Software Specification - Dafny Project

QS 2024/2025

Exercise 1: Serialising Abstract Syntax Trees (6 val = 1.5+1.5+1+1+0.5+0.5) Recall the serialisation and de-serialisation algorithms for binary trees studied in the lectures. The goal of this exercise is to adapt those algorithms for the abstract syntax trees (ASTs) of the simple expression language defined below:

```
datatype uop = Neg
datatype bop = Plus | Minus

datatype aexpr =
  Var(seq<nat>)
  | Val(nat)
  | UnOp(uop, aexpr)
  | BinOp(bop, aexpr, aexpr)
```

Informally, an expression can be: **(1)** a *variable node* storing a variable name (note that we model variable names as sequences of natural numbers, each representing a character); **(2)** a *value node* storing a natural number; **(3)** a *unary operation node* storing a unary operator and the sub-expression to which it is applied; and **(4)** a *binary operation node* storing a binary operator and the two sub-expressions to which it is applied. In order to serialise this expression language, we make use of the following code type:

```
datatype code = VarCode(seq<nat>)  | ValCode(nat) | UnOpCode(uop) | BinOpCode(bop)
```

Where each code type represents a specific AST node type in a serialised sequence. More concretely, we use: `VarCode(seq<nat>)` to represent a serialised variable node; `ValCode(nat)` to represent a serialised value node; `UnOpCode(uop)` to represent a serialised unary operation node; and `BinOpCode(bop)` to represent a serialised binary operation node. The serialisation code is given below:

```
function Serialize(a : aexpr) : seq<code>
{
 match a {
  case Var(s) => [ VarCode(s) ]
  case Val(i) => [ ValCode(i) ]
  case UnOp(op,a1) => Serialize(a1)+[UnOpCode(op)]
  case BinOp(op,a1,a2) => Serialize(a2)+Serialize(a1)+[BinOpCode(op)]
 }
}
```

1. Implement a recursive function `Deserialize(cs:seq<code>):seq<aexpr>` that given a sequence of AST codes, `cs`, generated by the `Serialize` function, outputs a sequence containing the corresponding original tree.

2. Prove that `Deserialise` is the inverse of `Serialise`, that is, for every AST `a`, it must hold that: `Deserialise(Serialise(a)) == [a]`.

3. Implement the following two recursive functions:

   - `SerializeCodes(cs:seq<code>):seq<nat>` that given a sequence of codes generates a sequence of natural numbers; and
   - `DeserializeCodes(nats:seq<nat>):seq<code>` that given a sequence of natural numbers generated by `SerializeCodes` recovers the original sequence of codes.

4. Prove that `DeserialiseCodes` is the inverse of `SerialiseCodes`, that is, for every for every sequence of codes `cs`, it must hold that: `DeserialiseCodes(SerialiseCodes(cs)) == cs`.

5. Using the functions implemented in Exercises 1.1 and 1.3, implement the following functions:

   - `FullSerialize(a:seq<code>):seq<nat>` that given an AST `a` generates a sequence of natural numbers; and
   - `FullDeserialize(nats:seq<nat>):seq<aexp>` that given a sequence of natural numbers generated by `FullSerialize` outputs a sequence containing the original AST.

6. Prove that `FullDeserialise` is the inverse of `FullSerialize`, that is, for every for every AST `a`, it must hold that: `FullDeserialize(FullSerialize(a)) == [a]`.

Note: In order to obtain the full score, the proofs of 1.2, 1.4, and 1.6 must be written in *calculational style*.

Exercise 2: Identifying Repetition (4 val = 2+2) The goal of this exercise is to implement and verify two Dafny methods for checking whether or not an array contains repeated elements.

1. File Ex2.dfy contains a *quadratic-complexity* method that takes an array `a` of natural numbers as input and returns `true` *if and only if* `a` contains no repetitions. Write the formal specification of the given method and add the loop invariants required for checking that it satisfies its specification.

2. Implement and specify a *linear-time* version of the given method and show that it also satisfies its specification. *Note:* In order to obtain the full score, the method should be implemented iteratively.

Exercise 3: A Simple List Library (2 val = 4×0.5) Consider the partial implementation of a *list node* class, `Node`, given in file Ex3.dfy. Each list node has two fields: **(1)** the field `data`, storing a natural 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)** `content` that stores the *set of naturals* 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.

Implement, specify, and verify the following methods:

1. `constructor (v:nat)`: creates a new node storing the given natural number, `v`;

2. `method add(v:nat) returns (r:Node)`: creates a new node `r` storing the given natural number, `v`, and sets its next element to be the current node (i.e., the one pointed to by `this`);

3. `method mem(v:nat) returns (b:bool)`: checks if the list headed by the current node contains the value `v`;

4. `method copy() returns (n:Node)`: creates a copy of the list headed by the current node, returning the head of the new list.

Note: The method `add` does not have to check whether or not the given element is already contained in the list.

Exercise 4: Implementing a Set with a List (4 val = 4×1) The goal of this exercise is to produce a naïve set implementation using the `Node` class implemented in Exercise 3. File Ex4.dfy contains a partial implementation of the `Set` class. Implement, specify, and verify the following methods:

1. `method mem(v:nat) returns (b:bool)`: checks if the current set contains the the value `v`;
2. `method add(v:nat)`: adds `v` to the current set;
3. `method union(s:Set) returns (r:Set)`: computes the union of the current set and the set `s` given as input;
4. `method inter(s:Set) returns (r:Set)`: computes the intersection of the current set and the set `s` given as input.

Note: The methods `mem` and `add` should have linear complexity in the number of elements contained in the set. The methods `union` and `inter` should have quadratic complexity. In order to obtain the full score, the methods `union` and `inter` must be implemented *iteratively*.

Exercise 5: Implementing a Set with a List and an Array (4 val = 1.5 + 0.5 + 2) The goal of this exercise is to produce an efficient implementation of a set using not only the `Node` class implemented in Exercise 3 but also an array for constant-time membership checking. To this end, you can assume that all the elements stored in the set are smaller than or equal to a natural number given as input to the constructor of the set. File Ex5.dfy contains a partial implementation of the more `Set` class. Complete the implementation, including:

1. the class ghost fields and validity predicate;
2. the specification and body of the class constructor;
3. the specifications and bodies of the methods `mem`, `add`, `union`, and `inter`. The behaviour of each method should be consistent with the behaviour of the corresponding method in Exercise 4. The methods `mem` and `add` must have constant time complexity (i.e., $O(1)$) and the methods `union` and `inter` must have linear complexity (i.e., $O(n)$).

Instructions

Hand-in Instructions The project is due on the 7th of October, 2024. Be sure to follow the steps described below:

- Your solution must be comprised of five files: one separate file for each exercise (`Ex1.dfy`, `Ex2.dfy`, `Ex3.dfy`, `Ex4.dfy`, and `Ex5.dfy`). Each Dafny file must be implemented within its own Dafny module.[1]
- Exercises 4 and 5 require the (some of the) methods defined and/or specified in Exercise 3. Use Dafny's `include` and `import` directives to avoid code duplication.
- Create a `zip` file containing the five answer files and upload it in **Fenix**. Submissions will be closed at 23h59 on the 7th of October, 2024. 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).