it is implicitly non-null.

We can fill in the body of this method in a number of ways, one possible implementation is:

```
method identic6(x: array<int>, n: int) returns (rez: bool)
ensures rez == (forall i :: 0 <= i < n ==> x[i] == x[i + 1])
{
  var i := 0;
  rez := true;
  while i < n
  {
    if x[i] != x[i+1] {
      rez := false;
    }
    i := i + 1;
  }
  return rez;
}
```

Dafny gives us several errors, and we try to solve them step by step. First will be to specify a precondition, e.g. `requires 0 <= n < x.Length`

For proving partial correctness, every program with loops has to have an invariant. A suitable invariant (which holds at the beginning of the loop, then at each loop iteration and helps us in the end to prove the postconditon) is: `invariant 0 <= i <= n && rez == (forall j :: 0 <= j < i ==> x[j] == x[j+1])`

This invariant specifies the potential values of the index array i and the properties of the array a until index i.

For proving termination, a suitable termination function is `n-i`.

More examples on arrays in Dafny are at [1].

2 Total correctness of recursive algorithms using Dafny

Consider the Fibonacci function:

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

In Dafny, we can use a `decreases` clause to specify what decreases with each recursive call in a function or method, to help prove termination. It is used to provide an evidence by specifying an expression that decreases in a well-founded order with each recursive call.

For `Fib` we write:

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

The `decreases` clause can be read as "with each recursive call of `Fib`, the argument `n` decreases".

In this example, the depth of recursion is clearly controlled by `n`. With each recursive call, `n` gets smaller, meaning that Dafny generates and checks the proof obligation $n - 2 < n$ for the first call, and $n - 1 < n$ for the second call, ensuring that the recursive calls move towards the base case ($n < 2$).

3 Homework (Total correctness of imperative algorithms using Dafny)

Take the exercises from the appendix which were studied in the seminar Algorithms and Data Structures I, in winter semester 2019. Try to implement each algorithm in Dafny and show its total correctness. Which difficulties have you encountered in each of the cases?

4 Homework (Total correctness of recursive algorithms using Dafny)

1. With the help of Dafny, find out the maximal value of the *termination term* accepted by the verifier for the Fibbonaci program (function above). Write the proof.
2. With the help of Dafny, try different `decreases` expressions for the next functions and explain why do they work or why they don't. Explain if the default `decreases` works.

(a)
```
function F(x: int): int {
    if x < 10 then x else F(x - 1)
}
```

(b)
```
function G(x: int): int {
    if 0 <= x then G(x - 2) else x
}
```

(c)
```
function H(x: int): int {
    if x < -60 then x else H(x - 1)
}
```

(d)
```
function I(x: nat, y: nat): int {
    if x == 0 || y == 0 then 12
    else if x % 2 == y % 2 then
        I(x - 1, y)
    else
        I(x, y - 1)
}
```

(e)
```
function J(x: nat, y: nat): int {
    if x == 0 then y
    else if y == 0 then
        J(x - 1, 3)
    else
        J(x, y - 1)
}
```

(f)
```
function K(x: nat, y: nat, z: nat): int {
    if x < 10 || y < 5 then x + y
    else if z == 0 then
        K(x - 1, y, 5)
    else
        K(x, y - 1, z - 1)
}
```

(g)
```
function L(x: int): int {
    if x < 100 then L(x + 1) + 10 else x
}
```

References

[1] Dafny — Tutorial Arrays. https://dafny.org/dafny/OnlineTutorial/guide.html.