Dafny coursework exercises

John Wickerson

Autumn term 2024

There are five tasks, involving some programming and some verifying. The tasks appear in roughly increasing order of difficulty, and each is worth 20 marks. Tasks labelled (⋆) are expected to be straightforward. Tasks labelled (⋆⋆) should be manageable but may require quite a bit of thinking. Tasks labelled (⋆⋆⋆) are highly challenging; it is not expected that many students will complete these.

Marking principles. If you have completed a task, you will get full marks for it and it is not necessary to show your working. If you have not managed to complete a task, partial credit may be given if you can demonstrate your thought process. For instance, you might not be able to come up with *all* the invariants that are necessary to complete the verification, but perhaps you can confirm *some* invariants and express (in comments) some of the other invariants that you think are needed but haven't managed to verify.

Unless instructed otherwise, if you are provided with method or function, you should not change the computation that it does. You may freely add ghost code or assertions because these do not affect the computation that the code does. Also, unless instructed otherwise, you should not add extra preconditions to code that you are given, because extra preconditions make code less useful, and a proportionate penalty will be imposed.

Submission process. You are expected to produce a single Dafny source file called `Surname1Surname2.thy`, where `Surname1` and `Surname2` are the surnames of the two students in the pair. This file should contain your solutions to all of the tasks below that you have attempted. You are welcome to show your working on incomplete tasks by decorating your file with `/*comments*/` or `//comments`. Some of the tasks contain questions that require short written answers; these answers can be provided as comments.

Plagiarism policy. You **are** allowed to consult the coursework tasks from previous years – the questions and model solutions for these are available. You **are** allowed to consult internet sources like Dafny tutorials. You **are** allowed to work together with the other student in your pair. Please **don't** submit these programs as questions on Stack Overflow! And please **don't** share your answers to these tasks outside of your own pair.

This year's coursework is centred around verifying a SAT solver.

Task 1 (⋆) This task focuses on *duplicate-free sequences of symbols*. These are sequences where no symbol appears more than once. A symbol is simply an integer.

```
type symbol = int
```

The precise definition of a duplicate-free sequence of symbols is as follows.

```
predicate dupe_free(xs:seq<symbol>)
{
  forall i,j :: 0 <= i < j < |xs|
    ==> xs[i] != xs[j]
}
```

(a) Here is a method that reverses a sequence of symbols. It works in a *recursive* fashion.

```
method rev(xs:seq<symbol>)
returns (ys:seq<symbol>)
requires dupe_free(xs)
ensures dupe_free(ys)
{
  if (xs == []) {
    ys := [];
  } else {
    ys := rev(xs[1..]);
    ys := ys + [xs[0]];
  }
}
```

Prove that if the method is given a duplicate-free sequence as input, then the output of the method is also duplicate-free. You may add any assertions or extra postconditions as required.

(b) Write a method called rev2 that fulfils the same contract as rev, but works in an *iterative* fashion (i.e., using a while-loop rather than recursion). Prove that your method fulfils this contract.

(c) Here is a lemma about concatenating two duplicate-free sequences. It says that if both xs and ys are duplicate-free sequences, then the concatenated sequence xs+ys is also duplicate-free.

```
lemma dupe_free_concat
  (xs:seq<symbol>, ys:seq<symbol>)
requires dupe_free(xs)
requires dupe_free(ys)
ensures dupe_free (xs + ys)
{}
```

Write down values for xs and ys that demonstrate that the lemma is *not* true. Devise an extra precondition to fix the problem, and then prove the lemma.

Task 2 (⋆⋆) This task is about extracting symbols from queries. The query datatype is defined below.

```
type literal = (symbol,bool)
type clause = seq<literal>
type query = seq<clause>
```

That is, a query is a sequence of clauses. Each clause is a sequence of literals. Each literal is a pair comprising a symbol and a **bool**. A symbol is implemented simply as an **int**. As an example: a query like $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$ can be represented using the query datatype as

```
[
  [
    (1, true),
    (2, true)
  ],
  [
    (2, false),
    (3, true)
  ]
]
```

You are given a function called symbols, which extracts the set of symbols that appear in a given query. For instance, the result of calling symbols on the query above is the set {1,2,3}.

You are also given a function called `symbol_seq`, which extracts the duplicate-free sequence of symbols that appear in a given query. For instance, the result of calling `symbol_seq` on the query above is the sequence `[1,2,3]`.

Though these functions are similar, it is useful to have both; `symbols` is convenient for specifying how code works, but `symbol_seq` is needed in the implementation because it is not possible to iterate through a set in Dafny.

You are also given versions of these functions that operate on a single clause. These are called `symbols_clause` and `symbol_seq_clause`, and they are used to define the functions that operate on a whole query.

(a) Here is the implementation of the `symbol_seq_clause` function mentioned above.

```
1  function symbol_seq_clause(c:clause)
2    : seq<symbol>
3  ensures dupe_free(symbol_seq_clause(c))
4  ensures forall x ::
5    x in symbol_seq_clause(c)
6      <==> x in symbols_clause(c)
7  {
8    if c == [] then [] else
9      var x := c[0].0;
10     var c' :=
11       remove_symbols_clause(c[1..], {x});
12     [x] + symbol_seq_clause(c')
13 }
```

It works in a recursive fashion. If the clause `c` is empty, then the result is the empty sequence. Otherwise, the result begins with the first symbol in `c`, say `x`. It ensures that the resulting sequence of symbols is duplicate-free by removing all occurrences of `x` from the rest of the clause before making the recursive call.

Prove that this function always terminates, that its result is always duplicate-free (line 3), and that it contains the same elements as the set produced by the `symbols_clause` function (line 4).

(b) Here is the implementation of the `symbol_seq` function mentioned above.

```
function symbol_seq(q:query) : seq<symbol>
```

```
 2  ensures dupe_free(symbol_seq(q))
 3  ensures forall x ::
 4    x in symbol_seq(q) <==> x in symbols(q)
 5  {
 6    if q == [] then [] else
 7      var xs := symbols_clause(q[0]);
 8      var q' := remove_symbols(q[1..], xs);
 9      symbol_seq_clause(q[0]) + symbol_seq(q')
10  }
```

Prove that this function always terminates, that its result is always duplicate-free (line 2), and that it contains the same elements as the set produced by the `symbols` function (line 3).

Task 3 (⋆⋆) This task is concerned with evaluating a query under a given valuation.

A valuation is a map from symbols to Boolean values.

```
type valuation = map<symbol,bool>
```

The `evaluate_clause` predicate takes a clause `c` and a valuation `r`, and it holds when at least one of the literals in `c` matches an entry in `r`.

```
predicate evaluate_clause(c:clause, r:valuation) {
  exists xb :: (xb in c) && (xb in r.Items)
}
```

For instance, if `c` is the clause `[(2, false), (3, true)]` and `r` is the valuation `1 => false, 2 => false, 3 => false`, then `evaluate_clause(c, r)` is true.

The `evaluate` predicate takes a query `q` and a valuation `r`, and it holds when `evaluate_clause` holds for all of the clauses in `q`.

```
predicate evaluate(q:query, r:valuation) {
  forall i :: 0 <= i < |q| ==>
    evaluate_clause(q[i], r)
}
```

Those two predicates contain **`exists`** and **`forall`** constructions, so although they are useful for *specifying*, they are not suitable for *implementing* because they are not executable. So, below are a pair of methods that evaluate a query under a given valuation in an *executable* way. The `eval_clause` method evaluates a single clause, while the `eval` method evaluates a whole query.

```
method eval_clause (c:clause, r:valuation)
returns (result: bool)
ensures result == evaluate_clause(c,r)
{
  var i := 0;
  while (i < |c|) {
    if (c[i] in r.Items) {
      return true;
    }
    i := i + 1;
  }
  return false;
}

method eval(q:query, r:valuation)
returns (result: bool)
ensures result == evaluate(q,r)
{
  var i := 0;
  while (i < |q|) {
    result := eval_clause(q[i], r);
    if (!result) {
      return false;
    }
    i := i + 1;
  }
  return true;
}
```

Prove that these methods meet their given postconditions; that is, the result of the `eval_clause` method always coincides with the `evaluate_clause` predicate, and the result of the `eval` method always coincides with the `evaluate` predicate.

Task 4 (⋆⋆) Finally, here is the top-level method that performs the actual SAT solving.

```
method naive_solve (q:query)
returns (sat:bool, r:valuation)
ensures sat==true ==> evaluate(q,r)
ensures sat==false ==>
  forall r ::
```

```
6      r in mk_valuation_seq(symbol_seq(q)) ==>
7        !evaluate(q,r)
8  {
9    var xs := symbol_seq(q);
10   var rs := mk_valuation_seq(xs);
11   sat := false;
12   var i := 0;
13   while (i < |rs|)
14   {
15     sat := eval(q, rs[i]);
16     if (sat) {
17       return true, rs[i];
18     }
19     i := i + 1;
20   }
21   return false, [];
22 }
```

The method has two return values: a Boolean value named `sat` together with a `valuation` named `r` that assigns a truth-value to each symbol in the query. If the solver finds the query to be unsatisfiable, it sets `sat` to false. If it finds a valuation that satisfies the query, it sets `sat` to true and sets `r` to the valuation it found.

Prove the postcondition on line 3, which states that if `naive_solve` returns `sat==`**`true`**, it must be the case that the valuation `r` really does satisfy the query.

Also prove the postcondition on line 4, which states that if `naive_solve` returns `sat==`**`false`**, it must be the case that the query is satisfied by none of the valuations that the method tries.[1]

Task 5 (⋆⋆⋆) A major weakness of the `naive_solve` implementation is that it explicitly enumerates all of the possible valuations before iterating through them. If a query contains n symbols, then there will be 2^n possible valuations, which could take up a huge amount of memory if n is large.

So, here is a slightly more efficient SAT solver, called `simp_solve`, which avoids explicitly enumerating all possible valuations.

[1] A better postcondition would be: if `naive_solve` returns `sat==`**`false`**, it must be the case that the query is satisfied by *no valuation at all*, not just none of the valuations that the method tries. This is provable, and amounts to checking that the `mk_valuation_seq` function really does enumerate all possible valuations, but I've deemed it beyond the scope of this coursework.

```
method simp_solve (q:query)
returns (sat:bool, r:valuation)
ensures sat==true ==> evaluate(q,r)
ensures sat==false ==> forall r :: !evaluate(q,r)
{
  if (q == []) {
    return true, map[];
  } else if (q[0] == []) {
    return false, map[];
  } else {
    var x := q[0][0].0;
    sat,r := simp_solve(update_query(x,true,q));
    if (sat) {
      r := r[x:=true];
      return;
    }
    sat,r := simp_solve(update_query(x,false,q));
    if (sat) {
      r := r[x:=false];
      return;
    }
    return sat, map[];
  }
}
```

The method works as follows. Given a query `q`, it does a three-way case split. If `q` has no clauses then it is trivially satisfiable (with the empty valuation). If the first clause in `q` is empty, then `q` is unsatisfiable. Otherwise, it considers the first symbol that appears in `q` – call that `x` – and makes two recursive solving attempts: one with `x` substituted with **`true`**, and one with `x` substituted with **`false`**. If neither recursive attempt succeeds, then `q` is deemed unsatisfiable.

Prove that `simp_solve` terminates and that it meets both of its postconditions.

Hint: here is one lemma that you will likely find helpful as an intermediate step in your proof. If symbol `x` *is not assigned a value by valuation* `r`*, then*

```
evaluate (update_query (x,b,q), r)
  == evaluate (q, r[x:=b])
```

In other words: updating a query under the valuation `x=>b` *is the same as updating the valuation itself and leaving the query unchanged.*