Cracking the Coding Interview in Dafny: Permutations

Nov 8, 2023 - Stefan Zetzsche

Introduction

Coding interviews are notoriously difficult, for many reasons. One may argue (half-seriously) that one issue is that it often is not entirely clear what the actual problem is that ought to be solved, and another is that whether a solution is correct or not is often a matter of negotiation rather than subject to a formal proof. In fact, it is not uncommon to find solutions to interview problems online that after close inspection turn out to be simply wrong. This blog post may be understood as an argument for spending less time on problems whose solutions require smart ideas (that one may or may not come up with on the spot), but instead focus on problems with simple solutions, but non-trivial correctness proofs. At its heart, however, this blogpost is really just a gentle introduction to Dafny, which allows us to write, in a very readable way, the specifications of a problem, its solution, and proof of correctness within the same language.

The Problem

The coding interview problem is described to us in natural language. As we would like to verify the correctness of a potential solution, we need to translate the problem into a formal language.

The Problem in Natural Language

Given a sequence of arbitrary type, return the set that contains all its unique permutations.

The Problem in Formal Language

Intuitively, a permutation of a sequence is a second sequence that contains the same elements as the first one, but potentially in a different order. In other words, after one forgets about the order of the elements, both structures are identical. For example, the sequence `[1,0,0]` is a permutation of the sequence `[0,0,1]`. In Dafny, a sequence without an order can be modelled by the type `multiset`. Multisets are generalizations of sets in the way that they can contain elements multiple times, instead of just once. The multiset that models both of the above sequences is `multiset{0,0,1}`. Note that this multiset is different from the set `set{0,0,1} == set{0,1}`. You may think of multisets that contain elements of type `T` as functions `f: T -> nat`, where `f(t)` specifies how many occurrences of the element `t` the multiset contains. In Dafny, the conversion of a sequence `s` into a multiset is denoted by `multiset(s)`. We define a sequence `p` to be a permutation of a sequence `s` (both with elements of a generic type `T`) if they satisfy the following predicate:

```
predicate IsPermutationOf<T(==)>(p: seq<T>, s: seq<T>) {
  multiset(p) == multiset(s)
}
```

Note that a predicate is a function that returns a `bool` even — that is, either `true` or `false`. To compare two multisets for equality, we need to be able to compare their elements for equality. For this reason, we require the type `T` to have the (in-built) `(==)`. Given above, have the definition of the set that contains all permutations of a sequence `s` is immediate:

```
ghost function AllPermutationsOf<T(==)>(s: seq<T>): iset<seq<T>> {
  iset p | IsPermutationOf(p, s)
}
```

Note that `AllPermutationsOf` returns a potentially infinite set of sequences. While this is not wrong, it's an over-approximation. Think of above [set comprehension](https://dafny.org/latest/DafnyRef/DafnyRef#sec-set-comprehension-expr) as a mathematical definition. Establishing its finiteness via a constructive argument will be part of a potential solution. This view is witnessed by the use of the keyword `ghost`, which implies that the function won't be compiled, as it's meant to be used for static verification purposes only. We restrict ourselves to types `T` with the characteristic `(!new)`, because a set comprehension involved in a function definition is not allowed to depend on the set of allocated references.

We are now able to formally state the problem task: Define the body of a function with the signature

```
function CalculateAllPermutationsOf<T(==)>(s: seq<T>): set<seq<T>>
```

and show that it is correct, in the sense that there exists a proof for the lemma

```
lemma Correctness<T(!new)>(s: seq<T>)
  ensures (iset p | p in CalculateAllPermutationsOf(s)) == AllPermutationsOf(s)
```

The Proposed Solution

There are many potential solutions to the problem. Here, we propose a purely functional approach that defines the set of all unique permutations of a sequence recursively. In the base case, when the sequence `s` has length zero, we simply return the singleton set that contains `s`. In the case that `s` is of length greater than zero, we first recursively calculate the set of all the permutations of the subsequence of `s` that starts at the second element, before inserting, via a separate function `InsertAt`, the first element of `s` into all the permutations in that set, at every possible position.

```
function CalculateAllPermutationsOf<T(==)>(s: seq<T>): set<seq<T>> {
  if |s| == 0 then
    {s}
  else
    set p, i | p in CalculateAllPermutationsOf(s[1..]) && 0 <= i <= |p|
             :: InsertAt(p, s[0], i)
}
```

The function `InsertAt`<T>(s: seq<T>, x: T, i: nat): seq<T>

```
  requires i <= |s|
{
  s[..i] + [x] + s[i..]
}
```

For example, to compute the set of all permutations of the sequence `s := [0,0,1]`, we first compute the set of all unique permutations of the subsequence `s[1..] == [0,1]`, which leads to `{[0,1], [1,0]}`. Then we insert the first element `s[0] == 0` into all the sequences of that set, at every possible position. In this case, the resulting set is `{[0,0,1], [0,1,0], [1,0,0]}`, as indicated below:

```
{
[0] + [0,1], [0] + [0] + [1], [0,1] + [0], // inserting 0 into [0,1]
[0] + [1,0], [1] + [0] + [0], [1,0] + [0]  // inserting 0 into [1,0]
}
```

During a coding interview, the focus would now typically shift to an analysis of the complexity of the algorithm, before concluding with a presentation of an alternative, more efficient solution. Here, we instead will concentrate on proving the correctness of the solution proposed above. In this case, proving the proposed correct is arguably harder than coming up with it in the first place.

The Proof of Correctness

To establish correctness, we need to prove, for all sequences `s`, the equality of the two (infinite) sets `A(s) := iset p | p in CalculateAllPermutationsOf(s)` and `B(s) := AllPermutationsOf(s)`. That is, we have to establish the validity of two implications, for all sequences `p`: i) if `p` is in `A(s)`, then it also is in `B(s)`; and ii) if `p` is in `B(s)`, then it also is in `A(s)`. We prove the two cases in separate lemmas, `CorrectnessImplicationOne(s, p)` and `CorrectnessImplicationTwo(s, p)`, respectively.

```
lemma Correctness<T(!new)>(s: seq<T>)
  ensures (iset p | p in CalculateAllPermutationsOf(s)) == AllPermutationsOf(s)
{
  assert (iset p | p in CalculateAllPermutationsOf(s)) == AllPermutationsOf(s) by {
    assert forall p :: p in CalculateAllPermutationsOf(s) <==> p in AllPermutationsOf(s)
        forall p
        ensures p in CalculateAllPermutationsOf(s) <==> p in AllPermutationsOf(s)
        {
          CorrectnessImplicationOne(s, p);
          CorrectnessImplicationTwo(s, p);
        }
    }
}

lemma CorrectnessImplicationOne<T(!new)>(s: seq<T>, p: seq<T>)
  ensures p in CalculateAllPermutationsOf(s) ==> p in AllPermutationsOf(s)
{...}

lemma CorrectnessImplicationTwo<T(!new)>(s: seq<T>, p: seq<T>)
  ensures p in CalculateAllPermutationsOf(s) <== p in AllPermutationsOf(s)
{...}
```

The rest of this post will be dedicated to filling in the bodies of the lemmas `CorrectnessImplicationOne` and `CorrectnessImplicationTwo`.

Proof of CorrectnessImplicationOne

The first implication is arguably the more intuitive or simpler one to prove. It states that any sequence `p` in the set returned by `CalculateAllPermutationsOf(s)` is indeed a permutation of `s`, that is, it is an element of `AllPermutationsOf(s)`, which means it satisfies the predicate `IsPermutationOf(p, s)`.

As is typical for Dafny, the proof of the lemma follows the structure of the function that it reasons about. In this case, we mimic the function `CalculateAllPermutationsOf` by mirroring its recursive nature as a proof via induction. In particular, this means that we cycle its case-split. In the case that `s` is of length zero (the induction base), Dafny is able to prove the claim automatically:

```
lemma CorrectnessImplicationOne<T(!new)>(s: seq<T>, p: seq<T>)
  ensures p in CalculateAllPermutationsOf(s) ==> p in AllPermutationsOf(s)
{
  if |s| == 0 {
    // induction base, no proof hints needed
  } else {
    // induction step
    if p in CalculateAllPermutationsOf(s) {
      assert p in AllPermutationsOf(s) by {
        assert IsPermutationOf(p, s) by {
          var s', i :| p' in CalculateAllPermutationsOf(s[1..]) && 0 <= i <= |p'|
                       && p == InsertAt(p', s[0], i);
          calc == {
              multiset(p);
              multiset(InsertAt(p', s[0], i));
              { Multiset#InsertAt(p', s[0], i); }
              multiset{s[0]} + multiset(p');
              { CorrectnessImplicationOne(s[1..], p'); } // induction hypothesis
              multiset{s[0]} + multiset(s[1..]);
              multiset{s[0]} + s[1..];
              { assert s[0:1] + s[1..] == s; }
              multiset(s);
            }
        }
      }
    }
  }
}
```

In the case that `s` is of length greater than zero (the induction step), we need to help Dafny a bit. Assuming that `p` is an element in `CalculateAllPermutationsOf(s)`, we aim to show that `multiset(p)` and `multiset(s)` are equal. What information about `p` do we have to derive such an equality? A close inspection of the recursive call in `CalculateAllPermutationsOf` indicates the existence of a second sequence `p'` that lives in `CalculateAllPermutationsOf(s[1..])` and an index `i` of `p'` such that `p == InsertAt(p', s[0], i)`. To transform `multiset(p) == multiset(InsertAt(p', s[0], i))` into something that's closer to `multiset(s)`, we establish the following additional lemma:

```
lemma MultisetAfterInsertAt<T(==)>(x: seq<T>, x': T, i: nat)
  requires i <= |x|
  ensures multiset(InsertAt(x, x', i)) == multiset{x'} + multiset(x)
{
  calc == {
    multiset(InsertAt(x, x', i));
    multiset(s[..i] + [x] + s[i..]);
    multiset(s[..i]) + multiset([x]) + multiset(s[i..]);
    multiset(s[0:i]) + multiset{x'} + multiset(s[i..]);
    multiset{x'} + multiset(s[..i]) + multiset(s[i..]);
    multiset{x'} + multiset(s[..i] + s[i..]);
    { assert s[..i] + s[i..] == s; }
    multiset{x'} + multiset(s);
  }
}
```

The statement of `MultisetAfterInsertAt` is relatively simple: the multiset of a sequence that is obtained by inserting an element `x` into a sequence `s` is the multiset of `s` plus the one-element multiset of the sequence `[x]` of length one. A proof of it can be obtained with a simple equational [calculation](https://dafny.org/latest/DafnyRef/DafnyRef#sec-calculations): we use the definition of `InsertAt`, the fact that `multiset(s + t) == multiset(s) + multiset(t)` for any two sequences `s`, `t`, the commutativity of the join `+` of multisets, and the equality `s[..i] + s[i..] == s`.

To make progress in the induction step of our main proof, we call above lemma with the arguments `MultisetAfterInsertAt(p', s[0], i)`. In consequence, we find that `multiset(InsertAt(p', s[0], i)) == multiset{s[0]} + multiset(p')`. Since `s[1..]` and `p'` are shorter in length than `s` and `p` (which are of same length), respectively, we can use them in a recursive call to `CorrectnessImplicationOne(s[1..], p')`, thus establishing the fact that the permutation `p'` in `CalculateAllPermutationsOf(s[1..])` implies `p' in AllPermutationsOf(s[1..])`. By construction, `p'` is indeed in `CalculateAllPermutationsOf(s[1..])`, thus we can deduce `multiset(p') == multiset(s[1..])` from the definition of a permutation. The remaining steps of the proof are now straightforward (and resemble the arguments in the proof of `MultisetAfterInsertAt`).

Proof of CorrectnessImplicationTwo

The second implication states that any permutation `p` of `s` is an element of `CalculateAllPermutationsOf(s)`. Combined with the first implication, we thus know that `CalculateAllPermutationsOf(s)` doesn't just a return a set that contains some of the permutations of `s`, but in fact all of them.

As with the first implication, the structure of the proof for `CorrectnessImplicationTwo` mimics the definition of `CalculateAllPermutationsOf`. In the case that `s` is of length zero (the induction base), Dafny is again able to prove the claim automatically. In the case that `s` is of length greater than zero (the induction step), we aim to establish that `p` is an element in `CalculateAllPermutationsOf(s)`, if it is a permutation of `s`:

```
lemma CorrectnessImplicationTwo<T(!new)>(s: seq<T>, p: seq<T>)
  ensures p in CalculateAllPermutationsOf(s) ==> p in AllPermutationsOf(s)
{
  if |s| == 0 {
    // induction base, no proof hints needed
  } else {
    // induction step
    if p in AllPermutationsOf(s) {
      assert p in CalculateAllPermutationsOf(s) by {
        var i := FirstOccurrence(p, s[0]);
        var p' := DeleteAt(p, i);
        assert p' in CalculateAllPermutationsOf(s[1..]) by {
          assert p' in AllPermutationsOf(s[1..]) by {
            PermutationBeforeAndAfterDeletion(p, s, i, 0);
          }
          CorrectnessImplicationTwo(s[1..], p');// induction hypothesis
        }
        assert p == InsertAt(p', s[0], i) by {
            InsertAfterDeleteAt(p, i);
        }
      }
    }
  }
}
```

An inspection of the definition of `CalculateAllPermutationsOf` shows that this means we have to construct a permutation `p'` that is in `CalculateAllPermutationsOf(s[1..])` and some index `i` of `p'`, such that `p == InsertAt(p', s[0], i)`. How do we find `p'` and `i`? Intuitively, the idea is as follows: we define `i` as the index of the first occurrence of `s[0]` in `p` (which we know exists, since `p`, as permutation, contains the same elements as `s`) and `p'` as the sequence that one obtains when deleting the `i`-th element of `p`. To formalize this idea, we introduce the following two functions:

```
function FirstOccurrence<T(==)>(p: seq<T>, x: T): (i: nat)
  requires x in multiset(p)
ensures i < |p|
ensures p[i] == x
{
  if p[0] == x then
    0
  else
    FirstOccurrence(p[1..], x) + 1
}

function DeleteAt<T>(s: seq<T>, i: nat): seq<T>
  requires i < |s|
{
  s[..i] + s[i+1..]
}
```

To conclude the proof, it remains to establish that `p` is an element of `CalculateAllPermutationsOf(s[1..])` and `p == InsertAt(p', s[0], i)`.

For `i`, the idea is to apply the induction hypothesis `CorrectnessImplicationTwo(s[1..], p')`, which yields the desired result if `p'` is a permutation of `s[1..]`. Since `p` was a permutation of `s`, and `p'` was obtained from `p` by deleting its `i`-th element, the statement is intuitively true. Formally, we establish it with the call `PermutationBeforeAndAfterDeletionAt(p, s, i, 0)` to the following, slightly more general, lemma:

```
lemma PermutationBeforeAndAfterDeletionAt<T(==)> (p: seq<T>, s: seq<T>, i: nat, j: nat)
  requires IsPermutationOf(p, s)
  requires i < |p|
  requires j < |s|
  requires p[i] == s[j]
  ensures IsPermutationOf(DeleteAt(p, i), DeleteAt(s, j))
{
  assert IsPermutationOf(DeleteAt(p, i), DeleteAt(s, j)) by {
    calc == {
      multiset(DeleteAt(p, i));
      multiset(p[..i] + p[i+1..]);
      multiset(p[..i]) + multiset(p[i+1..]);
      multiset(p[..i]) + multiset([p[i]]) + multiset(p[i+1..]) - multiset([p[i]]);
      multiset(p[..i] + [p[i]] + p[i+1..]) - multiset([p[i]]);
      { assert p[..i] + [p[i]] + p[i+1..] == p; }
      multiset(p) - multiset([p[i]]);
      multiset(s) - multiset([s[j]]);
      { assert s[..j] + [s[j]] + s[j+1..] == s; }
      multiset(s[..j] + [s[j]] + s[j+1..]) - multiset([s[j]]);
      multiset(s[..j]) + multiset([s[j]]) + multiset(s[j+1..]) - multiset([s[j]]);
      multiset(s[..j]) + multiset(s[j+1..]);
      multiset(s[..j] + s[j+1..]);
      multiset(DeleteAt(s, j));
    }
  }
}
```

The proof of `PermutationBeforeAndAfterDeletionAt` is mostly a direct consequence of the involved definitions, and otherwise uses properties of `multiset` that we've encountered before.

To establish `i`, that is, the equality `p == InsertAt(p', s[0], i)`, we recall that by construction `p' == DeleteAt(p, i)`. Substituting `p'` for its definition and using the postcondition `p[i] == s[0]` of `FirstOccurrence` thus leads us to the following lemma, which states that deleting an element of a sequence, before inserting it again at the same position, leaves the initial sequence unchanged:

```
lemma InsertAfterDeleteAt<T(==)> (s: seq<T>, i: nat)
  requires i < |s|
  ensures s == InsertAt(DeleteAt(s, i), i, s[i], i)
{}
```

This concludes our proof!

Conclusion

Using Dafny, we were able to i) translate the informal task of writing a function that returns the set of all permutations of a given sequence into a formal, mathematical specification; ii) implement a potential solution to the problem — in this case, in a purely functional style; and iii) formally verify, with a mathematical proof, that the proposed solution is correct. Along the way, we encountered Dafny's `seq`, `set`, `iset`, and `multiset` types, discussed [type parameters](https://dafny.org/latest/DafnyRef/DafnyRef#sec-type-parameters), proved theorems by induction, reasoned equationally with the `calc` statement, and made heavy use of Dafny's capability to subdivide complex lemmas into smaller, feasible components. I hope you enjoyed the ride. The full Dafny source code of this post is available here.