Software Specification - Dafny Project

QS 2021/2022

> 'I don't understand you,' said Alice. 'It's dreadfully confusing!' 'That's the effect of living backwards,' the Queen said kindly: 'it always makes one a little giddy at first.' 'Living backwards!? Alice repeated in great astonishment. 'I never heard of such a thing!'
>
> Lewis Carroll, Through the Looking-Glass and What Alice Found There

Exercise 1: Reversing Sequences (1val)

1. Implement a recursive Dafny function `revSeq` that, given a sequence of integers `s`, computes its reverse. For instance, it must hold that: `revSeq([1, 2, 3]) == [3, 2, 1]`.
2. Prove that the function `revSeq` is its own inverse; formally: for every sequence of integers `s`, it must hold that: `revSeq(revSeq(s)) == s`.
3. Prove the following distributivity property of `revSeq`:

$$\mathtt{revSeq}(\mathtt{s1} + \mathtt{s2}) == \mathtt{revSeq}(\mathtt{s2}) + \mathtt{revSeq}(\mathtt{s1})$$

Exercise 2: Reversing Arrays (1.5val)

1. Implement a method `reverseArr1` that, given an array of integers `arr`, returns a new array with the contents of `arr` in reverse order. Verify that the implemented method satisfies the following specification:

```
method reverseArr1 (arr : array<int>) returns (r : array<int>)
    ensures r[..] = revSeq(arr[..])
```

2. Implement a method `reverseArr2` that, given an array of integers `arr`, reverses the contents of the array **in place** without creating another array or sequence. Verify that the implemented method satisfies the following specification.

```
method reverseArr2 (arr : array<int>)
    ensures arr[..] = revSeq(old(arr[..]))
    modifies arr
```

Note: To obtain the full score at least one of the two methods must be implemented *iteratively.*

Exercise 3: Reversing Binary Trees (1val) Consider the following inductive datatype used to model binary trees whose nodes and leaves store integer values:

```
datatype Tree = Leaf(int) | Node(int, Tree, Tree)
```

1. Define a recursive function `revTree` that, given a tree `t`, computes its symmetric tree, as illustrated in Figure 1.

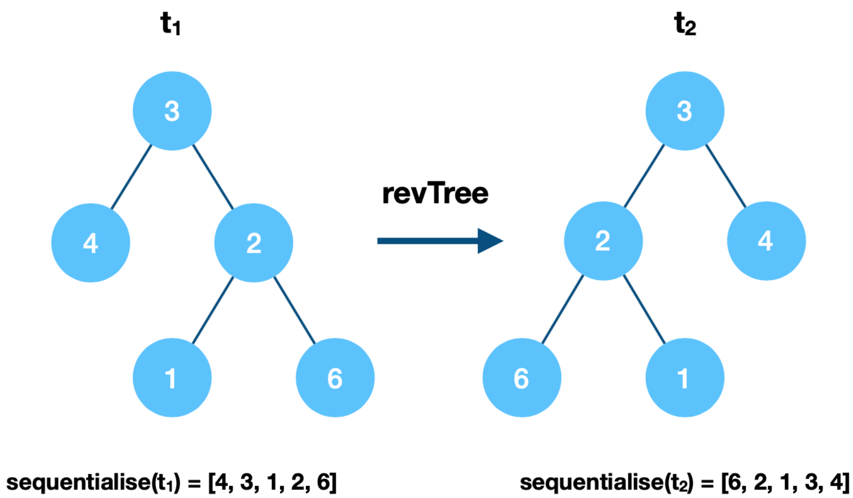

Figure 1: `revTree` and `sequentialise` example.

2. Prove that the function `revTree` is its own inverse; formally: for every tree `t`, it must hold that: `revTree(revTree(t)) == t`.

3. Define a recursive function `sequentialise` that, given a tree `t`, outputs a sequence containing all the integers in `t` from left to right. The expected behaviour of this function is illustrated in Figure 1.

4. Prove the following interchange property of the functions `revTree`, `sequentialise`, and `revSeq`:

$$\texttt{sequentialise}(\texttt{revTree}(\texttt{t})) == \texttt{revSeq}(\texttt{sequentialise}(\texttt{t}))$$

Note: To obtain the full score at least one of the proofs should be done in *calculational style*.

Exercise 4: Reversing Object-Oriented Lists (1.5val) Consider the following partial implementation of a *list node* class, `LNode`.

```
class LNode {
        ghost var list : seq<int>;
        ghost var footprint : set<Node>;
        var data : int;
        var next : LNode?;

        function Valid() : bool
            reads this, footprint
            decreases footprint;
        {
            (this in footprint) ∧
            ((next = null) ⟹ list = [ data ] ∧  footprint = { this }) ∧
            ((next ≠ null) ⟹
                (next in footprint) ∧
                footprint = next.footprint + { this } ∧
                (this ∉ next.footprint) ∧
                list = [ data ] + next.list ∧
                next.Valid())
        }
}
```

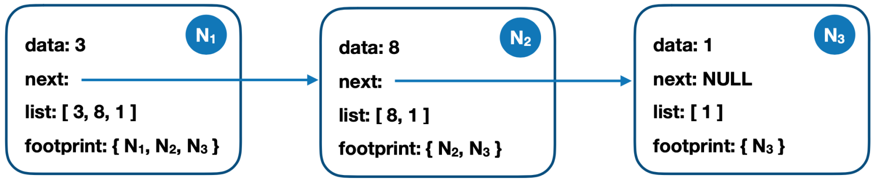

Figure 2: Example of a list consisting of three list nodes.

Each list node has two fields: **(1)** the field `data`, storing an integer value, and **(2)** the field `next`, storing the next list node, which may be `null`. Furthermore, for verification purposes, the ghost state of every list node includes the fields: **(i)** `list` that stores the sequence of integers contained in the list headed by the current node; and **(ii)** `footprint` that stores the set of all the list nodes reachable from the current node, including itself. Figure 2 gives a concrete example of a list composed of three list nodes.

1. Implement a method `reverse1` that returns the head of a new list with the contents of the *target* list in reverse order. Verify that the implemented method satisfies the following specification.

```
method reverse1 () returns (r : Node)
          requires Valid()
          ensures Valid() ∧ r.Valid() ∧ fresh(r.footprint)
          ensures r.list = revSeq(old(list))
```

2. Implement a method `reverse2` that receives a list node as input and reverses the list headed by the given node **in place**. Verify that the implemented method satisfies the following specification:

```
method reverse2 () returns (r : Node)
          requires Valid()
          ensures r.Valid() ∧ (r.list = revSeq(old(list)))
          ensures r.footprint = old(footprint)
          modifies footprint
```

Instructions

Hand-in Instructions The project is due on the 22nd of October, 2021. Be sure to follow the steps described below:

- Your solution must be comprised of four files: one file per exercise. Each exercise must implemented within its own Dafny module.[1]
- Questions 2, 3, and 4 require the properties established in Question 1. Use Dafny's `include` and `import` directives to avoid code duplication.
- Create a `zip` file containing the four answer files and upload it in **Fenix**. Submissions will be closed at 23h59 on the 22nd of October, 2021. Do not wait until the last few minutes for submitting the project.

Project Discussion After submission, you may be asked to present your work so as to streamline the assessment of the project as well as to detect potential fraud situations. During this discussion, you may be required to perform small changes to the submitted code.

[1] See `https://dafny-lang.github.io/dafny/OnlineTutorial/Modules` for a quick tutorial on how to use Dafny modules.

Fraud Detection and Plagiarism The submission of the project assumes the commitment of honour that the project was solely executed by the members of the group that are referenced in the files/documents submitted for evaluation. Failure to stand up to this commitment, i.e., the appropriation of work done by other groups or someone else, either voluntarily or involuntarily, will have as consequence the immediate failure of this year's Software Specification course for all students involved (including those who facilitated the occurrence).