Lessons of Formal Program Design in Dafny

Ran Ettinger(✉)

Ben-Gurion University of the Negev, Beer Sheva, Israel
ranger@cs.bgu.ac.il

Abstract. Building on the long tradition of program derivation, whereby starting from a formal specification and progressing in small steps of refinement we end-up with correct executable code, this paper presents an approach for teaching that craft using the language and verifier Dafny. Some lessons from the first six years of teaching this material to final-year CS and SE undergraduate students are reported, with emphasis on the merits (and challenges) of using Dafny during live interactive sessions in the classroom.

Keywords: Refinement laws · Specification statement · Auto-active verification · Insertion sort

1 Introduction: About the Course

The textbook "Programming from Specifications" (*PfS*) by Carroll Morgan [8] introduces a student into the world of program derivation in a smooth and formal way. Equipping the novice formal programmer with motivation, some logical background on the predicate calculus, and elementary means known as *laws of refinement* for developing correct imperative programs (using a programming notation based on Dijkstra's guarded commands [2]), the book dedicates its tenth chapter to presenting a case study for developing a first program with nested loops: *insertion sort*.

The insertion sort case study is reformulated, in this paper, using the language and verifier Dafny [5]. This acts as a basis for reporting on some experiences from the first six years of teaching a substantial subset of the *PfS* material in a course entitled "correct-by-construction programming" (*ccpr*[1]). This is an elective course given to final-year CS and SE undergraduate students at Ben-Gurion University. Following *PfS*, the course teaches how to design algorithms and programs that are guaranteed to meet their specification. Starting with a mathematical description of the program's requirements, the course presents a formal method for turning such specifications into actual code, in a stepwise approach known as refinement. Techniques of algorithm refinement are presented, for the derivation of loops from invariants, as well as recursive procedures.

[1] Course website: https://www.cs.bgu.ac.il/~ccpr191.

The developed algorithms are typically very short, but challenging, as we aim to construct correct and efficient code. The programming throughout this course is done in the language Dafny, using its integration into Microsoft Visual Studio [7]. This environment enables the annotation of programs with their specifications. Moreover, it includes an automatic verifier, such that a program can be executed only after its functional correctness has been established (with some potential exceptions to be discussed later in the paper). A switch to Visual Studio Code is currently scheduled for the next iteration of the course, along with some other changes, including the adoption of SPARK/Ada (that has commenced in the 2020 iteration of the course) with its GNATprove verifier and GNAT Programming Studio (GPS) IDE.

The main textbook of this course is "Programming from Specifications" by Carroll Morgan [8] and related material can be found in further sources, including [1–4,6]. A subset of Morgan's laws of refinement is being introduced gradually in the first third of the course[2], through live sessions of program derivation in class. Most programs are taken from Morgan's book. For example, in the first few weeks, we learn how to develop a loop, correctly, by deriving iterative programs for computing a Fibonacci number, the factorial of a natural number, the non-negative floor of the square root of a given natural number (through linear and then binary search). This is followed by the development algorithms to search for an element in a read-only array. Equipped with a basic familiarity of how to design and implement iterative algorithms, using loop invariants, in small and provably-correct steps of refinement, we are ready for the first bigger case study. This will be our first program to update the contents of a given array, and our first derivation of a nested loop.

2 Lessons 10–12: Insertion Sort

The *ccpr* course has been given at BGU in each Fall semester since October 2013. It is made of two sessions a week, of two hours each, over a period of 13 weeks. Approaching the middle of the semester we typically dedicate three sessions to the *PfS* case study of insertion sort [8, Chapter 10]. In the 2019 iteration of the course, reported here, these three sessions commenced on the 10th lecture of the semester. The complete derivation comprising 11 steps of refinement, is given in detail in Figs. 1, 2, 3, 4, 5, 6, 7 and 8. The eventual code is shown in Fig. 9. (As I make frequent references to line numbers of program elements, in what follows, it may be helpful, if possible, to have two copies of the paper in front of you. This is how I evaluate homework submissions: reading the call to a method or lemma in one part of the program, and quickly browsing in the other copy to study its specification, comparing it to the expected specification according to the presently exercised law of refinement).

2.1 Specification for a Sorting Algorithm

A possible specification for sorting an array of integers in a non-decreasing order is shown on lines 1–10 of the program in Fig. 1. Referring to the predicate

[2] https://www.cs.bgu.ac.il/~ccpr191/Laws_Of_Refinement.

```
 1  predicate Sorted(q: seq<int>)
 2  {
 3     ∀ i,j • 0 ≤ i ≤ j < |q| ⟹ q[i] ≤ q[j]
 4  }
 5
 6  method InsertionSort(a: array<int>, ghost A: multiset<int>)
 7     requires multiset(a[..]) = A
 8     ensures Sorted(a[..])
 9     ensures multiset(a[..]) = A
10     modifies a
11  {
12     // Step 1: introduce local variable + strengthen postcondition
13     var i := InsertionSort1(a, A);
14     StrongerPostcondition1(a,i,A);
15  }
16
17  predicate Inv1(a: array<int>, i: nat, A: multiset<int>) reads a
18  {
19     i ≤ a.Length ∧
20     Sorted(a[..i]) ∧
21     multiset(a[..]) = A
22  }
23
24  lemma StrongerPostcondition1(a: array<int>, i: nat, A: multiset<int>)
25     requires Inv1(a,i,A) ∧ i = a.Length
26     ensures Sorted(a[..]) ∧ multiset(a[..]) = A
27  {}
28
29  method InsertionSort1(a: array<int>, ghost A: multiset<int>)
30        returns (i: nat)
31     requires A = multiset(a[..])
32     ensures Inv1(a,i,A) ∧ i = a.Length
33     modifies a
```

Fig. 1. A specification for sorting along with a first step of refinement, reflecting a design for the anticipated outer loop.

`Sorted` from lines 1–4, the postcondition on line 8 expresses the expectation that when exiting `InsertionSort`, the sequence of elements stored in the given array (denoted `a[..]` in Dafny) will be in a non-decreasing order. The fact that this sequence is a permutation of the original contents of the array is expressed here as a combination of the precondition and postcondition on lines 7 and 9 respectively, using the additional parameter `A`. Being a ghost parameter, `A` acts here as a logical constant, storing the bag of values in the given array; in Dafny we achieve this through the type `multiset` (line 6) and the operator with the same name (lines 7 and 9) that collects the bag of values from the sequence of numbers stored in the array. Following Morgan's presentation, we typically start the session by considering how to specify the requirements of sorting, highlighting the need to express the fact that the eventual array contents must be a permutation of the original: in the absence of the postcondition on line 9, a "correct" implementation could possibly set all elements of the array with the value 7. As this session provides a first example of an algorithm that updates the contents of a heap object, we see here for the first time the `modifies` clause, on line 10. In class, I typically start without it, showing how Dafny complains correctly about an assignment to the array, saying it "may update an array element not in the enclosing context's modifies clause". In Morgan's terminology of a specification statement, the key ingredients here (aside from the definitions of the variables and the predicate) are the frame (line 10), the precondition (line 7), and the postcondition (lines 8–9). Morgan's original specification is slightly cleaner in that it expresses the multiset property of lines 7 and 9 as an invariant of the program (with respect to the contents of the array `a` and the constant `A`); to the best of my knowledge, this is not currently supported by Dafny.

Some students feel inclined to add a precondition stating that the array is not empty or that it has at least two elements. They are correct in their observation that below two elements there is no need to do anything. But I try to make it clear that it is against the rules of our game to change the specification. This specification should be seen as a binding contract, between the programmer (them and me, in the classroom) and our invisible client. Should it indeed be helpful to assume the array has at least two elements, they could always start the implementation with an alternation, asking if it has at least two elements in an `if`-statement; the `then` part will call a method whose precondition can explicitly state that the array has at least two elements. I also teach them never to leave an `if`-statement with no `else` part. Instead, we show that the else is redundant using the *skip command* refinement law. Morgan's approach to alternation is more general, explicitly requiring that the precondition will entail the disjunction of all guards of a guarded command.

In contrast to the common practice of (a-posteriori) verification of existing code, we shall develop the code through a process of stepwise refinement. In class, as an exercise, we sometimes agree in advance to aim at the development of specific code. Still it is important to keep in mind the spirit of correct-by-construction programming, with the code and proof being developed side by side. At the end of the process, we will have two versions of the code: the inlined version as shown later in this paper (Fig. 9), and the complete version, comprising 11 methods (Figs. 1, 2, 3, 4, 5, 6, 7 and 8). One advantage of the complete version, in spite of its length, lies in its persistent documentation of the refinement process. A student who missed that class or was unable to follow the interactive development, would ideally be able to reconstruct the full process from the published final version[3].

2.2 Refinement Steps 1–5: The Outer Loop

In the first five steps of refinement we develop a loop for successive insertion of elements into their sorted location in the prefix of the array. This process acts as a derivation of a precise specification for the anticipated `Insert` operation, to move the next element into its correct (sorted) location in the prefix to its left. As in *PfS*, this example is the first in the course in which we end-up with a nested loop. The code for the nested loop itself will be developed subsequently, in refinement steps 6–11 below, starting from the derived specification for the `Insert` operation.

[3] Final version of the insertion sort algorithm from the 2019 iteration of the *ccpr* course (including detailed proofs for the human reader): https://www.cs.bgu.ac.il/~ccpr191/wiki.files/CCPR191-InsertionSort-complete-10Dec18.dfy.

The first step of refinement, shown in Fig. 1, takes the original specification of `InsertionSort` (lines 6–10) and implements it by providing a method body (lines 11–15). This is our course's form of expressing refinement in Dafny. Whenever the refined program involves yet-to-be-implemented specification statements, additional methods are being specified (here `InsertionSort1` on lines 29–33), and can already be invoked (line 13), leaving their implementation for later refinement steps. This first step introduces the local variable `i`, to act as a loop index, and strengthens the postcondition in anticipation for the loop. For this step to be correct, we have an obligation to prove that the new postcondition (line 32) is indeed stronger than the original postcondition (lines 8–9). A convenient way to document such proof obligations in Dafny is through the specification of a lemma (lines 24–26). The generation of this specification is taught as a mechanical process of copying-and-pasting: the new postcondition (line 32) acts as the lemma's precondition (line 25); the older postcondition (lines 8–9) acts as the lemma's postcondition (line 26); and all relevant variables are sent as parameters.

In technical terms, the lemma acts as a ghost method, with no side effect, and in this case with value parameters only. Seeing an invocation of the lemma (line 14), Dafny takes the responsibility to verify that the lemma's precondition holds; in this case Dafny trusts that it does hold, as the call immediately follows the invocation of `InsertionSort1` (line 13) whose postcondition is, by design, the lemma's precondition. And then Dafny assumes that on return from the lemma, its postcondition holds, which is again by design the original postcondition of `InsertionSort`. And hence Dafny has no reason to complain that the postcondition of `InsertionSort` might not hold. In this sense, Dafny trusts its user to prove at some point in the development that the lemma is correct. In class we sometimes leave the lemma unproved at first, just as we do with specifications of further methods, leaving their development for a later step. In this case, however, Dafny gets convinced of the correctness of this lemma with no need for proof. This is the meaning of the lemma's empty body (line 27). Had Dafny been unable to prove correctness of the lemma, it would have complained that a postcondition of the lemma might not hold.

What is it that makes the lemma correct in this case? Following *PfS*, the designed loop invariant `Inv1` expresses the expectation that the index `i` does not exceed the size of the array (line 19), and that the first `i` elements are sorted (line 20). The fact the loop invariant and the negation of its guard hold (line 25), ensures that the first `a.Length` elements (hence the entire array) are sorted. And the second conjunct of the lemma's postcondition directly follows from the third conjunct of the loop invariant (line 21), stating that the multiset of values in the array is indeed the expected multiset, as stored in `A`.

In logical terms, such a lemma, formulated with input parameters only, expresses what Morgan refers to as entailment [8]: the expectation that for all values of the input parameters, according to their types, the result of the lemma's precondition implies the result of the postcondition. In other words, for all values on which the precondition holds, the postcondition must hold too. (Output parameters from a lemma add an existential portion to the formula, that there exist values of these parameters, for which the implication holds).

At the end of this first step of refinement, as said, we are left to continue the development by implementing method `InsertionSort1`. Its specification has been derived by that of `InsertionSort` with two differences: the postcondition has been strengthened, as discussed above, and the frame has been extended (line 30) to accommodate modifications to the value of the loop index, `i`. Using output parameters from methods through the `returns` construct (line 30), along with the `modifies` clause (line 33) is our way of expressing Morgan's frame in Dafny. And adding `i` to the frame here is a direct effect of Morgan's refinement law called *introduce local variable.*

```
29  method InsertionSort1(a: array<int>, ghost A: multiset<int>)
30          returns (i: nat)
31      requires A = multiset(a[..])
32      ensures Inv1(a,i,A) ∧ i = a.Length
33      modifies a
34  {
35      // Step 2: sequential composition + contract frame
36      i := InsertionSort2a(a,A);
37      i := InsertionSort2b(a,i,A);
38  }

40  method InsertionSort2a(a: array<int>, ghost A: multiset<int>)
41          returns (i: nat)
42      requires A = multiset(a[..])
43      ensures Inv1(a,i,A)

55  method InsertionSort2b(a: array<int>, i0: nat, ghost A: multiset<int>)
56          returns (i: nat)
57      requires Inv1(a,i0,A)
58      ensures Inv1(a,i,A) ∧ i = a.Length
59      modifies a
```

Fig. 2. Sequential composition: establish the invariant first and only then get to the loop.

In a second step of refinement, as further preparation for the loop, we decompose the implementation (of method `InsertionSort1`) into a sequence of two operations, as can be seen on lines 36–37 of Fig. 2. The first operation will establish the loop invariant (as can be witnessed in its specification on line 43), and the second operation will be the loop itself. The postcondition of the first operation in a sequential composition, according to the simplest version of this law of refinement, may act as the precondition to the second operation. Whenever we aim for a loop, as we do here, we choose the loop invariant to be this property (lines 43 and 57). Note however that in the precondition of `InsertionSort2b` we refer to `i0` rather than `i`. This is our way of implementing parameter passing to variables in the frame: according to the common convention, we append the digit `0` to the name of a variable whose initial value is required and whose value may be modified in the method. In contrast to Morgan, each refinement may introduce new scopes for variables, and accordingly, the initial variable (such as i0 here) is a genuine parameter, not merely a (ghost) logical constant. While these variables and assignment statements could be seen to have negative impact on the performance of the derived implementation, it is good to recall that by collecting the code at the end of the refinement process, inlining all method bodies, such variables can be removed.

As we anticipate that modifications to the array's contents will be performed only in the loop body, we express this decision explicitly by removing a from the frame of the initialization method, leaving only the loop index in its frame (line 41). This is a refinement step known as *contract frame*.

```
40 method InsertionSort2a(a: array<int>, ghost A: multiset<int>)
41       returns (i: nat)
42    requires A == multiset(a[..])
43    ensures Inv1(a,i,A)
44 {
45    // Step 3: assignment
46    LemmaInsertionSort2a(a,A);
47    i := 0;
48 }
49
50 lemma LemmaInsertionSort2a(a: array<int>, A: multiset<int>)
51    requires A == multiset(a[..])
52    ensures Inv1(a,0,A)
53 {}
```

Fig. 3. A first example of assignment: the proof obligation resembles the original specification, with substitution (of the assignment's LHS by its RHS) performed on the postcondition.

In a third step of refinement, shown in Fig. 3, we choose to implement InsertionSort2a, presenting a first assignment statement, to initialize the outer loop index. The proof obligation of an assignment statement is expressed as a lemma specification (lines 50–52). The lemma states that the precondition entails a modified version of the postcondition, obtained by substituting the assignment's left-hand side with the corresponding right-hand side. Here, starting with a copy of the postcondition of InsertionSort2a, the loop index i has been substituted by 0 in the lemma's postcondition (line 52, compared to line 43). The lemma's precondition (line 51) in such cases remains unchanged (as in line 42). As the correctness of this lemma is proved by Dafny with no difficulties, we *implement* it immediately with an empty body (line 53). This is the first step that introduces no further specifications: the refined version is executable code.

Shown in Fig. 4, the fourth step of our refinement session introduces the outer loop. As the first refinement of a specification with initial variables, we see here for the first time a convention of copying the initial value to the output variable (line 61). As in some of the previously demonstrated laws, *iteration* requires no proof obligation. Instead, we must be sure to start with a specification that expresses the loop invariant in its precondition (line 57, using the initial variable `i0`) and its postcondition must be phrased as a conjunction of the loop invariant and the negation of the loop guard (line 58). Following Morgan's iteration law, the specification of the loop body should express the loop invariant and the loop guard in its precondition (line 73, again with initial variables), and the postcondition (line 74) must involve both the loop invariant and an indication that the loop variant is strictly decreasing, yet not below some lower bound (typically chosen to be `0`); the frame of the loop body remains unchanged (lines 72 and 75).

In class, it is helpful to see how commenting out the first conjunct of the loop body's postcondition on line 74 leads to an error reported on line 64: "This loop invariant might not be maintained by the loop". Alternatively, commenting out the *second* conjunct on line 74 (involving termination of the loop) leads to an error reported on line 63, stating that the "decreases expression might not decrease". In contrast to that, I sometimes forget to include the guard in the loop body's precondition (line 73), and we get no error; only later in the development we come to notice its absence and learn to appreciate its significance: such a specification would be *infeasible* as the result of the precondition not being strong enough here is that *there exists no value* for the output parameter `i` that satisfies the postcondition.

```
55  method InsertionSort2b(a: array<int>, i0: nat, ghost A: multiset<int>)
56          returns (i: nat)
57      requires Inv1(a,i0,A)
58      ensures Inv1(a,i,A) ∧ i = a.Length
59      modifies a
60  {
61      i := i0;
62      // Step 4: iteration
63      while i ≠ a.Length
64          invariant Inv1(a,i,A)
65          decreases a.Length-i
66      {
67          i := InsertionSort3(a, i, A);
68      }
69  }
70
71  method InsertionSort3(a: array<int>, i0: nat, ghost A: multiset<int>)
72          returns (i: nat)
73      requires Inv1(a,i0,A) ∧ i0 ≠ a.Length
74      ensures Inv1(a,i,A) ∧ 0 ≤ a.Length-i < a.Length-i0
75      modifies a
```

Fig. 4. The outer loop: the specification of the loop body is mechanically derived with copies of the invariant (twice), the guard, and the variant function.

The loop body is expected to make two changes: it should increment the loop index and it must insert the next element into its sorted location in the growing prefix of the array. Focusing on the loop index first, our fifth step of refinement, shown in Fig. 5, reflects a decision to increment `i` at the end of the loop body. This step, known as *following assignment*, is quite simple to perform. The specification of method `Insert` (lines 83–86) reflects the expectations from the remaining part of the loop body (line 79, to be followed both in the program text and execution time by the assignment to `i` on line 80) is nearly identical to the specification of the loop body (lines 71–75), with only a few differences.

The single update due to the *following assignment* law causes each reference of `i` in the postcondition to be substituted by `i+1` (line 85). Since we anticipate no further changes to `i`, we remove it from the frame, causing one subsequent change, replacing `i0` by `i`. It is important to note the order here: first substitution (of `i` only, not of `i0`) then rename of `i0` back to `i`. At the end of this modification, the variant-related part of the postcondition becomes trivially true: the `a.Length-i < a.Length-i0` is now the obviously correct condition `a.Length-(i+1) < a.Length-i` and the `0 <= a.Length-i` is now `0 <= a.Length-(i+1)`, which is equivalent to the first conjunct of the loop invariant (line 19 on Fig. 1), applied here in line 85 to `i+1`. So we do not repeat this (by-now-redundant) part in the postcondition of `Insert` (line 85). Indeed, it is frequently the case that this combination of *following assignment* and *contract frame* makes the variant portion of the loop body's postcondition trivially true.

```
71  method InsertionSort3(a: array<int>, i0: nat, ghost A: multiset<int>)
72          returns (i: nat)
73      requires Inv1(a,i0,A) ∧ i0 ≠ a.Length
74      ensures Inv1(a,i,A) ∧ 0 ≤ a.Length-i < a.Length-i0
75      modifies a
76  {
77      i := i0;
78      // Step 5: following assignment + contract frame
79      Insert(a,i,A);
80      i := i+1;
81  }
82
83  method Insert(a: array<int>, i: nat, ghost A: multiset<int>)
84      requires Inv1(a,i,A) ∧ i ≠ a.Length
85      ensures Inv1(a,i+1,A)
86      modifies a
```

Fig. 5. Updating the loop index and deriving a specification for the remaining computation (the insert operation).

2.3 Refinement Steps 6–10: The Inner Loop

In the second session dedicated to insertion sort, we get to the development of the inner loop. This is more challenging, compared to the derivation of the outer loop, mostly due to the need to change the contents of the array. Accordingly, the loop invariant, the proof obligations, and the proof itself might all be more complicated. The first step in the development of this inner loop is shown in

```
 83 method Insert(a: array<int>, i: nat, ghost A: multiset<int>)
 84    requires Inv1(a,i,A) ∧ i ≠ a.Length
 85    ensures Inv1(a,i+1,A)
 86    modifies a
 87 {
 88    // Step 6: introduce local variable + strengthen postcondition
 89    var j := Insert1(a,i,A);
 90    StrongerPostcondition2(a,i,j,A);
 91 }
 92
 93 predicate SortedExceptAt(q: seq<int>, k: nat)
 94 {
 95    ∀ i,j • 0 ≤ i ≤ j < |q| ∧ i ≠ k ∧ j ≠ k ⟹ q[i] ≤ q[j]
 96 }
 97
 98 predicate Inv2(q: seq<int>, i: nat, j: nat, A: multiset<int>)
 99 {
100    j ≤ i < |q| ∧
101    SortedExceptAt(q[..i+1],j) ∧
102    (∀ k • j < k ≤ i ⟹ q[j] < q[k]) ∧
103    multiset(q) = A
104 }
105
106 predicate method InsertionGuard(a: array<int>, i: nat, j: nat,
107       ghost A: multiset<int>)
108    requires Inv2(a[..],i,j,A)
109    reads a
110 {
111    1 ≤ j ∧ a[j-1] > a[j]
112 }
113
114 lemma StrongerPostcondition2(a: array<int>, i: nat, j: nat, A: multiset<int>)
115    requires Inv2(a[..],i,j,A) ∧ ¬InsertionGuard(a,i,j,A)
116    ensures Inv1(a,i+1,A)
117 {}
118
119 method Insert1(a: array<int>, i: nat, ghost A: multiset<int>)
120       returns (j: nat)
121    requires Inv1(a,i,A) ∧ i ≠ a.Length
122    ensures Inv2(a[..],i,j,A) ∧ ¬InsertionGuard(a,i,j,A)
123    modifies a
```

Fig. 6. Preparation for the insertion loop, defining a loop invariant and a guard, this time in its own *predicate method*, aiming for enhanced clarity of annotations.

Fig. 6. Recalling the definition of the outer loop invariant (`Inv1` on lines 17–22 of Fig. 1), the specification of `Insert` (lines 83–86) could be interpreted as saying that given a state in which the first `i` elements in an array are sorted and there is at least one more element to sort, namely `a[i]`, we wish to *insert* it into its correct location such that the first `i+1` elements will be sorted. (The specification also says that we must maintain the existing elements in the array; confining array modifications to swapping pairs of elements will satisfy this requirement.)

To explore the definition of the inner loop invariant (`Inv2` on lines 98–104, using an additional predicate on lines 93–96) and the definition of the loop guard, expressed in its own `predicate method` (lines 106–112) such that it can be used both in executable code and in annotations, it is helpful to consider the specification of lemma `StrongerPostcondition2` (lines 114–116, invoked on line 90). Fortunately again, this lemma is proved by Dafny, hence the empty curly braces (line 117) for its proof. In words, following Morgan's design, this is indeed true since when the loop invariant holds and the loop guard does not (line 115), the state is such that the first `i+1` elements are sorted except at index `j` (line 101) and `a[j]` is sorted (among the first `i+1` elements) too; the latter is true thanks to a healthy combination of the loop invariant and the negation of the guard: the inserted element, at location `j`, is guaranteed to be sorted to its right thanks to the loop invariant (line 102) and it is guaranteed to be sorted to its left thanks to the negation of the guard (from line 111), since at that state either it is the leftmost element, or it is not smaller than the element to its left, which along with the loop invariant (line 101 again, taken together with line 100) means that the inserted element (at index `j`) is indeed greater-or-equal all elements to its left.

As can be guessed by reading the loop guard, we are aiming for a loop body that repeatedly swaps the inserted element with the element to its left, until it reaches its expected location (either when there are no more elements on its left, in case it is the smallest, or when the element to its left is not larger). It is instructive to see here how the loop invariant records key properties from the history of the computation. Failing to record in the loop invariant (line 102) the fact that at each iteration, and most importantly at the end of the last (line 122), all the *previously* considered elements which are *currently* placed to the right of the inserted element are greater than the inserted element. Commenting out this property, removing it from the loop invariant (line 102), immediately leads to failure in the proof attempt of lemma `StrongerPostcondition2` (line 117). In homework assignment submissions, it is not uncommon to find a comment attached to such an unproved lemma, *waving hands* about what is expected to be true at the point of lemma invocation (line 90 in this case, after the loop). My response in such cases is that the separation of concerns in our proof method is such, that the lemma reflects a logical property that stands by itself; if proven correct (along with separate proofs for all the other obligations), it guarantees that the program satisfies its specification; yet when the lemma by itself is logically incorrect, I simply try to provide a counterexample, in this case with a smaller element to the right of the inserted one; students might argue that my counterexample does not make sense, and that at the end of the loop we will never find such smaller elements to the right of the inserted element; and indeed the fact that we are unable to prove correctness does not necessarily imply that our code is incorrect; it simply means we need to try harder, for example by strengthening the loop invariant, recording there more information from the history of the computation. To such students, it may be helpful to see here on Fig. 6 that the question of whether the loop invariant and the negation of its guard imply for all states that the postcondition of the loop holds can be addressed even before we have implemented the loop.

The development of the inner loop itself is documented in the steps 7–10 of our refinement scenario, as shown in Fig. 7, culminating in a specification for the final operation, of swapping two adjacent elements of the array (lines 173–176). It follows the same line as steps 2–5 of the outer loop: *sequential composition* with *contract frame*, *assignment*, *iteration*, and then *following assignment* with *contract frame*. With Morgan's rich repertoire of refinement laws there is a variety of paths for deriving the same eventual code. Indeed, in class we cover some

```
method Insert1(a: array<int>, i: nat, ghost A: multiset<int>)
      returns (j: nat)
   requires Inv1(a,i,A) ∧ i ≠ a.Length
   ensures Inv2(a[..],i,j,A) ∧ ¬InsertionGuard(a,i,j,A)
   modifies a
{
   // Step 7: sequential composition + contract frame
   j := Insert2a(a,i,A);
   j := Insert2b(a,i,j,A);
}

method Insert2a(a: array<int>, i: nat, ghost A: multiset<int>)
      returns (j: nat)
   requires Inv1(a,i,A) ∧ i ≠ a.Length
   ensures Inv2(a[..],i,j,A)
{
   // Step 8: assignment
   LemmaInsert2a(a,i,A);
   j := i;
}

lemma LemmaInsert2a(a: array<int>, i: nat, A: multiset<int>)
   requires Inv1(a,i,A) ∧ i ≠ a.Length
   ensures Inv2(a[..],i,i,A)
{}

method Insert2b(a: array<int>, i: nat, j0: nat, ghost A: multiset<int>)
      returns (j: nat)
   requires Inv2(a[..],i,j0,A)
   ensures Inv2(a[..],i,j,A) ∧ ¬InsertionGuard(a,i,j,A)
   modifies a
{
   j := j0;
   // Step 9: iteration
   while InsertionGuard(a,i,j,A)
      invariant Inv2(a[..],i,j,A)
      decreases j
   {
      j := Insert3(a,i,j,A);
   }
}

method Insert3(a: array<int>, i: nat, j0: nat, ghost A: multiset<int>)
      returns (j: nat)
   requires Inv2(a[..],i,j0,A) ∧ InsertionGuard(a,i,j0,A)
   ensures Inv2(a[..],i,j,A) ∧ j < j0
   modifies a
{
   j := j0;
   // Step 10: following assignment + contract frame
   Swap(a,i,j,A);
   j := j-1;
}

method Swap(a: array<int>, i: nat, j: nat, ghost A: multiset<int>)
   requires Inv2(a[..],i,j,A) ∧ InsertionGuard(a,i,j,A)
   ensures Inv2(a[..],i,j-1,A)
   modifies a
```

Fig. 7. Four steps of refinement in the development of the inner loop, deriving a specification for the swap operation. Note the similarity to steps 2–5.

more laws, but still quite a small subset of the original catalog. (The additional laws we *do* cover include *alternation*, *skip command*, *leading assignment* and *weaken precondition*.)

```
173 method Swap(a: array<int>, i: nat, j: nat, ghost A: multiset<int>)
174     requires Inv2(a[..], i, j, A) ∧ InsertionGuard(a, i, j, A)
175     ensures Inv2(a[..], i, j-1, A)
176     modifies a
177 {
178     // Step 11: assignment
179     LemmaSwap(a, i, j, A);
180     a[j-1], a[j] := a[j], a[j-1];
181 }
182
183 lemma LemmaSwap(a: array<int>, i: nat, j: nat, A: multiset<int>)
184     requires Inv2(a[..], i, j, A) ∧ InsertionGuard(a, i, j, A)
185     ensures Inv2(a[..][j-1 := a[j]][j := a[j-1]], i, j-1, A)
186 {}
```

Fig. 8. One last step of refinement, swapping the inserted element with the (larger) array item on its left. Note the "sequence assignment" in the proof obligation.

2.4 A Final Step of Refinement: Swapping Adjacent Array Elements

Our last step of refinement, performed in the third and final session dedicated to insertion sort, is shown in Fig. 8. This final step is particularly interesting in the way its proof obligation uses sequence assignment in the lemma specification. It is for the purpose of this substitution that we expressed the inner loop invariant as a predicate that expects a sequence rather than an array of integers, as one of its parameters (line 98 on Fig. 6). According to the proof obligation for the *assignment* law of refinement, note how the specification of `LemmaSwap` is similar to that of `Swap`, except that the frame is empty, and in the postcondition (line 185), only the `a[..]` has been substituted at two locations, based on the LHS of the multiple assignment, with values from its RHS. Magically, as was the case with all prior lemma specifications in this derivation of insertion sort, this lemma too is proved by Dafny (line 186). Indeed, when the first `i+1` elements are sorted except at `j`, and `a[j]` is greater-or-equal all elements to its right, yet it is smaller than `a[j-1]`, swapping them (`a[j]` and `a[j-1]`) generates a sequence in which the first `i+1` elements are sorted except at `j-1` and `a[j]` in its new location is indeed smaller-or-equal all elements to its (new) right, as these are the elements right of `j` as well as `j-1`, now at location `j`.

In class, it is actually only in this third session that we transform the inner loop invariant to take a sequence, rather than the array, as a parameter. This enables the expression of the proof obligation for the swap assignment using sequence assignments. In retrospect, sending a sequence rather than the array could be a more appropriate choice for the outer loop invariant too. This way, there would be no need to explain Dafny's `reads` frame (line 17 in Fig. 1), which is not present in Morgan's approach.

One more transformation we typically perform on the third session is of the guard of the inner loop. Following Morgan, we initially express this guard using an existential quantifier (that at least one of the first `j` elements of the array is larger than the inserted value), and at this stage we replace it with the more efficient guard as shown here in the paper. We use a lemma to demonstrate that when the loop invariant holds, these two formulations of the guard are equivalent.

In conclusion of this session, we observe that 11 steps of refinement were performed, developing executable code that is now scattered in 11 methods and one predicate method. Inlining these methods, in order to collect the code, would yield the version shown in Fig. 9.

```
method {:verify false} InsertionSort_TheCode(a: array<int>,
        ghost A: multiset<int>)
    requires multiset(a[..]) = A
    ensures Sorted(a[..])
    ensures multiset(a[..]) = A
    modifies a
{
    var i := 0;
    while i ≠ a.Length
    {
        var j := i;
        while 1 ≤ j ∧ a[j-1] > a[j]
        {
            a[j-1],a[j] := a[j],a[j-1];
            j := j-1;
        }
        i := i+1;
    }
}
```

Fig. 9. Collecting the correct-by-construction code at the end of the refinement process.

In the version of insertion sort we have just completed developing, as it turns out, we were somewhat lucky that each lemma was proved with no need for manual intervention. To appreciate this, suppose we were to define the predicate `SortedExceptAt` not as we did (on lines 93–96, Fig. 6), but rather in the following equivalent way: `k < |q| && Sorted(q[..k]+q[k+1..])`. As a result, we would get errors both in lemma `StrongerPostcondition2` and in `LemmaSwap`, stating that "A postcondition might not hold". In such cases I encourage my students to spend some time in trying to prove correctness, yet not too much time. The official *order* is *not to fight* Dafny, as we do not learn how Dafny works. We return to discuss this challenge in the next section, reporting on the final homework assignment for this course.

3 Assessment

The final grade in the 2019 iteration of the *ccpr* course was determined by one homework assignment (20%), a must-pass midterm examination (20%), and a final assignment (60%). The assignments were performed by teams of at most three members. The first assignment[4] involved two exercises: (1) binary search, and (2) search for two elements in a sorted sequence of integers whose sum is a given number. The midterm examination was (for the third year running) a multiple-choice quiz[5]. The final assignment[6] involved three exercises: (1) merge sort; (2) inserting an element to a maximum-heap data structure; and (3) inserting an element into a binary-search tree. Of those, the `HeapInsert` seemed to be the trickiest. Here is the specification for this exercise:

```
predicate AncestorIndex(i: nat, j: nat) decreases j-i
{
   i = j ∨ (j > 2*i ∧
   ((AncestorIndex(2*i+1, j) ∨ AncestorIndex(2*i+2, j))))
}

predicate hp(q: seq<int>, length: nat)
   requires length ≤ |q|
{
   ∀ i,j • 0 ≤ i < length ∧
   0 ≤ j < length ∧ AncestorIndex(i, j) ⟹ q[i] ≥ q[j]
}

method HeapInsert(a: array<int>, heapsize: nat, x: int)
   requires heapsize < a.Length
   requires hp(a[..], heapsize)
   ensures hp(a[..], heapsize+1)
   ensures multiset(a[..heapsize+1]) = multiset(old(a[..heapsize])+[x])
   modifies a
```

The `HeapInsert` challenge was not given in isolation. The most involved program we develop during the semester is based on one more case study from *PfS*, for Heap Sort [8, Chapter 12]. Our complete solution to Heap Sort[7] contains nearly 500 lines of non-blank-non-comment executable code and annotations. Most of the ingredients for deriving a correct heap-insert algorithm were available in the heapsort solution; the goal of this exercise was to encourage the students to read the complete solution more closely; yet the development of fully verified solutions was beyond my expectations. Accordingly, the assignment description included the following text: "The submitted programs are expected to compile and verify with no errors (except perhaps for lemma specifications annotated with a {`:verify false`}, whose body is left empty) in Dafny 2.2.0. The correctness of all your non-proved ({`:verify false`}) lemma specifications should be made clear by a verbal comment, explaining why for all values of its parameters (according to their types), if the lemma's precondition holds then its postcondition must clearly hold too. Recall that for the lemma to be correct, this form of logical implication MUST hold by itself, independently of properties known to the reader from any other part of the program. Please note that

[4] https://www.cs.bgu.ac.il/~ccpr191/Assignments/Assignment_1.
[5] https://www.cs.bgu.ac.il/~ccpr191/Previous_Exams.
[6] https://www.cs.bgu.ac.il/~ccpr191/Assignments/Final_Assignment.
[7] https://www.cs.bgu.ac.il/~ccpr191/wiki.files/CCPR191-HeapSort-complete-30Dec18.dfy.

some of the properties required for completion of the development might be very difficult to prove in a formal way (as can be witnessed for example in the published `HeapSort` solution). In each such case you are indeed highly encouraged to formulate an appropriate lemma, explain to the human reader the reason for its correctness, and then leave the lemma unverified in the form stated above."

Table 1. Levels of success: an algorithm to insert an element into a maximum-heap.

Success level of correctness proof	SGs
Fully verified	8
Fully verified inconsistently: on occasion, the proof of one lemma fails	1
Perfectly convincing {`:verify false`} lemma specifications	0
Mostly well-argued {`:verify false`} lemma specifications	6
Badly-argued (probably correct) {`:verify false`} lemma specifications	3
Logically-incorrect {`:verify false`} lemma specifications	9
Seemingly correct code with {`:verify false`} methods	2
Incorrect implementation	3
Did not submit a solution to this portion of the final assignment	2

Table 1 provides a summary of the level of correctness and proof achieved by the 78 course participants, who teamed-up as 34 Submission Groups (SGs). I was encouraged and impressed by the fact that 8 submissions were fully verified. The 9 submissions with logically-incorrect lemma specifications show that there is certainly room for improvement, in my teaching. And perhaps more importantly, I would hope to improve the approach—possibly adopting ideas from Leino's "Program Proofs" (draft) book [6]—in a way that will help upgrade the 6 submissions on the fourth line to the empty third line. I believe that the simpler it would become, for the students, to provide proofs in which the only unproved properties will be easy to explain, to the human reader, some of the students in the 8 teams of the top row would settle for that third row. This will have saved them the time and energy of "fighting" with a theorem prover.

Performance of students on the heap-insert exercise, as reported above, may hopefully raise some optimism: perhaps it is not too late to introduce students to formal methods on the final year of their undergraduate studies (even though it is definitely advisable to start much earlier [9]), and hopefully new generations of practitioners (and of teachers) with skill and experience in formal program design could be raised this way.

References

1. Backhouse, R.: Program Construction: Calculating Implementations from Specifications. Wiley, New York (2003)
2. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Hoboken (1976)
3. Gries, D.: The Science of Programming. Springer, Heidelberg (1987)
4. Kaldewaij, A.: Programming: The Derivation of Algorithms. Prentice-Hall Inc., Upper Saddle River (1990)
5. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
6. Leino, K.R.M.: Program Proofs. Lulu (2020)
7. Leino, K.R.M., Wüstholz, V.: The Dafny integrated development environment. In: F-IDE. EPTCS, vol. 149, pp. 3–15 (2014)
8. Morgan, C.: Programming from Specifications, 2nd edn. Prentice Hall International (UK) Ltd., Hertfordshire (1994)
9. Morgan, C.: (In-)formal methods: the lost art - a users' manual. In: Liu, Z., Zhang, Z. (eds.) SETSS 2014. LNCS, vol. 9506, pp. 1–79. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29628-9_1